Journal 5, Improving performance

1 Objectives

As seen in the performance test, the current transfer speeds of the system are far away from the desired. The lack of speed is most probably due to excessive overhead in the type of transfer currently used, where driver requests are used to read and write single registers.
Alternatives could be:

- Transfer arrays of data to the driver and let the driver transfer several DWORDs per request.
- Map the device memory/registers to a user-space memory range, to allow for direct access without going through the driver.
- Use bus mastering/DMA to let the board perform the read/write operations.

2 Problem analysis

The described alternatives are discussed in the following sections.

2.1 Transfer arrays of data to the driver and let the driver transfer several DWORDs per request

Sadly, due to limitations in the x86 architecture, the host system is not able to transfer more than one DWORD per transmission to the PCI Express board. Therefore, there will always be a large amount of overhead (3 out of 4 DWORDS when writing to the board, and 6 out of 7 DWORDS when reading from the board).
On Windows, the kernel calls *READ* / *WRITE_REGISTER_BUFFER_ULONG* exist, which are able to read and write arrays of ULONGs. However, by looking through the definition of these calls, it was discovered that they simply perform a *READ* / *WRITE_REGISTER_ULONG* for each DWORD in the array, and therefore do not reduce the overhead on the PCI Express interface.
It was seen in the performance test though, that performing a driver request in itself is rather inefficient. Thus lowering the amount of necessary driver requests by making each request transfer more than one DWORD would most probably improve performance. For such functionality, the *READ* / *WRITE_REGISTER_BUFFER_ULONG* functions might as well be used.

2.2 Map the device memory/registers to the user space

Instead of transferring data to the driver, and then getting the driver to transfer the data to the board, another possibility is to map the memory / registers of the board to the memory range of the user-mode application. This will provide the application with direct access to the memory / registers of the board, thus removing the need for performing numerous time-consuming driver requests.
The interface to the user-mode application is also simplified, as it is just an integer array that can be used with existing functions, instead of having to use a *DeviceIOControl()* / *ioctl()* request.

2.3 Use bus mastering/DMA to let the board perform the read/write operations

As explained earlier, the host system can only issue single DWORD transfers on the PCI Express interface. However, this limitation only exists for transfers initiated by the host system. If implemented, the board itself should be able to issue transfers of up to 1024 DWORDs (= 4 kB) in both directions. This would drastically lower the amount of overhead on the PCI Express interface when transferring more than a few DWORDs.
To perform such a transfer, the host system would first need to setup the transfer, by transferring an address and a size to the board, and then issuing a “go” command. The board should then create and transmit the read or write commands, perform the transfer, and issue an interrupt to the host system when done.
Apart from the reduced overhead on the PCI Express interface, this also has the advantage that it does not use the CPU of the host system, thereby allowing applications to keep running while transfers are in progress. The problem with this approach is that it is rather advanced, and will

require a large modification of the *PIO* module in the FPGA design, and the device driver on the host system.

2.4 Choice of approach

From the above three approaches, it is chosen to implement the second, and map the device memory to user-space. Even though the bus mastering / DMA approach would be more interesting, and would probably provide a better result, it is deemed too advanced for the scope and timeframe of this project. The first approach could probably provide performance close to that of the second approach when transferring large amounts of data, but the second approach can potentially increase performance when transferring a few DWORDs also, while providing a simpler and more intuitive interface to the user-mode application as well.

3 Mapping device memory to user-space

The mapping of device memory to user-space now needs to be incorporated into the drivers. The approaches differ for the Windows and Linux platforms. On Windows, the mapping is done in a *DeviceIOControl()* request, which just returns a pointer to the mapped memory area. On Linux, a separate request type, *mmap*, is used instead.

