THE MPRACE FRAMEWORK: AN OPEN SOURCE STACK FOR COMMUNICATION WITH CUSTOM FPGA-BASED ACCELERATORS

Guillermo Marcus, Wenxue Gao, Andreas Kugel, Reinhard Männer

Department of Computer Science V,
ZITI - University of Heidelberg
B6, 26 - 68159 Mannheim, Germany
email: {guillermo.marcus,wenxue.gao,andreas.kugel,reinhard.maenner}@ziti.uni-heidelberg.de

ABSTRACT

We present an open source stack for the development of custom FPGA boards, primarily but not limited to PCI Express interconnects. Supporting current Linux distributions, the stack consists of a PCI driver, an IP core for a DMA engine, a hardware abstraction library for IO operations, and a buffer management library for efficient handling of data transfers between an application and a FPGA design. The stack has been validated in diverse hardware and software platforms and provides several building blocks that facilitate the use of accelerators in applications. The DMA Engine IP provides high performance data transfers in PCIe 4-lane boards with Xilinx PCIe cores, with 380 MB/s read and 700 MB/s write maximum measured performance. The buffer management library allows the utilization of 80-95% of this bandwidth with reduced resource consumption and minimal effort.

1. INTRODUCTION

In earlier times, creating an add-on board to a computer was a relatively easy task: Wire the cables, select the address to map to it, connect it and use Peek and Pokes to communicate with it. If high speed transfers were needed, it could be handled by the integrated Direct Memory Access (DMA) controller in the host.

However, as the systems increase in complexity, additional layers of protocols and software are being added: Peripheral Component Interconnect (PCI), PCI Express (PCIe) slots, virtual memory, paging, memory protection. In order to profit from the bandwidth made available by these improvements, standards need to be followed and more complex code needs to be written. In addition, because the devices in the new add-ons are also more complex, the DMA controllers are moved from the mainboard into the device logic. Fig. 1 presents an example of a custom data adquisition board, based in a Field Programmable Gate Arrays (FPGAs).

Because of this increase in complexity, researchers who want to design their custom add-on boards, in particular FPGA boards, have to spend a significant amount of time designing and programming support logic in the FPGA and support code in the host. The goal of this paper is to present a framework, a collection of IP cores and software libraries, that enable developers to focus in their research topics by facilitating the supporting functions required.

Several boards evaluated compromise a device dedicated for high-speed IO (an PLX ASIC or a FPGA) and another device dedicated for the target design (normally a FPGA). Additional resources like on-board memory can be described using the framework functionality. Additional boards, both related and unrelated are also supported. A typical usage of our MPRACE-2 board is shown in Fig. 1.

The following sections document the different sections of our framework: a PCI device driver, the *mprace* IO library and the buffer management library; and presents an overview of the performance with a recent FPGA board in modern hardware.

2. THE PCIE DMA ENGINE

The DMA Engine Intellectual Property (IP) core is responsible for autonomous transfers between the device card and the host computer main memory over a PCIe link. It therefore unloads the CPU in the host from the transfer operations, and improves the performance significantly. This is a non-trivial task, as the memory partition and protection on modern operating systems require careful preparation to be successful and efficient. As comparison, the best Programmed IO (PIO) read performance available is 2.4 MB/s and the best PIO write is 30.5 MB/s, while DMA transfers amount for several hundred MB/s.

We developed the DMA Engine to run on the PCIe transaction layer, as the Xilinx PCIe cores that we use handle the physical and data link layer of the PCIe specification. The design has been validated in four FPGA boards: a Xilinx ML-605 (single Virtex-6), an AVNET V5-LX110T (single Virtex-5), and two custom developed Virtex-4 boards: an ABB (single Virtex-4) and an MPRACE2 (dual Virtex-4).

