École normale supérieure de Lyon
Magistère d'informatique et modélisation
2nd year internship

High Bandwidth Data Transfer with OPIOM & Myrinet: Application to Remote Video

Myricom
Arcadia, 91006 CA — United States

Brice Goglin

June — August 2001

with Patrick Geoffray

Introduction

Myricom, Inc. is a privately held California Corporation founded in 1994. It produces network interfaces and switches both for clusters and LANs, and also specialized, high-speed, computing and communication components for OEM and military-system companies. Myricom is today's world leader in high performance interconnects for cluster computing.

Since Myrinet interfaces possess their own built-in memory and high performance chipset which are accessible to the operating system, their use offers great versatility to the user.

GM is a message-passing system for Myrinet networks. Users now often use the GM driver which allows them to communicate reliably on the network with very high bandwidth and low latency.

This high performance hardware and software allows us to implement protocols in the Myrinet interface instead of working in the kernel. In this way we can use the DMA transfer between PCI devices without wasting time by going through the host.

The goal of my work (internship) is to provide a fast NFS server over Myrinet. Due to time limitations, I used my GM modifications for remote video. I will describe my GM additions for fast data sending from disks to the network and then to a remote frame buffer.

Chapter 1

Overview of Myrinet

1.1 Myrinet hardware

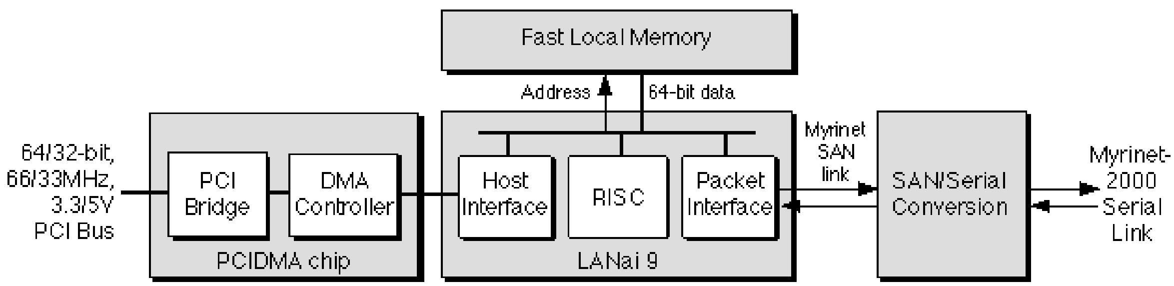

Figure 1.1: Myri 2000 Serial Link/PCI interface.

The third and latest generation of Myricom products is called *Myrinet 2000*. The PCI network interface (see figure 1.1) is based on a processor called *LANai*. In its lastest version, LANai 9, the RISC core runs up to 200 MHz.

It possesses up to 8 MB of SRAM (Static RAM, used for cache memory) on the interface which can be accessed with 64-bit addressing and a constant latency of 2 cycles, for a total memory bandwidth of 1.6 GB/s. This memory is recognized by the BIOS as PCI device DMAable memory. It is also mapped in the physical memory and can be accessed by the operating system as regular memory.

A DMA engine provides the capability to copy data between the host and the LANai memory over the PCI bus. The packet interface supports 2+2 Gigabit/s full-duplex (2 Gb/s for input and 2 Gb/s for output) transfer over the Myrinet link. Moreover the latency of the communications can be as low as a few μs.

Myricom also provides commutation elements based on its *XBar* chipset. This full-crossbar supports 16 ports communicating at full speed in both directions. A set of line cards may be assembled in a switch enclosure to connect up to 128 ports as a Clos topology, guaranteeing the full bissection over the whole switch. The connection between interfaces and switches uses fiber cables (up to 200m) or short distance copper links.

1.2 Myrinet drivers: from BIP to GM

The openess of the Myrinet design led to a profusion of firmware implementations. BIP is one of the earliest implementation, realized at the University of Lyon. It was a small but efficient prototype which provided limited functionality. BIP's design sacrifices the support for multi-user, reliability and security to provide very high throughput and ultra low latency with Myrinet.

GM is a more ambitious implementation developed and supported by Myricom. GM is designed to be used by real customers on production machines. Although the reliability and the security price is expensive, GM achieves 99.9 % of the Myrinet link bandwidth and a user-level latency of 7 μs with the latest Myrinet interface.

GM supports many different architectures and operating systems (Linux, Solaris, Windows, Tru64, Irix, VxWorks, etc). I used it on IA32 stations with Linux kernels 2.2.19 and 2.4.x.

GM provides many functions for the user and is most often used for its reliable communication. A key property of GM is its asynchronism: send and receive are processed by the interface without requiring the host to block.

Communications can also easily be overlapped with computations without using a specific communication thread. Actually you can consider that such a thread runs in the LANai.

Overlapping is a very important thing for customers who need powerful clusters: they will not waste time waiting for communications to end.

1.3 GM details

1.3.1 Internals

For the user, GM is just a library of functions which can be called in any user process (or in the kernel, even if the current GM 1.5 version is not really aimed at this). This library is only a covering interface which provides a lot of functions but hides complicated internal processing.

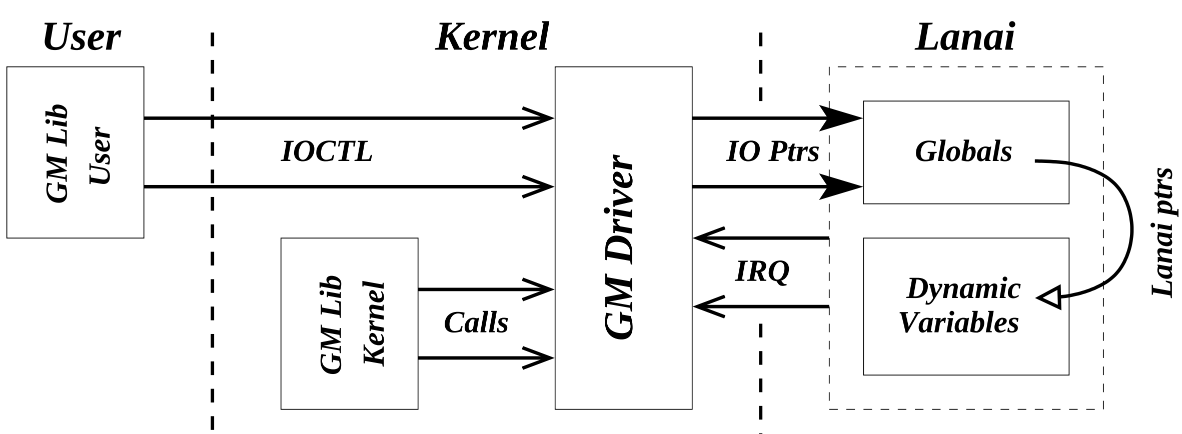

Figure 1.2: GM Internals.

