ffLink: A Lightweight High-Performance Open-Source PCI Express Gen3 Interface for Reconfigurable Accelerators

David de la Chevallerie
Embedded Systems and Applications Group
Hochschulstraße 10
64289 Darmstadt, Germany
dc@esa.cs.tu-darmstadt.de

Jens Korinth
Embedded Systems and Applications Group
Hochschulstraße 10
64289 Darmstadt, Germany
jk@esa.cs.tu-darmstadt.de

Andreas Koch
Embedded Systems and Applications Group
Hochschulstraße 10
64289 Darmstadt, Germany
ak@esa.cs.tu-darmstadt.de

ABSTRACT

We describe the architecture and implementation of ffLink, a high-performance PCIe Gen3 interface for attaching reconfigurable accelerators on Xilinx Virtex 7 FPGA devices to Linux-based hosts. ffLink encompasses both hardware as well as flexible operating system components that allow a tailoring of the infrastructure to the specific data transfer needs of the application. When configured to use multiple DMA engines to hide transfer latencies, ffLink achieves a throughput of up to 7 GB/s, which is 95% of the maximum throughput of an eight-lane PCIe interface, while requiring just 11% of device area on a mid-size FPGA.

Keywords

Heterogeneous Computing, High Throughput, FPGA, VC709, PCI Express, PCIe Gen3, Intel64, AXI4, DMA, Double-Buffering, Linux-Driver

1. INTRODUCTION

Despite promising work on next-generation accelerator interconnects such as NVLink [2], PCIe remains the most widely used method of connecting accelerators to host computers. Reconfigurable accelerators face the challenge, that the interface functionality provided by hardwired IP blocks on recent reconfigurable devices only covers the lower protocol layers, and significant engineering effort has to be expended to actually allow high-throughput data transfers between host and accelerator.

We present ffLink, the first open-source solution aiming to fully support the PCIe Gen3 IP block integrated in Xilinx Virtex 7 devices. Using the eight lane (x8) configuration commonly used on current FPGA accelerator boards, ffLink achieves transfer rates of up to 7 GB/s. The hardware side of ffLink was carefully designed to maximally reuse the well-supported Xilinx IP library, while software-side device drivers for Linux x86-64 provide multiple data transfer mechanisms to match the needs of the current application. On the FPGA, even a fully-scaled system encompassing multiple parallel DMA engines requires significantly less chip area than optimized commercial (closed-source) PCIe Gen3 interfaces.

This work was presented in part at the international symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2015) Boston, MA, USA, June 1-2, 2015.

2. RELATED WORK

For the prior versions of PCIe a wide spectrum of implementations for FPGAs exists. An early Gen1 design is described in [11], while Speedy [5] is typical of early Gen2 work. A key effort to make Gen2 interfaces more practical is RIFFA [7], which formed the base for refinements by other researchers, e.g., an extension to allow both streaming and random accesses [10]. A similar effort is EPEE [6], while Kavianipour et al. [8] concentrated only on streaming accesses (as did the original RIFFA). Performance-wise, these Gen2 systems achieve throughputs of 750 MB/s to 3.8 GB/s for x8 bus configurations.

As the Gen3 interface block in the Xilinx Virtex 7 devices is completely different than the Gen2 block used in prior chips, no academic or open-source efforts to advance to Gen3 have been published by the time this work is submitted. True Gen3 interfaces for FPGAs have so far solely been the domain of commercial efforts. Northwest Logic [4] provides a closed-source IP core for the Xilinx XC7VX690T FPGA, used, e.g., on the Xilinx VC709 evaluation board [14], which requires 31-44% (depending on configuration) of the FPGA area. The IP block is documented to reach up to 6.8 GB/s. In practice, the two reference designs provided with the VC709 that use the block achieve transfer rates of 5.6 and 6.0 GB/s in our host machine. The commercial Xillybus interface [1] is considerably smaller, but tops out at just 800 MB/s even in a Gen3 x8 configuration.