978-1-4244-8848-3/11/$26.00 ©2011 IEEE

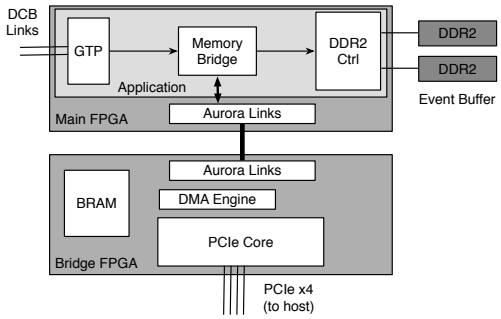

Fig. 1. DMA Engine inside FPGA fabric of a MPRACE-2 board. Typical application for data adquisition, where incoming hit packets via DCB links are put into the event buffer after the arbitration of the memory bridge. The host starts DMA transactions when the event buffer has held enough number of events.

Inside the DMA engine depicted in Fig. 2, three virtual channel buffers are applied for the **Tx** arbitration, **DS_Ch** for downstream DMA channel, **US_Ch** for upstream DMA channel and **MRd_Ch** for PIO read channel. PIO write does not need specific channel buffers because the **MWr** Transaction Level Packet (TLP) of a PIO write transaction is a posted transaction packet and the payload inside it is directed to the memory space. The benefit from channel buffers lies in the possibly larger utilization of the PCIe channels.

The DMA engine has two independent channels of opposite directions, one for upstream (DMA read, from the board to the host), and one for downstream (DMA write, from the host to the board). These two channels run simultaneously and hence, the bandwidth in both directions is well utilized. The DMA Engine can handle time-outs in the case not sufficient data is available for a DMA read transaction. DMA transactions can also be terminated when a time-out happens and is detected by software.

For a DMA operation, the *DONE* status detection is important for efficiency and stability. In our DMA design, *DONE* status can be detected by polling the DMA status register or by servicing interrupts. PIO transactions are also supported and must be supported, because a DMA can only be started by a PIO write, aimed at the DMA control registers. The status register reading in the polling mode is also done via a PIO read.

For a minimum CPU resource consumption during data transfers, scatter/gather DMA is normally used, in which case the DMA transaction is unified as a linked list of descriptors. A DMA descriptor consists of parameters such as the source address, destination address, DMA size in bytes, address to the next descriptor and the DMA mode control. The address for the next descriptor points to the next DMA parameters resident in host memory, so that the DMA engine

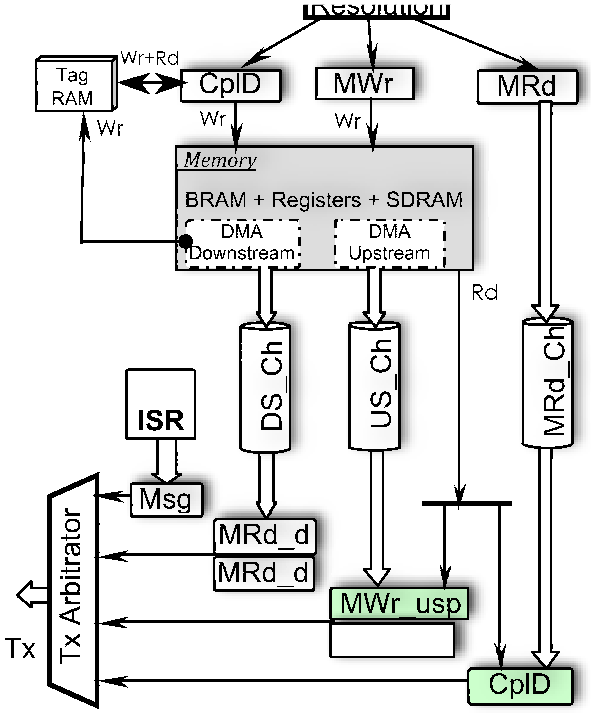

Fig. 2. DMA engine block diagram. DMA as well as PIO is supported in this application. DMA channels and PIO transactions are multiplexed at TX arbitration, with the help of channel buffers inside the DMA engine. ISR corresponds to Interrupt Status Register, which triggers the Msg sending, especially for the DMA DONE status acknowledge. The green blocks contain payload in their TLPs.

can fetch the next DMA command after the current DMA is *DONE*. In this way, the DMA transfer is still automatic while still providing the flexibility needed by the user and the operating system.