The library gives user requests to the GM driver module (see figure 1.2). If your process is a user process, it needs IOCTL (with the GM device) to communicate with the kernel. In kernel mode, functions of GM can access all of the GM driver data. (The kernel library is just a nicer interface for the user).

The driver keeps some pointers to the LANai memory which is accessible as PCI device memory (see appendix A.1). Thus it can write or read in the LANai, which is the way most requests are processed: the host writes requests in the LANai request queue. The LANai processes these requests asynchronously and will send an interrupt to the host each time it finishes one.

The LANai is controlled by the MCP (*Myrinet Control Program*), which is loaded into the LANai memory during the insertion of the GM module. In this way you can modify the MCP and test it just by reinserting the GM module. Using this very simple MCP modification method I have been able to add new functions to the standard GM MCP.

1.3.2 Using GM

All communication functions in GM are asynchronous. A send request is written in the MCP send queue with a pointer to a callback function (which will be called when the send has completed), and you can use blocking or non-blocking functions to get GM events. This method is useful for overlapping.

Send and receive buffers can be easily allocated as DMAable memory. You can send a message specifying that it needs a *provided* buffer in the receiver's main memory. When this received message has been copied into this provided buffer, the receiver will get a *receive event*.

You can also send a *directed* message which will be copied at a specified target address without events.

So there are two main models of communication with GM: you can build solid protocol by using usual sends and waiting for events. Or, you can let the sender write in the receiver memory without wasting time by providing buffers, waiting for events, etc[1].

[1]More data about the GM API can be found in [4].

Chapter 2

Optimizing data transfer

2.1 State of the art

2.1.1 Usual path from disks to the network

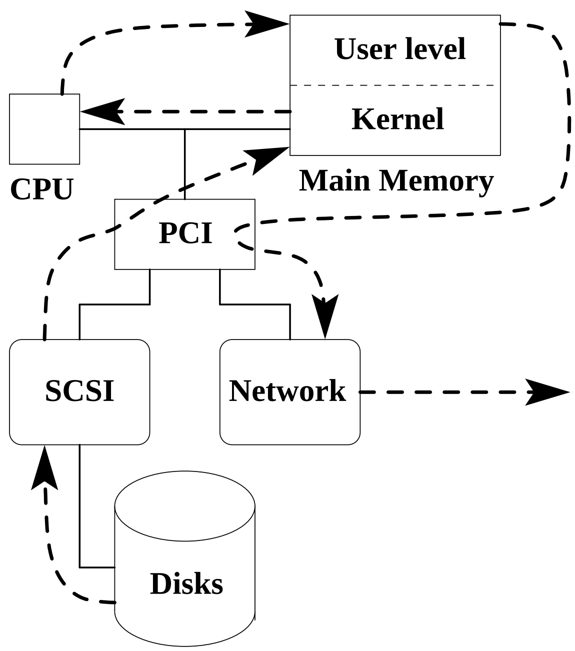

Figure 2.1: Usual data path from SCSI disks to the Network.

Figure 2.1 shows the path from the disks to the network when you give a read request in an NFS (or any other kind of data) server.

With Linux, data being sent by the network interface must first go from the SCSI disks and interface to the main memory. In most of cases, blocks are stored in the kernel memory. The VFS (*Virtual File System*) will then extract the desired segment of data from these blocks and copy it into the user space. It will then be sent by the network interface (sometimes after being copied again in the kernel space to be encapsulated).

Even if this path is not always required, it shows the huge detour data must follow to go from disks to the network.

2.1.2 Limits of this path

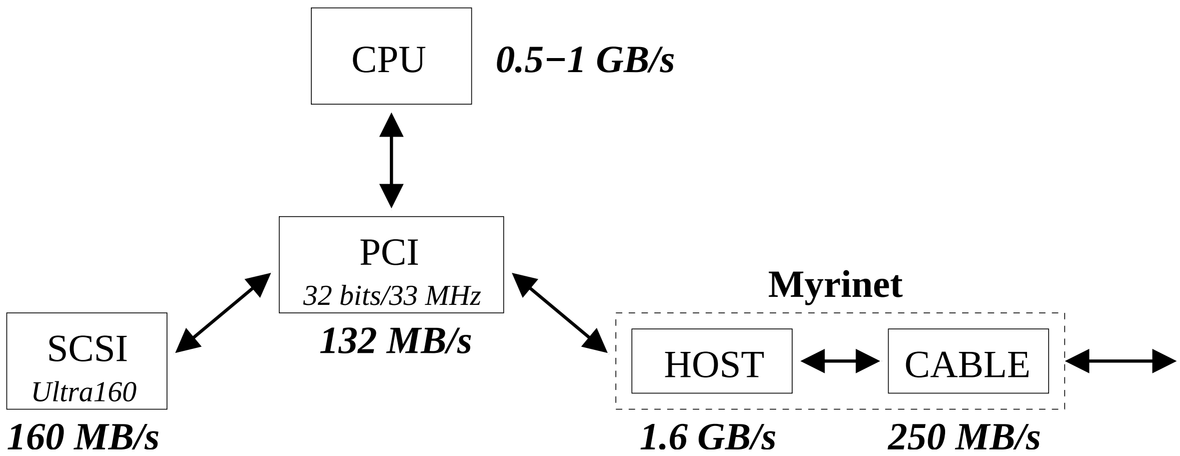

Figure 2.2: Bandwidth in DMA engines.

Copying data between kernel and user space requires a high performance processor. That is why NFS servers always use several powerful processors. However, this is not the only limit of this sending method: figure 2.2 shows the bandwidth supported by different DMA engines in a usual computer.

The PCI bus seems not much slower than the SCSI interface. Actually, the most important fact here is that the PCI bus is used two times, and in two different ways: from the SCSI interface to the host, and next from the host to the network interface. The real bandwidth of the PCI bus DMA engine is also divided by two. Moreover, delays needed for the arbitration of the two different ways decreases the real bandwidth again.

Nowadays, an NFS server rarely supports more than 30 MB/s even if you use a 64-bit/66 MHz PCI bus (524 MB/s) and a multiprocessor host.

2.2 Bypassing the kernel

In the past few years, PCI interfaces have become able to provide powerful applications to the host. Usually video cards can process 3D calculations using their built-in powerful and specialized CPU and a lot of memory.

The Myrinet interface is one of the first non-video interfaces that provides large and fast memory and a powerful processor. If you have to use such an interface to send data, why not using all its possibilities and performance instead of always using the overloaded main CPU ?