3.1 Implementiation on Windows

On Windows, the mapping is done in a *DeviceIOControl()* request. This means that two new control codes will need to be added, one for mapping, and one for unmapping. These are defined in *ioctrl.h* as follows:

```
#define MDS_MMAP_BAR CTL_CODE( FILE_DEVICE_UNKNOWN, \
                                    0x030, \
                                    METHOD_BUFFERED, \
                                    FILE_ANY_ACCESS)
#define MDS_MUNMAP_BAR CTL_CODE( FILE_DEVICE_UNKNOWN, \
                                    0x031, \
                                    METHOD_BUFFERED, \
                                    FILE_ANY_ACCESS)
```

The method used is described in the Microsoft support document “How To Map Adapter RAM into Process Address Space“[1]. The mapping functionality uses a so-called Memory Descriptor List (MDL) for each mapping. To be able to unmap the memory when it is not used anymore, this MDL needs to be stored in the driver. This is done in the *DEVICE_EXTENSION* structure specified in *s3_1000.h*, by adding an array of six *PMDL* objects to the structure (one for each BAR).
The functionality itself is implemented in the *Control()* function in *s3_1000.c*, as new cases in the switch-case statement. First, a new MDL for the specified memory range is allocated. *MemoryStart* and *MemoryLength* are arrays containing the system virtual address and length assigned to the BAR-resources of the board when the driver is loaded. The index of the BAR to map is supplied by the user-mode application in *pBuffer[0]*.

```
case MDS_MMAP_BAR:
   length = 1;
   deviceExtension->Mdl[pBuffer[0]] = IoAllocateMdl(
                                       deviceExtension->MemoryStart[pBuffer[0]],
                                       deviceExtension->MemoryLength[pBuffer[0]],
                                       FALSE, FALSE, NULL);
```

[1] “How To Map Adapter RAM into Process Address Space”: http://support.microsoft.com/kb/189327

The allocated MDL is then updated to describe the physical address range of the BAR-resource, instead of the virtual:

```
MmBuildMdlForNonPagedPool(deviceExtension->Mdl[pBuffer[0]]);
```

The last thing to do is to map this physical address range into user-space. This is done with the *MmNonCached* option, to disable caching. For certain devices caching might improve performance, but depending on the functionality of the BAR, caching is not always applicable. The assigned user-space memory address is stored and returned through *pBuffer[0]*.

```
pBuffer[0] = (((ULONG) PAGE_ALIGN( MmMapLockedPagesSpecifyCache(
                                    deviceExtension->Mdl[pBuffer[0]],
                                    UserMode, MmNonCached, NULL,
                                    FALSE, NormalPagePriority)))
                      + MmGetMdlByteOffset(deviceExtension->Mdl[pBuffer[0]]));
break;
```

When the mapping is no longer needed, it needs to unmapped, and the MDL has to be freed. The user-mode application supplies the index of the BAR to unmap (in *pBuffer[0]*), together with the assigned user-space memory address (in *pBuffer[1]*).

```
case MDS_UNMAP_BAR:
  length = 0;
  MmUnmapLockedPages((PVOID) pBuffer[1], deviceExtension->Mdl[pBuffer[0]]);
  IoFreeMdl(deviceExtension->Mdl[pBuffer[0]]);
  break;
```

3.2 Implementation on Linux

On Linux, mapping device memory to user-space is done through the *mmap()* call from the user-mode application. The *mmap()* call is specified in *asm/mman.h* and is defined as follows:

```
void *mmap(void *start, size_t len, int prot, int flags, int fd, off_t offset);
```

The first argument is the desired memory address in user-space to map the device memory to. This can just be specified to NULL, to let the kernel decide on the mapping itself. The second argument is the number of bytes to map. This should be specified in multiples of the page size (which can be retrieved through the *PAGE_SIZE* macro, specified in *asm/page.h*). The *prot* and *flags* arguments describe the access types and mapping options. The next argument is the file descriptor, and the last argument is the offset of the device to start the mapping at.
To map a complete BAR, the size and offset of it is needed. Relying on hardcoded values is easy but neither failsafe nor flexible, so instead three *ioctl()* calls are implemented. One that sets which BAR to use, and two that retrieve the needed values. The control codes in *s3pcie.h* are:

```
#define S3PCIE_IOGETOFFSET     _IOR(S3PCIE_IOC_MAGIC, 6, int)
#define S3PCIE_IOGETSIZE       _IOR(S3PCIE_IOC_MAGIC, 7, int)
#define S3PCIE_SETBARFORMMAP   _IOW(S3PCIE_IOC_MAGIC, 8, int)
```

To store the selected BAR, a static variable called *current_bar_mmap* is added to the driver. This is set using the following case (simplified):

```
case S3PCIE_SETBARFORMMAP:
  get_user(current_bar_mmap, (int __user *) arg);
  break;
```