In a Virtex-4 dedicated to IO like in the MPRACE-2 design, the full design (PCIe core + DMA Engine + Aurora links to the computing FPGA) consumes 15926 4-input LUTs (8541 SLICEs).

3. THE PCI DRIVER

In modern operating systems, the function of a device driver is to provide the logic needed to communicate the devices connected to the system with the applications using them. The PCI driver fits on our software stack at the lowest level, providing the OS dependent functions that makes most of the remaining libraries OS independent.

The driver supports multiple, generic PCI devices in 32- and 64-bits systems, and provides mapping of PCI(e) base address registers (BARs), access to the device configuration space, memory management for kernel- and user- space

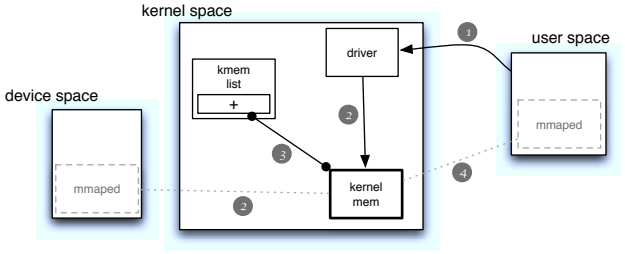

Fig. 3. Typical kernel memory allocation

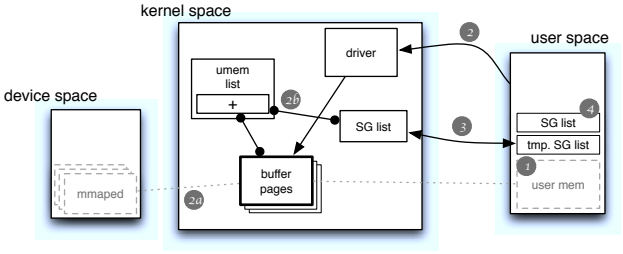

Fig. 4. Preparation of a user memory buffer for access by the device

memory, hotplug capability, dynamic char device allocation, configuration via SysFS and a C / C++ user interface, as well as a compatibility layer for applications using our old driver.

The basic infrastructure of the driver follow the guidelines of the book Linux Device Drivers[1]. Traditional Linux device drivers can be described as monolithic drivers: They encapsulate all their functionality inside an opaque interface, exporting information to the device and application using well defined interfaces. The driver code executes exclusively in kernel space, and is expected to manage all operations related to the device from this position.

In contrast, our device driver seeks to be a generic PCI driver. We try to leave most of the logic in user space, exporting the required kernel structures and functions when needed. This creates in practice an hybrid driver, with support functions in kernel space for operations that cannot be done otherwise, and leaving all device logic on user space. This approach has given good results in the past[2][3], and is the method used by commercial solutions like Jungo[4], PLX[5], NVIDIA[6] and AMD[7].

Therefore, the driver structure can be divided in two big, separate blocks: a kernel driver, which we call the driver, and a user space interface which we call the Application Programming Interface (API). The driver takes care of device initialization, memory and IO mapping to both device and user space, interrupts and SysFS interface. The API abstracts the functionality provided by the kernel driver, making the interface platform and version independent.

In Fig. 3 we can see a simplified representation of a kernel memory allocation. First, (1) the application running in user space requests the driver a new kernel memory buffer of a fixed size. Then, (2) the driver allocates the memory, and during the same process maps the area to the device space, making the buffer available to the device. Next, (3) it adds a new entry to the `kmem_list` list for management purposes. The driver finally returns to the application with the device address (to pass via commands to the device) and an ID of the buffer. Finally, in a separate step, (4) the buffer is mapped to the user space for access by the application.