2.2.1 General considerations

Data transfers between the host and SCSI/Network interfaces are possible because the memory of the PCI interface can be mapped in the physical memory of the host (see appendix A.1). Thus, DMA engines can easily copy data among themselves.

Typically programmers use the fact that the PCI memory is mapped in the host memory, or that the host memory is mapped into the PCI's memory. Actually the memory of *all* interfaces is mapped into *all* other PCI interfaces on the same PCI bus. So DMA transfer is enabled for example between a SCSI and a network interface.

The main interest of such a transfer is to avoid wasting time and CPU usage when going through the kernel/user space, and to keep the maximum bandwidth for the PCI bus by using it only once per transfer.

However, bypassing the host and also the processing in the kernel means that you cannot extract or encapsulate data (like VFS or IP drivers usually do in the kernel). Such a data server will also only be able to send *blocks* from the disks. Usual network protocol will not be available.

This leads to the main interest of using the Myrinet interface: GM sending requires no processing in the host. Packets are created in the LANai (by the MCP which adds a header and fragments).

2.2.2 Opiom and the LANai SRAM

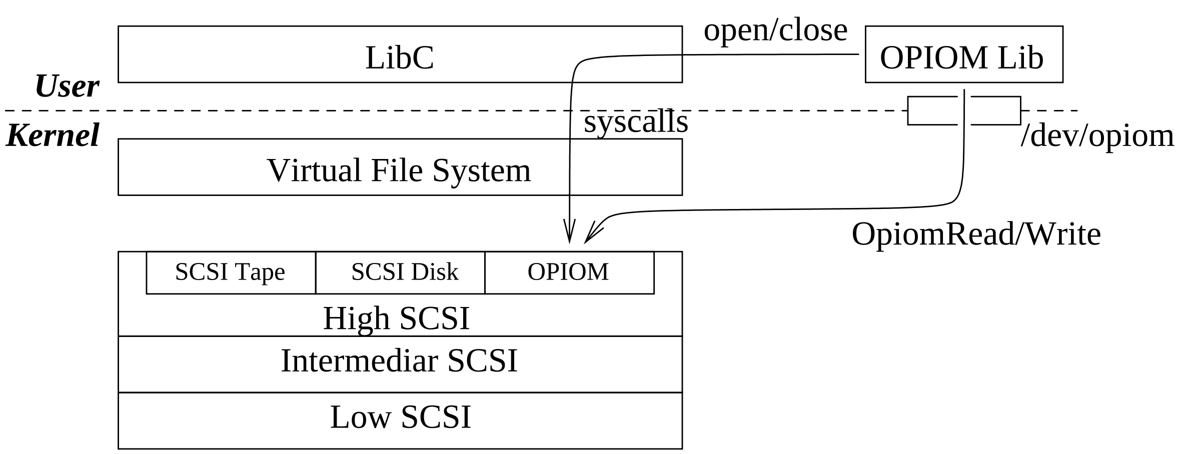

Figure 2.3: The OPIOM Library and Driver.

No usual library allows the user to copy blocks from disks to a PCI interface. That is why Patrick Geoffray wrote OPIOM (*Off-Processor IO over Myrinet*) two years ago (see [3]).

Such a library replaces the usual `read` and `write` system calls by block manipulations (see figure 2.3). Thus OPIOM provides user functions which transfer blocks of data between a disk and a physical address which can be in the memory of a PCI interface. You can use the usual `libc` calls (`open`, `close`, `lseek`) for all functions which do not access the disks.

Actually OPIOM needs a specific driver (based on the generic SCSI driver) because the usual Linux SCSI stack only manipulates virtual addresses in its upper level. Thus the

Opiom SCSI is a modified SCSI generic driver which directly manipulates physical addresses.

2.2.3 Expected advantages for data servers

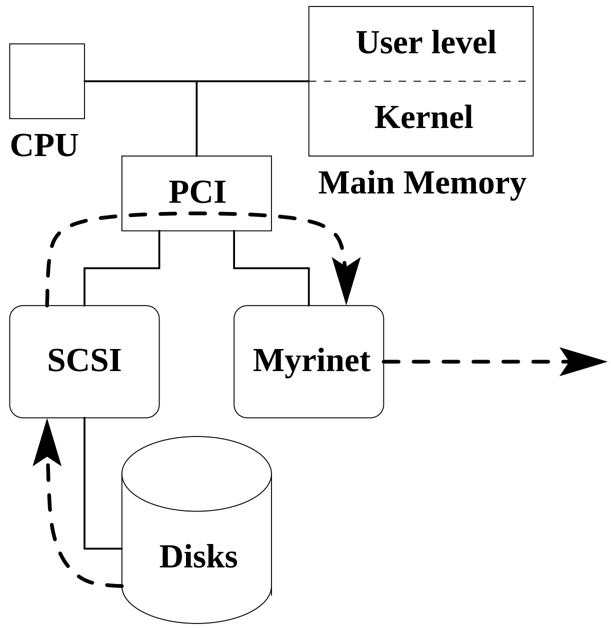

Figure 2.4: Fast path from SCSI to Myrinet with OPIOM.

Using OPIOM I can copy blocks from SCSI disks to the Myrinet interface with full PCI bus bandwidth (see figure 2.4) and with an almost idle host: the main CPU is only used to send the OPIOM request. The real transfer is entirely done by the SCSI and the Myrinet interface.

I only needed to safely allocate buffers in the LANai SRAM and add new GM send functions to send these blocks from the LANai to the network.

With recent SCSI disks and interfaces I can send up to 320 MB/s on the network, without using the main CPU ! However, for an NFS server I will also need writing and caching functions (see figure 2.5), and producing these functions is non-trivial. For example, the Myrinet interface is not supposed to be used as the slave for OPIOM writing.

Such an implementation was too complex for my internship (see [6]). Therefore, I only implemented send functions for read-only data servers and then developed some new possibilities in the client.