Some additional checking is performed to make sure that a valid BAR has been selected (that the selected BAR is within 0-5 and that it is assigned an address different from NULL/0). The two other

cases use the *pci_resource_start / len* calls to obtain the needed information about the BAR registers (also simplified):

```
case S3PCIE_IOGETOFFSET:
   put_user( pci_resource_start(dev,current_bar_mmap)%PAGE_SIZE,
             (int __user *) arg);
   break;

case S3PCIE_IOGETSIZE:
   put_user( pci_resource_len(dev,current_nar_mmap), (int __user *) arg);
   break;
```

The code in the user application that performs the memory mapping is then:

```
int dev;
dev = open("/dev/s3pcie", O_RDRW, S_IRUSR | S_IWUSR);

unsigned int *addr;
int offset, size, map_size;
int bar = <<bar_number>>;

ioctl(dev, S3PCIE_SETBARFORMMAP, &bar);
ioctl(dev, S3PCIE_IOGETOFFSET, &offset);
ioctl(dev, S3PCIE_IOGETSIZE, &size);

map_size = ((offset + size + PAGE_SIZE – 1)/PAGE_SIZE) * PAGE_SIZE;

addr = (unsigned int*) mmap( NULL, (size_t) map_size, PROT_READ|PROT_WRITE,
                             MAP_SHARED, dev, 0);
```

The *map_size* expression looks a bit cryptic, but is simply making sure that the size of the mapping is a multiple of the page size.
In the device driver, the *mmap* call looks quite different:

```
static int s3pcie_mmap(struct file* filp, struct vm_area_struct* vma);
```

This function needs to be added to the file operations structure:

```
static struct file_operations s3pcie_fops =
{
   .owner = THIS_MODULE,
   .mmap  = s3pcie_mmap,
   .ioctl = s3pcie_ioctl,
}
```

In Linux, this kind of memory mapping is handled through a so-called virtual memory area (vma), which is described through the *vm_area_struct* structure. This structure is automatically created by the kernel, from the information passed to the *mmap()* call from the user-mode application. Just like the device driver, the structure also holds an operations structure, with pointers to functions that are called when various events happen. These are not really necessary in this case, but an open and close call that print a debug message when a vma is opened or closed, are implemented anyway:

```
void s3pcie_vma_open(struct vm_area_struct* vma);
void s3pcie_vma_close(struct vm_area_struct* vma);
```

```
static struct vm_operations_struct s3pcie_vm_ops =
{
   .open  = s3pcie_vma_open,
   .close = s3pcie_vma_close,
}

void s3pcie_vma_open(struct vm_area_struct* vma)
{
   printk(KERN_ALERT “s3pcie : VMA Open, virt %1x, phys %1x\n”,
          vma->vm_start, vma->vm_pgoff << PAGE_SHIFT);
}

void s3pcie_vma_close(struct vm_area_struct* vma)
{
   printk(KERN_ALERT “s3pcie : VMA Close\n”);
}
```

The actual *mmap* functionality is up next:

```
static int s3pcie_mmap(struct file* filp, struct vm_area_struct* vma)
{
   printk(KERN_ALERT “s3pcie : MMAP\n”);
   unsigned long vm_size, vm_pgoffset;
```

First, the size and page offset are retrieved from the *vma* structure, and it is set to be noncached:

```
   vm_size = vma->vm_end – vma->vm_start;
   vm_pgoffset = (pci_resource_start(dev,current_bar_mmap) >> PAGE_SHIFT) +
                 vma->vm_pgoff;

   vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);
```

The mapping is done through the *io_remap_pfn_range()* function:

```
   if(io_remap_pfn_range(  vma, vma->vm_start, vm_pgoffset,
                           vm_size, vma->vm_page_prot))
   {
     return –EAGAIN;
   }
```

The last thing to do is to add the operations structure to *vma*, and call the open function:

```
   vma->vm_ops = &s3pcie_vm_ops;
   s3pcie_vma_open(vma);

   return 0;
}
```

3.3 Test of mapping

The created functionality is now tested. This is done in the same way on both Windows and Linux, by mapping an 8 kB blockram in the FPGA to user space. A for-loop in the user application then writes sequential values to the blockram (DWORD address 0x00 = 0, 0x01 = 1, 0x02 = 2, …, 0x800 = 2048). The computer is rebooted (thereby clearing whatever data that might be in the host system memory, but maintaining the power to the FPGA), and the mapping is performed again. The values are now read back from the blockram. It is seen that the values returned match those that were originally written, and it is thus concluded that the mapping indeed provides access to the FPGA blockram, and therefore works as intended.
Further information can be found in the “IOControlDemo” applicationguide.