Memory allocated in user space differs significantly from kernel memory in that it is not a contiguous block, but a collection of segments of physical memory (as small as a page) ordered virtually in user space via mapping. The list describing the mapping of the segments is called a scatter/gather (SG) list. From the user point of view, this mapping is transparent: the application accesses the memory as a contiguous block in its user space. It is even possible (actually, quite common), that a page initialization being delayed by the OS until first accessed (via a minor page fault) or that a page be stored swap memory on disk.

However, in order for the device to access a user space buffer, all pages of a buffer must be available, mapped into memory and into device space. But the mapping to device space is not contiguous as in user space, mostly because some platforms share the memory and device buses. This means the device needs a SG list for device space, which in turn means each physical memory segment must be mapped to the device space in order. The list is then transferred to the device as needed for accessing the buffer. This is done by the device driver following the procedure in Fig. 4, detailed as follows:

On the first place, (1) the buffer is allocated by the application or by the driver library on user space. Next, (2a) the buffer pages are locked in memory and mapped by the driver into device space and (2b) a new SG list is created, along an entry into the `umem_list`. The driver then returns to the library basic information, like the size of the SG list and handle ID. Afterwards, (3) the library calls the driver in order to retrieve the SG entries of the list. This list is (4) copied into another structure inside the library, and then discarded. The application can now pass to the device, as needed, the provided SG list which contains device addresses for every entry.

Our driver registers an interrupt handler for each device during initialization, and creates per device a wait queue for each possible interrupt source (every device is assigned a maximum number of sources). When the application requests to wait for an interrupt, it makes an IOctl call to the driver. The driver then waits for an event in the corresponding event queue, and sends the process to sleep. When the correct interrupt source arrives, the interrupt handler sends an event to the wait queue. This will awake the process, but not immediately. Instead, it will reschedule the process,

which will be executed on the next scheduler round after the interrupt handler is finalized. This allows for the handler to finalize fast and cleanly, and to resume the application process without complications. The drawback is that the interrupt acknowledge is not generic, it has to be modified to support every additional device, as every device requires an unique response. This is the only critical point in the driver that requires device specific code, beside the normal initialization IDs, but provides very fast release of interrupts, which is critical for several of the applications we intend to support, like high-throughput data acquisition.

The C++ interface reorganizes the IOctl calls around 3 basic classes: *PciDevice*, *KernelMemory* and *UserMemory*. Both *KernelMemory* and *UserMemory* represent the memory buffers which can be created, and provide functions to get the mapping information needed for a device to access them. However, their creation is handled directly by the *PciDevice* class, which also provides all other functions, like IO BAR mappings and PCI configuration space. An additional C interface is available, which follows the same structure.

4. THE MPRACE LIBRARY

The MPRACE library is the next level library in our software stack. It sits directly above the PCI driver, and provides additional, higher functionality to our FPGA boards. Its main purpose is to provide common operations used by applications, operations like register IO, DMA transfers and FPGA configuration. In this sense, the MPRACE library is the successor of our uelib library, which provided similar functionality to our older boards: the μEnable, ATLANTIS and MPRACE-1.

From the application point of view, the library abstracts the functionality of most of the common cores added to our designs, allowing the user to concentrate in the development of FPGA applications and software instead of routine tasks. While register IO is a relatively simple task, boards like the MPRACE-2 have a DMA engine capable of SG transfers, an operation that requires significant initialization on the software host side. This is very much simplified when filling a specially allocated buffer and instructing the board to transfer it.

Register read and write are the most basic IO operation that any system can perform, but even such a simple task can be performed in several different ways. In the most basic mapping, a register behaves exactly as a memory location: writing to it sets the register to a new value, reading from it returns its current value. Some architectures handle this as a completely separate address space (i.e. the SPARC architecture), while others handle registers as memory mapped areas, where a memory address is catched and redirected to access the register instead. A similar method is used to map whole areas of a PCI device into the PCI bus: the device provides one or more base address registers (BARs) and their sizes in the device configuration area, each which are assigned (mapped) during boot to an non-colliding address range in the PCI bus address space. Then it is the task of the operating system and the driver to map the BARs and make them accessible to the application. This library maps these areas into the application memory space and uses wrapper functions to provide a clear and consistent interface for the user: namely a *getRegister()* and *setRegister()* pair of functions.