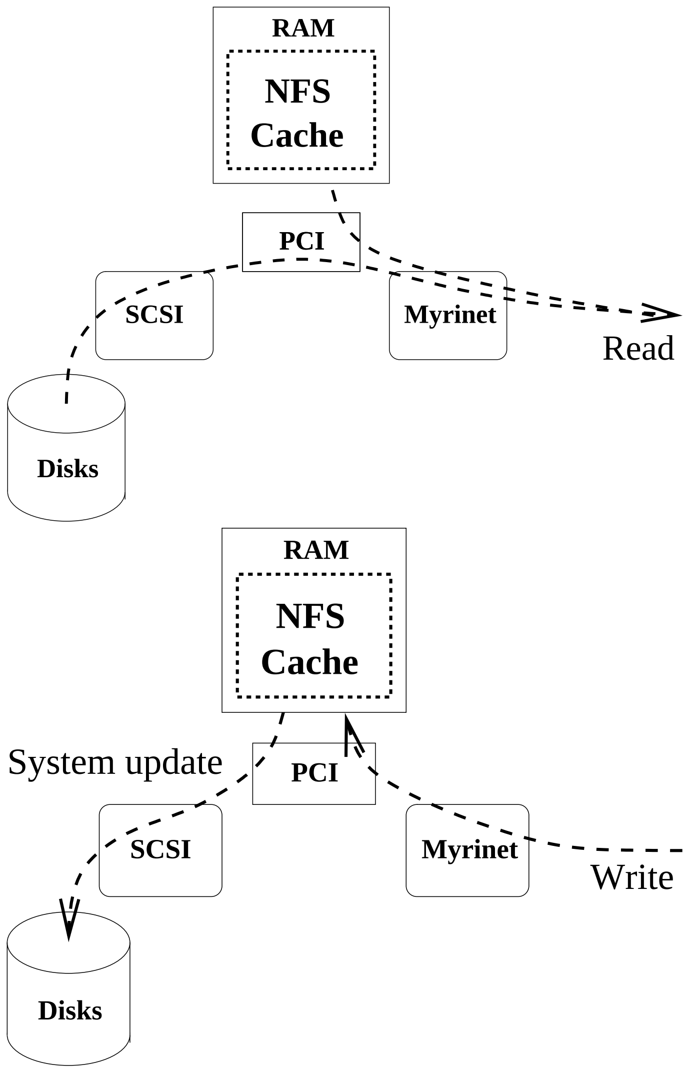

Figure 2.5: NFS Read/Write Model.

Chapter 3

New implementation of data servers

Using OPIOM I can provide fast data transfer from a SCSI disk to the Myrinet card. As it is impossible with GM to send on the network without temporarily storing the data in the LANai SRAM, I need to provide a workspace for Opiom in this memory.

Recent Myrinet interfaces possess between 2 and 8 megabytes of built-in memory. As the MCP only uses about 1 MB, I can use a large Opiom workspace. However, I also need to safely allocate buffers for applications in this workspace. That is why we decided that I must implement a memory managing library for the LANai SRAM.

3.1 Malloc for the LANai SRAM

3.1.1 Theoretical considerations

Nowadays the Opiom workspace will be used to store blocks coming from disks. Most file systems provide blocks whose size is a multiple of 512 bytes. Thus, my malloc function will only be able to manipulate blocks of 512 bytes.

Even if several future applications will probably require smaller buffers (for example to keep a message during a broadcast in MPICH-GM), this size should remain a good compromise between the implementation's difficulty, efficiency and space wasting.

The Linux kernel uses the Buddy algorithm to manage its 4 kB pages (see [1]). Thus it seemed interesting to reuse this existing implementation to manage my 512-bytes pages. But looking at [5] and [2] I noticed that this algorithm (and its variants) is not really adapted to my main goal, especially because I do not manage a huge number of pages and do not want to waste some of them. So I decided to find another well adapted solution by myself.

3.1.2 Managing allocated segments

An AVL tree is known as an efficient solution to provide fast insertion, deletion and access in an ordered set of objects (cost in $\log$ of the number of elements). Thus I use this structure to store allocated buffers ordered by their beginning address (see figure 3.1).

I also store in each node of this AVL tree a pointer to the free segment (if it exists) just after the allocated segment. It will be useful to provide a fast merging method (see below).

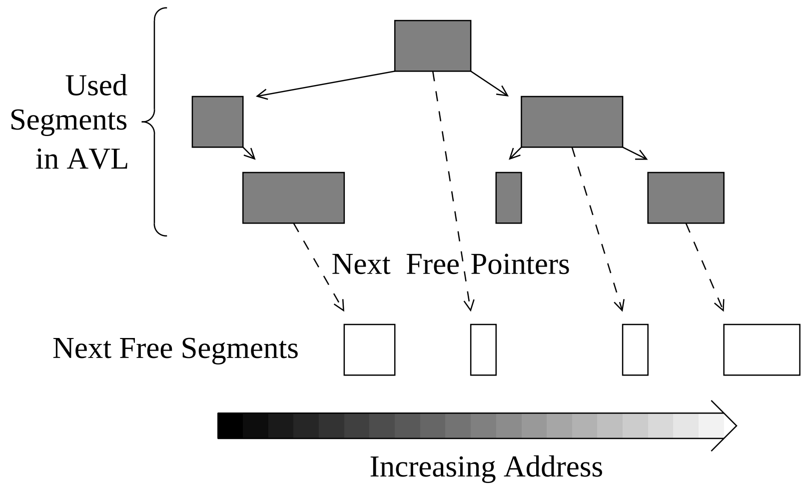

Figure 3.1: Used segments and next free pointers storage.

3.1.3 Managing free segments

Storing free segments is not an easy problem because of the variety of different possible needs. Sometimes programmers will need very fast `malloc/free` functions but accept wasting space (as in the `libc` implementation).

For my purpose, I do not want to waste space. (I only have a few thousands blocks). I will particularly need to merge contiguous free segments. That is why I store the next free segments in the AVL tree of used segments: when freeing an allocated segment, I will merge it with the previous free segment and the next one. So I need easy access to them.

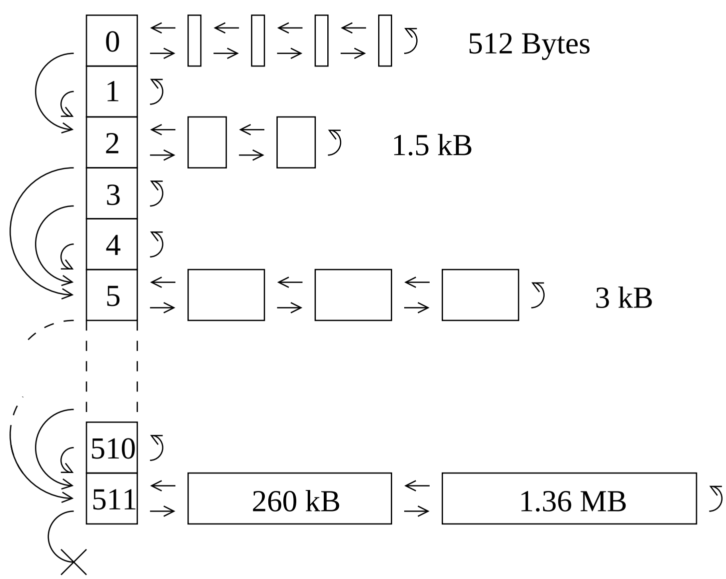

Figure 3.2: Free segments storage.