4 Performance of mapping

The performance of the mapping has been tested, using the same procedure and environments as for testing driver- and application-controlled reading and writing (see the "Performance test" applicationguide). The results can be seen in figure 4.a.

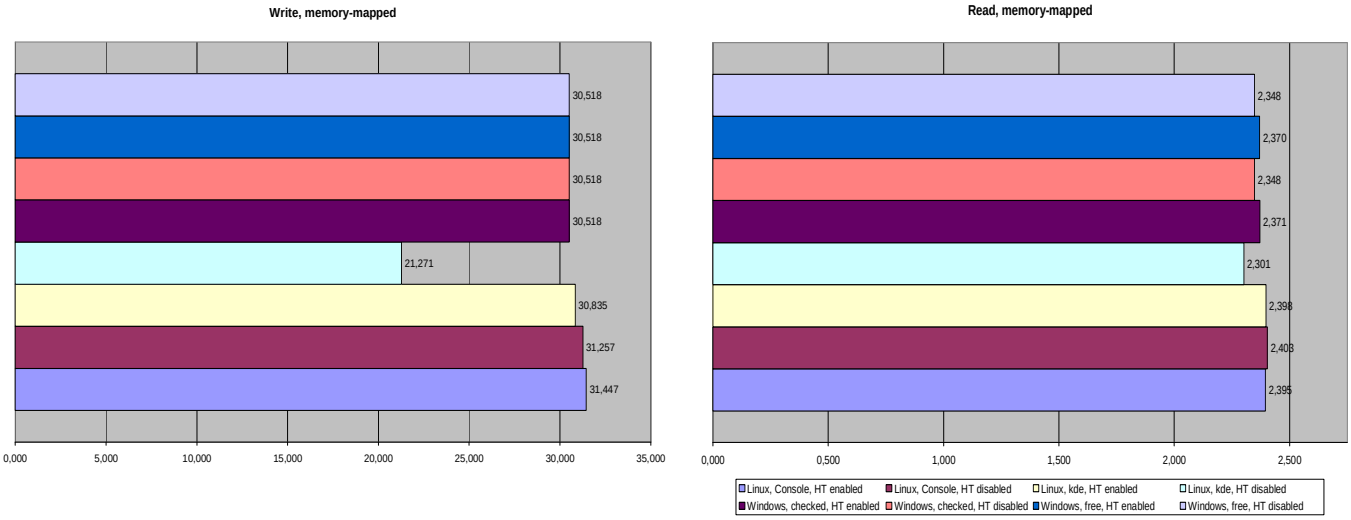

Figure 4.a: The results of the test. All graphs show the effective transfer speeds, values are in MB/s.

From the graphs it can be seen that the performance is almost identical to what was achieved using the driver-controlled approach. Close to 30 MB/s when writing and 2.4 MB/s when reading. This was to be expected, as the memory-mapped approach accesses the memory directly, without any need for driver-requests – just like the driver-controlled approach. The only odd result was experienced on Linux under kde with HT disabled. Both read and write speeds are noticeably lower than in the other environments. The reason for this is unknown, but together with the previous performance test this just goes to show that the effects of HT can be rather unpredictable.
Even so, mapping device memory to the user process has improved both the usability and lowered the overhead (and thereby improved the performance) when performing small transfers.

5 Conclusion

Three ways of improving the transfer speeds have now been investigated. One of these, mapping device memory to user-space, has been implemented. This effectively removes the overhead present in the host OS when performing a driver request, as now only two requests are needed – one to map the memory, and one to unmap it when done. This makes sure that the transfer speed for all sizes of transfers is very close to the best-case 30 MB/s / 2.5 MB/s (write / read respectively) achieved in the previous performance test in journal 4.
To further improve on this performance, DMA functionality will need to be implemented. This has some good potential, as it will allow the PCI Express transactions to have a data payload of more than 1 DWORD, while also offloading the CPU in the host system. As this will require large modifications to both the FPGA design and the drivers, it is left to a future project to implement and prove.