A *DMABuffer* class gives a consistent interface to two basic, dissimilar entities: the Kernel Memory buffer and the User Memory buffer, both structures provided by the underlying PCI driver. While the mapping to user space is very clear and comes in the form of a regular pointer, the additional data required by the DMA Engine to perform a transfer is very different on each case. The *DMABuffer* class provides encapsulation of this data and a convenient way to pass it to the library on each transfer request.

Because a kernel buffer is by definition a contiguous area, a user pointer, a physical address and a size is enough to completely describe the buffer and its mapping into user space, and all of them are provided by the *KernelMemory* class of the PCI driver. On the other hand, a user memory buffer is composed of a collection of memory pages. From user space it is a single contiguous area, but the sequence of the physical pages involved is defined in its SG list. However, the SG list provided by the driver is not guaranteed to be neither available nor arranged in a way compatible with whatever structure is required by the DMA Engine used by the board. Therefore, we abstract this additional mapping with a new *DMADescriptorList* class.

The *DMADescriptorList* class is a virtual class, which simply abstracts the access methods for an array of hypothetical *DMADescriptor* elements. What really interests us is the *DMADescriptorListWG* class, which implements it for the *DMAEngineWG*, as shown in Fig. 5. Each element of the *DMADescriptorListWG* array is of type *DMADescriptorWG*. The class actually composites two arrays. One array is a linear array of *DMADescriptorWG* elements, where each element references to a native descriptor (section C). This is the array that is accessed regularly when manipulating the descriptor list. The second array is used to store the native descriptors as required by the board, and organized as a collection of blocks where each block is a single memory page from an independent DMA Buffer, this is the section A. This means the native descriptor list is stored in a DMA buffer which is accessible by the board. Each block links a native descriptor to a element of the SG list, providing the board access to the list required for processing a DMA transfer, while at the same time linking the necessary data structures to the *DMABuffer* being transferred. The

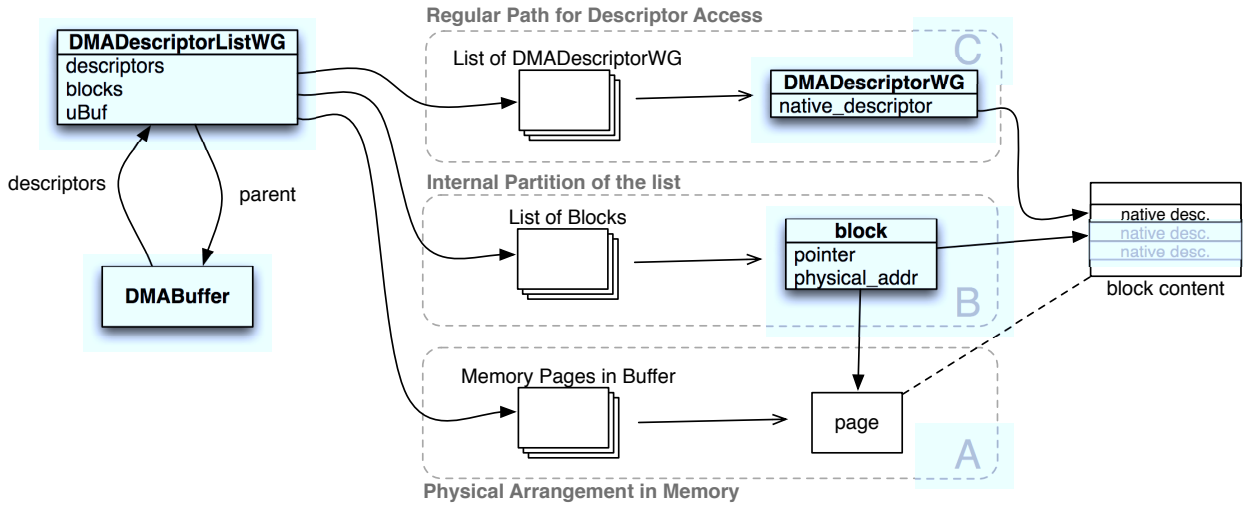

Fig. 5. DMA Descriptor List diagram

DMADescriptorListWG limits the number of *DMADescriptorWG*s that can be stored per block in order to guarantee that there is neither a partial descriptor in a block nor the descriptors cross a page memory boundary. This relationship is depicted in section B. Section B is initialized by the *DMAEngineWG*, and once initialized, allows for fast manipulation of the descriptors.