Managing free segments without wasting space implies fast access to the smallest adequate free segment during `malloc`. Thus I used a list of same-sized free segments for each size, and pointers to the next non-empty list. However, it implies a few thousands different lists and thus expensive operations to manage it.

In order to reduce managing time of all these lists I group big segments in one special last list (see figure 3.2). The number of lists is now low (511 in my implementation, instead of a few thousand) and the number of different large free segments in the last list is less than 16 (for a 4 MB Opiom workspace).

3.1.4 Implementation in GM

The primary interest of such a memory-managing library is to provide easy access to the Opiom workspace in the LANai SRAM.

The address of this workspace is obtained by GM and the library automatically gives a well-addressed segment. The user just has to ask for a segment without taking care of the physical location of the buffer (in the PCI memory).

I am managing the LANai memory but I decided to run the library in the host. Actually the only way the LANai gives events to the host is by using interrupts. We cannot wait for an interrupt each time the user asks for a `malloc` or a `free` because it will be too slow and complicated.

In fact, using the host does not imply any problem because we manage the LANai memory without really writing information in this memory.

The last choice to be decided was between implementing the library in the kernel or as a user program. A user library cannot process concurrent application requests. And for future usage in an NFS server, safe data storage will be needed. It will be especially useful to prevent hackers from modifying data in the LANai SRAM. Thus I decided to keep all physical pointers in the kernel, associating a `pid` to each allocated segment and only giving *segment identifiers* to users.

In this way concurrent applications possess their own safe buffer and cannot access to the others. My memory manager is now integrated in the GM driver and accessible to the user with new functions of the GM library.

3.2 Sending from the LANai SRAM

After copying data from SCSI disks to an Opiom buffer in the LANai, I must send it.

GM provides the *Myrinet Control Program* which is the LANai's firmware. It is copied into the LANai memory when installing the GM driver. This program is not supposed to send data which is already in the LANai SRAM. Thus I modified the MCP in order to implement Opiom sending methods.

3.2.1 Send request processing in the MCP

The MCP is programmed in the C language (with gcc extensions) and needs its special `lanai-gcc` compiler. The MCP organization is a state machine describing the state of buffer DMA sending and receiving.

There are also a list of handlers. Transitions are chosen by the MCP after reading some software and hardware flags.

When receiving a send request in its queue the MCP looks for its two static sending buffers. When one buffer is free, it begins DMA copying with the first system-page-sized part of the message in this buffer. Then it sends this buffer on the network while the second buffer is being filled.

Sending is also organized as a 2-stage pipeline: copying from the host and then sending.

The current GM 1.5 version cannot send packets on the network if their header is not preceding them. There is no space for such an header in the data coming from the disk, and I do not want to break Opiom performance by keeping space between different parts of the Opiom data (noncontiguous destination addresses will break DMA copy performance).

Thus, I will copy data from the Opiom buffer to the usual static sending buffers, and then we will send them.

3.2.2 Opiom sending functions

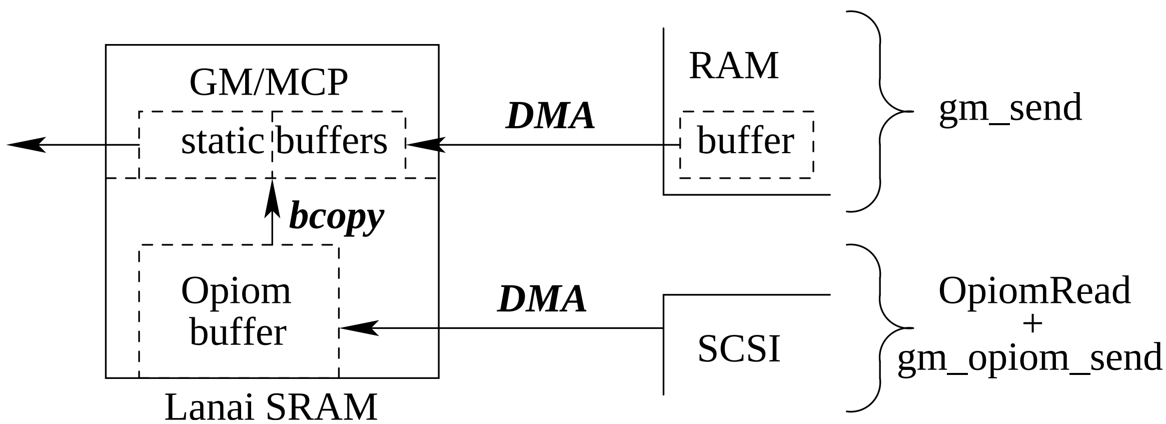

Figure 3.3: Comparison between usual and Opiom send.

Sending data from the Opiom buffer requires me to copy it to a GM static buffer. Thus I needed a fast `bcopy` function in the MCP (see figure 3.3.

Using the fact that the LANai can access the SRAM in only two cycles and copy 32-bits words, I can copy up to 2 bytes per cycle. With the 133 MHz LANai 9 I was using, that meant 266 MB/s (two times faster than DMA copying from the host memory with a 32-bit/33 MHz PCI bus).

Actually such a copy requires a huge part of the memory bus (1.067 MB/s with this kind of LANai). As the memory bus is also used to load LANai instructions, it is impossible. In fact, out of 4 memory bus cycles, one is used for the current sending on the network and another one to load instructions. In reality I can only copy 133 MB/s. This amount is approximately the bandwidth that a usual station can get from the SCSI disks.

Most recent 200 MHz LANai 9 provide new memory bus cycle organization so that this limitation is highly reduced.

After copying data in the GM static buffers, it should be easy to send it on the network. I would only have to take the code of usual sending.

Another large difficulty appears here: transitions in the machine state of the MCP are processed depending on software and hardware flags. Most of these flags are associated to usual DMA copies from the host memory. However, for Opiom sending, I do not use such a DMA.

Thus I had to solve this difficulty by changing flags so that the MCP will behave as if my synchronous `bcopy` instruction is an asynchronous usual DMA which is already finished.

Actually if you do not use this hack I can send too but the performance is very bad: the MCP gets a workless state and begins working again after a timeout.

Using this method, I implemented Opiom sending functions for *usual send* (requiring a provided buffer in the receiver's main memory and implying a receive event) and *directed send* (writing in the receiver's virtual memory without any notification). Thus all GM send methods can now be used with Opiom.

3.2.3 Automating Opiom sending

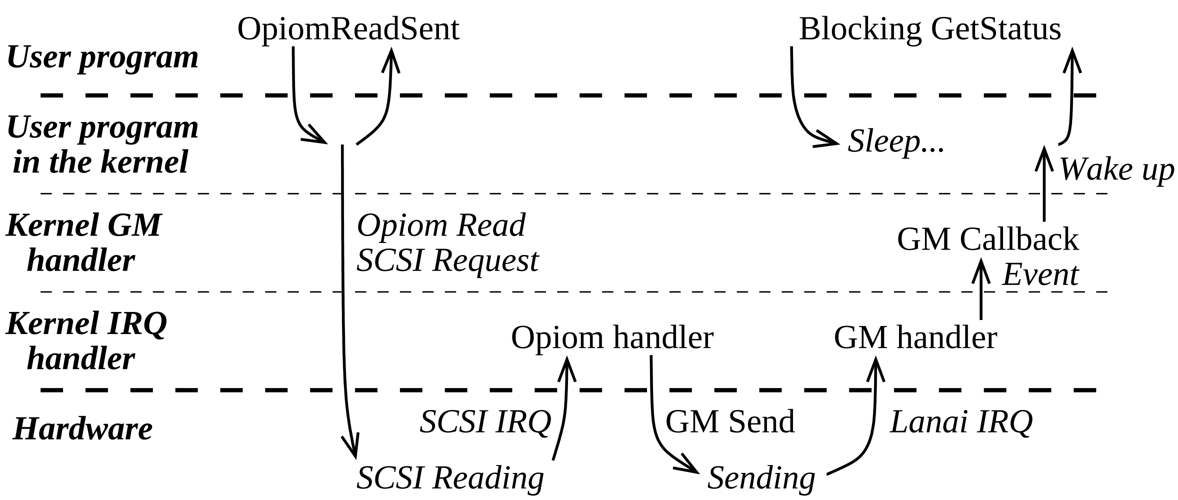

Figure 3.4: OpiomReadSent processing with different kernel threads.

The `OpiomRead` functions is asynchronous. Thus real Opiom sending requires me to use `Opiom_GetStatus` for a blocking wait until the SCSI request completion, and then I can use `gm_opiom_send` to send on the network.

We found it interesting to group all of these in a global `Opiom_ReadSent` functions. You can see in figure 3.4 the model of such a process. Actually, when a SCSI request is finished, the kernel IRQ handler (a dedicated thread) gets an IRQ. It uses the Opiom IRQ handler which notices the user of this event.

The GM send can easily be processed in the Opiom IRQ handler. The main difficulty appears when you have to wait for the `Opiom_ReadSent` completion. Actually you wait for the GM send completion, but I cannot block in the Opiom IRQ handler because it will disable *all* other IRQ.

Thus I use a new thread in the kernel. This thread blocks until a GM event comes and then runs the associated callback to notify the user of the request completion.

And this method keeps the asynchronity of the Opiom request: the user still has to use `Opiom_GetStatus` to block until the completion of an Opiom request.

You can notice that the send is done in the kernel and that implies that you must open a GM port in the kernel. It will only be used to send Opiom blocks. All other messages will be sent or received on another port, opened in the user program.

3.2.4 Bandwidth limits

With all these modifications I can Opiom-send about 110 MB/s on a 133 MHz LANai 9 and sature the Myrinet 2000 cable (250 MB/s) with a recent 200 MHz LANai. I need to be able to get such a bandwidth from the SCSI disks.

Recent Ultra-3 SCSI interfaces can theoretically support up to 320 MB/s. Thus I should be able to provide very high bandwidth data transfer over Myrinet.

In the future GM 2.0 version, the MCP can send a message even if the header is not preceding it. Not using the `bcopy` functions in the LANai will imply a very low latency (about 2 μs, as in BIP) for Opiom send, and no bandwidth limitations with the old LANai.

Chapter 4

Application to remote video

Opiom and my new GM sending functions are now able to provide fast data servers. I applied these features to remote video.

4.1 Interests of the model

4.1.1 Real needs

The most powerful computer is the IBM SP3 in San Francisco. It is often used by physicists who have very precise desires about the way they get their results: real time video on an about 3000×3000 pixel resolution! That means about 1 GB/s in the frame buffer!

Actually this cluster contains several thousands processors. Four nodes are used to get results and show it. Thus each node has about 250 MB/s to process. But you will never find a computer able to push 250 MB/s from the network to the video card without very specific hardware.

That is why IBM developed new interfaces which are both a network and a video card and can DMA copy with a very high bandwidth without going in the kernel.

I am going to explain how to replace these very expensive interfaces (about $ 50 000 each) by a simple Myrinet interface (which costs about 25 times less).

4.1.2 Technical requirements

I am using DMA copy from SCSI disks to the Myrinet interface. I can also use DMA copy from the Myrinet interface to another PCI device, especially to the frame buffer (see figure 4.1).

Recent video interfaces own DMA engines supporting more than 200 MB/s. With a high performance PCI bus (that means, not a 32 bits/33 MHz one) we can copy 30 1600x1200-frames in 32bits each second.

For full screen video the address range in the frame buffer is totally contiguous. However, if you want to write in a window, you will need one DMA per line. The MCP can be modified to do such a cutting, and the LANai is powerful enough so that it can easily do that and let the host sleep.

The only limit to this model is the fact that you need your hardware to be able to DMA copy from the Myrinet interface to the video card. If your video card is a PCI one and is on

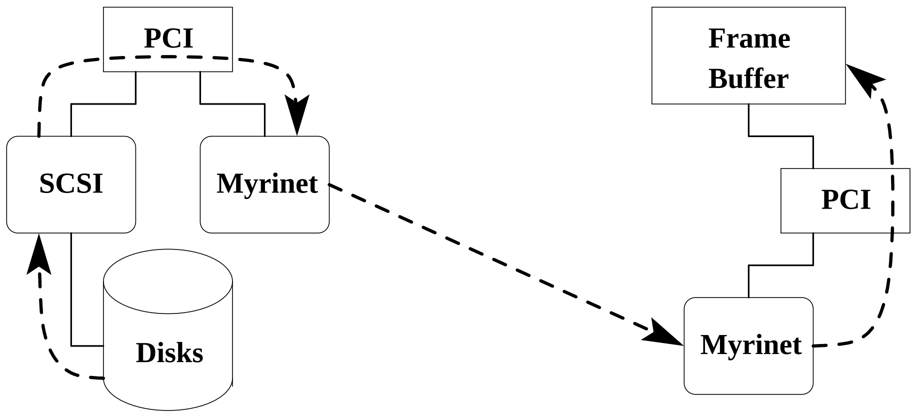

Figure 4.1: Fast path from SCSI to a Remote Frame Buffer.

the same PCI bus as the Myrinet interface it will work.