5. THE BUFFER MANAGER LIBRARY

To achieve the closest match to the theoretical performance maximum of a communication channel with the minimum resource usage has been for a long time a goal of device driver and application developers. Using diverse buffer management algorithms, like the partition of a buffer in chunks or using double buffering to achieve these goals, has been done for years. However, FPGAs pose a particular challenge as their communication patterns are determined by the design loaded, precluding optimizations of data transformation. Then, the application must take care of these optimizations. The Buffer Management library provides a framework to facilitate this task.

The Buffer Management Library (*bufmgr* for short) provides optional functionality for efficient use of the DMA buffers and DMA transfers. It provides a common interface to transfer a certain buffer in memory using only a limited amount of buffers efficiently, following certain known buffer algorithms. In addition, it provides the ability to decrease the number of copy operations needed when data has to be converted to a certain format before or after the transfer, by merging the data transformation into the data copy loop of the selected algorithm.

Our previous bufmgr library was implemented as a collection of classes, as it was presented in [8]. The current extension adds templatized classes to merge the data transformation, the buffering algorithm and the underlying transfer library, which is all now modular. These changes make it more flexible, allow the compiler to optimize the results in a better way and allows a broader range of interfaces to be used. Being able to switch the transfer functions easily allows the library to be used with our old *uelib* library, the new *mprace* library, or any other function set to handle the data transfers (by example, CUDA pinned memory for GPU transfers).

6. PERFORMANCE

While tests have been performed in a variety of hosts and FPGA cards, the results presented here are from a Intel Xeon E5420 @ 2.5GHz with 4 GB of RAM, and a Virtex-5 AVNET V5-LX110T evaluation kit with a 4-lane PCIe interconnect. Results for the non-templatized buffer managers are already presented at [8], and the templatized results for the MPRACE-1 are extremely similar, so they are not included in this paper. Instead, we focus our analysis with the new IO library and the higher bandwidth boards.

Fig. 6 summarizes the performance for the DMA write operations. The mprace curve is the performance of the *mprace* IO library which represents the maximum we could achieve with a 4-lane PCIe link with a maximum bandwidth of 10 Gbit/s. As the link performs 10/8 bit encoding, the maximum attainable transfer is limited to around 1 GB/s. From the protocol overhead and from our experience with other 4-lane PCIe boards, a maximum of 800 MB/s is reachable. Therefore, a 700 MB/s result for DMA writes can be considered a good result. An additional variance of 2-10% can be seen as a consequence of the chipset/platform used.

The results for the buffer managers provides a differ-

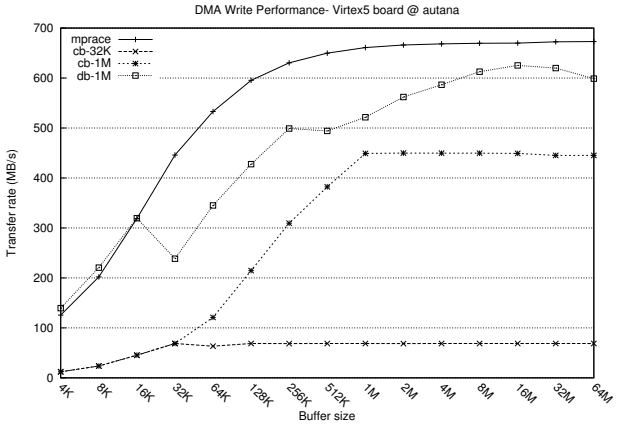

Fig. 6. DMA Write performance of the mprace library in a Virtex-5 board, as well as Chunk Buffers (cb) and Double Buffers(db). The number in the label name represents the buffer size used.