However, if the bus is different, or if your video card is an AGP one (in fact AGP is another PCI bus) it will probably not work. PCI specifications require bridges to be able to DMA copy from one bus to another one but no one uses this feature. So most of the bridges are unable to do that.

4.2 Using the LANai performance

I am now going to describe how to use GM and Opiom sending to provide remote video. Thus we now look in the receiver.

4.2.1 Usual send functions

The easiest way to provide remote video over Myrinet is to send one message per line and to ask the receiver to provide a receive buffer for this line.

The first disadvantage of this method is the fact that providing a buffer implies a system call. With a few hundred lines per images and about 30 images per second, it will be a very expensive operation.

In this case, using *directed send* is a good solution: you do not need to provide the receive buffer. And I do not especially need events for each received lines.

I tried this method with usual computers (athlons with 32-bit/33 MHz and 3dfx video interface). Assuming that the SCSI is fast enough, I obtain approximately 100 MB/s which is equivalent to 30 1024x768-32bits-frames per second.

This bandwidth is satisfying because the theorical limit is only 133 MB/s (due to the PCI bus). But this method requires the receiver to update the video details by sending a message to the server each time the window is moved or resized. I will describe in 4.2.2 how to avoid all of this protocol.

This method requires the user to map the frame buffer in the user program. You will see in appendix A.2 that this is a non-trivial operation.

Moreover getting the physical frame buffer address is not much easier: you cannot use the kernel PCI functions because of the variety of different video interfaces. The easiest solution is to ask the X server (with the DGA library for example) even if documentation about it is really difficult to find.

4.2.2 A modified MCP to process received images

The usual packet types (directed or not) also seemed not really adapted to remote video. Thus I added a new kind of GM packet designed for remote video.

Using physical address

First the receiver does not use the virtual address: with the physical address of the frame buffer you can avoid the mapping. You have to keep in mind that virtual addresses are useful for usual main memory managing. But PCI memory physical addresses never change. So such a mapping is only useful to provide easy access from a usual program. In the MCP, that means in the hardware, it is useless.

One message per image

Sending one line per message is required by the fact that the address range in the frame buffer is not contiguous. To overcome this, I programmed the managing of my new kind of message so that the receiver's MCP gets an entire image and the DMA engine copies visible parts of the lines at their respective position in the frame buffer.

Actually the usual receiver's MCP never DMA copies more than a system-page-sized part (4 kB with Linux/x86) because contiguous virtual pages are not necessarly physically contiguous. As a line of 800 32-bits pixels uses 3.2 kB, this modification results in very little decrease in global performance.

Avoiding the protocol

Updating the video details of the receiver in the sender seems useless because these details are only used in the receiver and will come back to it in GM packets.

The best solution is also to store video details in the receiver's LANai. The receiver's MCP only has to read these variables to know where it has to write in the frame buffer.

This modification has another advantage: the time between the video details real modification and the receiving of the first image at the updated position is small (a syscall is faster than an exchange of GM message).

Final implementation and performance

After reading an image with Opiom I send it entirely in a GM message. The receiver writes video properties in the LANai with a new GM function (screen size and depth, image and window size, frame buffer beginning address). The MCP reads them and copies visible parts of the lines from the received image in the frame buffer (see figure 4.2).

With such an implementation, remote video is really easy to use. With usual hardware and 133 MHz LANai you can transfer about 30 frames per second (1024x768, 32-bits per pixel).

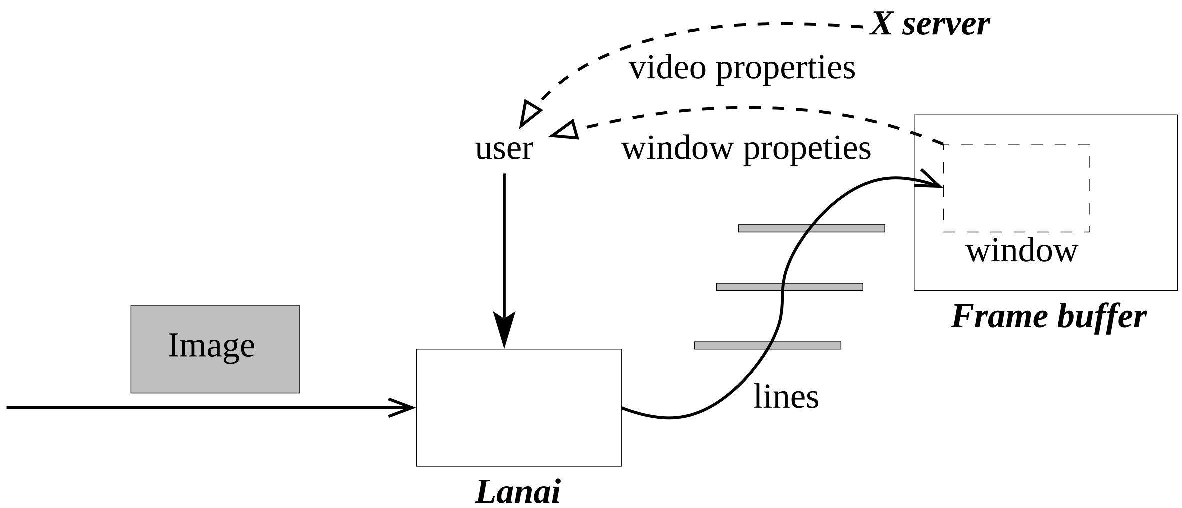

Figure 4.2: Remote Video Model.

4.3 Linux World demonstration

The Linux World tradeshow took place in San Francisco between the 26th and the 30th of August, 2001. Like many computer companies, Myricom was there presenting a bunch of demonstrations, especially a video demonstration using my implementation.

First I took some DVDs, decrypted them and extracted a few thousand MPEG2 frames. I wrote all these frames on a hard drive.

To yield high performance throughput, I parallelize requests and pipeline reading and sending using several different buffers.

I wrote an X client with two windows and added the possibility to keep several different video properties in the same Myrinet interface. The client was able to give window moving or resizing events to the LANai.

By this way, I built a server reading about 100 MB/s and 2 remote video clients showing four different movies (35 frames/s, 720x480, 16-bits). All of the clients were totally idle while the server was performing its operations using less than five per cent of the processor.

Conclusion

Using the existing Opiom implementation and some modifications in GM, especially in the MCP, it is possible to provide very effective data transfer.

The LANai SRAM managing allows you to use concurrent applications with the Opiom workspace in the LANai, with the safety that a file server can require.

My remote video implementation proves that going on with the idea of DMA transfer between PCI device bypassing the kernel is an interesting solution to provide fast transfer between almost idle stations.

Even if networks other than Myrinet are able to transfer so much data, I demonstrated significant advantages of the Myrinet interface: the LANai is powerful and its MCP can easily be modified for specific purposes.

Future interesting research and development may use these ideas and results to provide a very effective NFS server over Myrinet as much as specific parallel file system implementations.

I expected a lot from this internship and I am very happy with the time I spent at Myricom this summer. I learned a lot of things concerning operating systems, especially the Linux kernel, the way it manage usual features (processes, memory, file systems, etc.) and how it interacts with hardware resources. I would never have thought that I would have been able to develop such an impressive implementation. It was really very interesting.

Contents

Appendix A

PCI memory mapping

A.1 Physical memory and mapping in the kernel

A.1.1 Physical Mapping from the BIOS

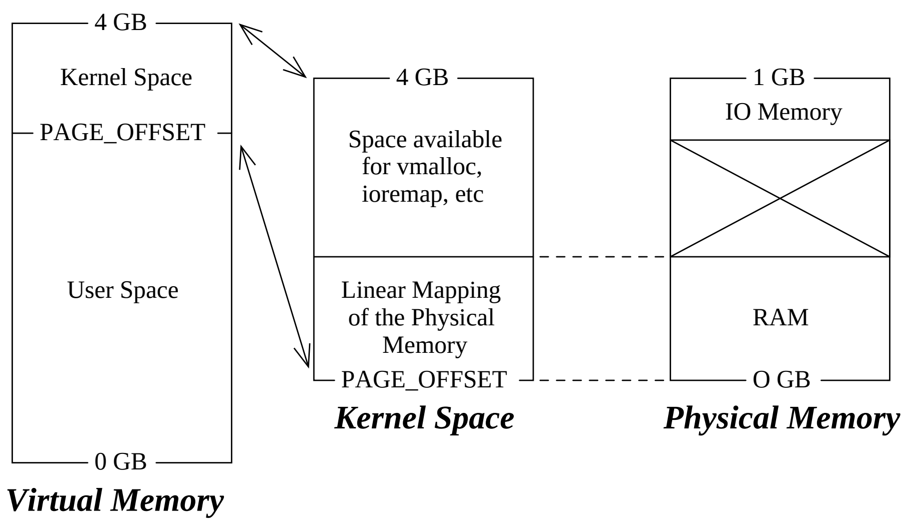

Figure A.1: Physical and Virtual memory.

When you boot your computer the BIOS scans all the PCI devices notably to find their own memory. Each time it finds such a memory, it maps it at the end of the first gigabyte of physical memory while the main memory is at the beginning (see the right part of the figure A.1). This mapping is the only way for the system to access to a precise address is the main memory or in the PCI memory[1].

A.1.2 Mapping in the kernel address space

The kernel address space is usually between the third and the fourth gigabyte. After booting, this address space only contains a linear mapping of the main memory at the beginning of

[1]More data concerning all the following can be found in [7].

the third gigabyte (see figure A.1).

All of the remaining address space up to the fourth gigabyte can be used for dynamic variables (allocated with `vmalloc`) or to remap IO memory (especially PCI).

PCI memory mapping in the kernel address space is really trivial. The `ioremap` function uses the beginning physical address (fixed by the BIOS, see A.1.1) and the size. It allocates a new segment in the remaining kernel space, associates it to the physical address range and returns the beginning virtual address. After this you can directly access to the PCI memory using the new mapping as if it were whatever virtual address in the main memory.

A.2 Mapping in a user process

Mapping PCI memory in the user space is not so trivial. Actually all needed functions already exist but the system requires a careful manipulation of the low level kernel functions.

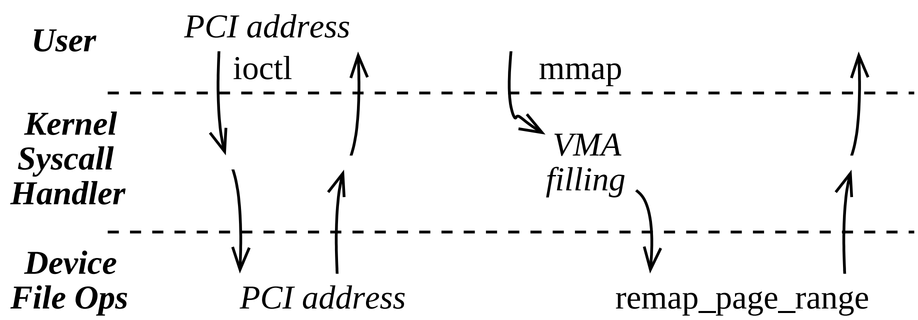

Figure A.2: PCI memory mapping in a user process.

As the kernel provides the function `ioremap` to map a virtual address in the kernel space, the function `remap_page_range` maps a physical memory address range in the user space. Obviously you cannot directly call this function from the user level. You have to switch in kernel using an IOCTL or another system call.

Actually the `mmap` system call exists especially to map a file in the virtual memory. Thus I just have to create a simple new device with an associated `mmap` file operation.

When I call the `mmap` on my device, the system creates a new *Virtual Memory Area* in the usual system call handler. That means it adds new pages to the process address space. Then it calls the specific device `mmap` operation. That is where I can use `remap_page_range` to apply the desired mapping.

Actually the `mmap` is not really aimed at such mapping so its offset parameter cannot be considered as a good way to give the physical PCI address. That is why I use an IOCTL to fix the physical address in my device driver before calling `mmap`.

Such a mapping is not really needed for the Myrinet interface: GM maps the LANai memory in the kernel space and the user does not need to access it in user mode because of all the functions that the GM library provides. However, if you need to use the frame buffer as I do (see 4.2.1), i.e., mapping the video memory in the user space, you may use all of this.

Bibliography

[1] Daniel Bovet and Marco Cesati. *Understanding the Linux kernel*. O'Reilly, 2001.

[2] Erik D. Demaine and J. Ian Munro. Fast allocation and deallocation with an improved buddy system. In *Foundations of Software Technology and Theoretical Computer Science*, pages 84–96, 1999.

[3] Patrick Geoffray. *Contribution au support de communication des serveurs vidéo distribués suivant une architecture réseau de type grappe de PC*. PhD thesis, Université Claude Bernard Lyon I, 2001.

[4] The GM API. Myricom, Inc. `http://www.myri.com/scs/GM/doc/gm_toc.html`.

[5] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: solved? *ACM SIGPLAN Notices*, 34(3):26–36, 1999.

[6] Network file system protocol specification. Sun Microsystem, Inc., 1989. `http://csgrad.cs.vt.edu/~fdrake/cs5204/rpc/rfc1094.html`.

[7] Alessandro Rubini. *Linux device drivers*. O'Reilly, 2001.