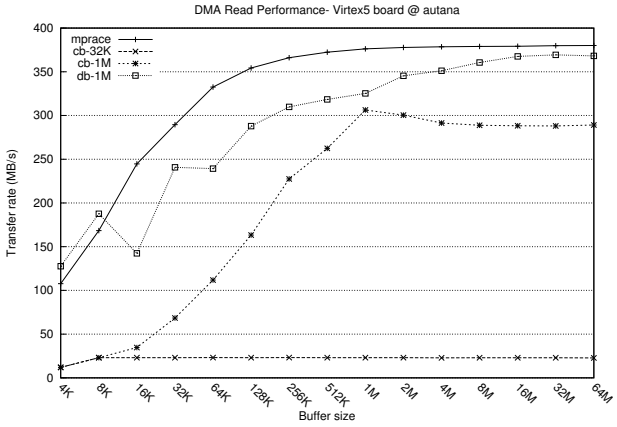

Fig. 7. DMA Read performance of the mprace library in a Virtex-5 board, as well as Chunk Buffers (cb) and Double Buffers(db). The number in the label name represents the buffer size used.

ent picture. In particular, the overhead of the algorithms is higher in comparison with the previous results, because the bandwidth involved is significantly higher. In addition, a bigger transfer is needed to achieve a higher transfer rate, which sets a penalty for smaller transfers and a limitation in the minimum amount that can be used with the buffer managers. Therefore, using a Chunk Buffer has a significant overhead, which is almost eliminated for big transfers when using a Double Buffer, achieving almost 95%. In order to improve the performance further for smaller transfers, we are implementing a templatized Pooled Buffer as documented before.

Similarly, Fig. 7 summarizes the performance for DMA read operations. Here, the maximum performance of the library is limited to 380 MB/s, which seems to be based in the FPGA in our current designs, as using simpler test engines provides near 800 MB/s. The buffer managers present similar behaviour as with the case of a DMA write, where the Chunk Buffer shows a overhead until it saturates for the chunk size, while the Double Buffer allows better transfer rates, up to 95% of the total. It is expected that Pooled Buffers will improve the performance for smaller transfers over the Double Buffer scheme.

7. AVAILABILITY

The source code is available under the GPLv2 license at our website *http://li5.ziti.uni-heidelberg.de/mprace* .

8. CONCLUSION

As the hardware interfaces increase in complexity, so does the required software to fully utilize the available bandwidth. The libraries made available with this framework make the development of a new custom board much simpler, by providing the IP cores and a big portion of the code necessary to interface with it efficiently. With performance in the range of 700 MB/s for DMA writes and 380 MB/s for DMA reads, it promotes experimentation and gives researchers the tools to concentrate in their topics instead of the surrounding implementation details.

9. REFERENCES

[1] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*. O'Reilly Media, 2005.

[2] C. Hinkelbein, A. Kugel, R. Männer, and M. Müller, "Reconfigurable hardware control software," in *RSP '02: Proceedings of the 13th IEEE International Workshop on Rapid System Prototyping (RSP'02)*. Washington, DC, USA: IEEE Computer Society, 2002, p. 84.

[3] M. Müller, "Evaluation of an fpga and pci bus based readout buffer for the atlas experiment," Ph.D. dissertation, 2004.

[4] Jungo, "Driver development toolkit," http://www.jungo.com/.

[5] PLX, "Pci io accelerators," "http://www.plxtech.com/products/io/".

[6] NVIDIA, "Nvidia unified driver architecture," http://www.nvidia.com/object/feature_uda.html.

[7] AMD, "Ati catalyst technology," "http://www.amd.com/us/products/technologies/ati-catalyst/Pages/catalyst.aspx".

[8] G. Marcus, G. Lienhart, A. Kugel, and R. Männer, "On buffer management strategies for high performance computing with reconfigurable hardware," *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pp. 1 – 6, Aug 2006.