High-performance PCIe interfaces always require Direct Memory Access (DMA) mechanisms, as programmed I/O (PIO), where the processor itself transfers each datum, is highly inefficient and typically limits throughputs to just 11 MB/s (read from accelerator) and 38 MB/s (write to accelerator). A key difference in the realization of the DMA mechanisms lies in the use of scatter/gather (the engine is able to collect/distribute data from/to non-contiguous physical address ranges in a single transfer) versus buffering (data is retrieved/deposited only from/to a physically contiguous buffer by the engine and *copied* to non-contiguous physical address ranges in software). While scatter/gather capability is highly efficient, it requires complex custom-designed DMA engines taking up much chip area (e.g., the Northwest Logic approach). We will show that it is possible to achieve a similar level of performance using a good combination of multiple small, off-the-shelf 32b DMA engines from the Xilinx IP library and a double-buffering scheme [15, 9, 10] to hide transfer latencies.

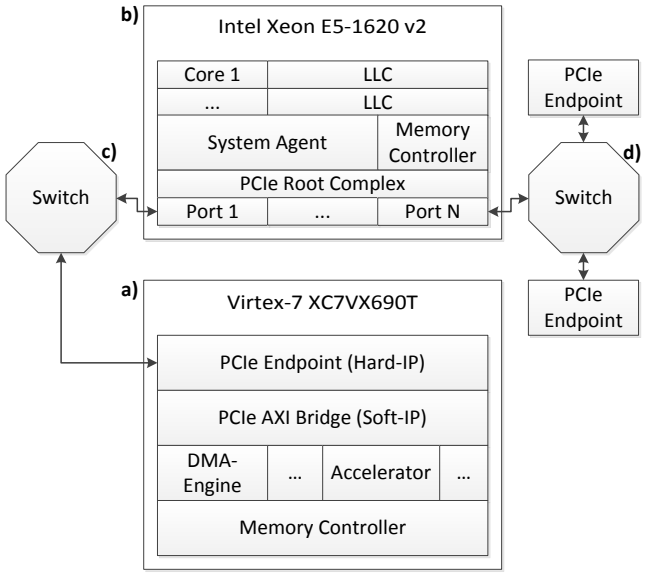

Figure 1: Overall system architecture

3. SYSTEM ARCHITECTURE OVERVIEW

In current computer systems (Figure 1), PCIe is realized as an interconnection of multiple communication partners (End Points) over switches (Figure 1.c and .d) to individual Ports, which belong to a single Root Complex (Figure 1.b).

For high-throughput and low-latency, a Root Complex is typically integrated on the CPU. It not only initiates transfers from/to its associated End Points under CPU control (PIO), but also makes the memory attached to the CPU(s) available to End Points capable of initiating their own accesses (acting as DMA *bus masters*).

On the FPGA, the actual electrical-level (PHY) End Point of a Gen 3 interface is typically implemented as a hardwired IP block (Figure 1.a). It communicates upwards in a packet-based protocol (Transaction Layer Packets, TLP), which is then translated to the on-chip bus interfaces (today often ARM AMBA AXI) by a bridge IP block implemented in configurable logic. The FPGA memory controller(s), DMA engine(s), and the command/status registers of the accelerator(s) themselves are thus connected using their native AXI interfaces to the AXI-TLP bridge. The detailed composition of these components is discussed in Section 5.

On the software side, the accelerator(s) and data-transfer infrastructure are made visible to the Linux operating system using a device driver (see Section 6).

4. PCI EXPRESS FUNDAMENTALS

The PCIe protocol is organized in three layers, with each layer adding a wrapper to the payload (Figure 2). The lowest physical layer (PHY) is responsible for the electrical transfer of data and the establishment of the link between two devices. Here, the link width and speed are negotiated, e.g., a connection using eight lanes with 8 GT/s per lane (Gen3 x8). Each lane is a bidirectional channel with two differential wires per direction.

Data is transmitted in Gen3 using a 128b/130b encoding scheme (18% more efficient than the 8b/10b scheme used in Gen2). The wrapper overhead consists of 4 B of Start-of-

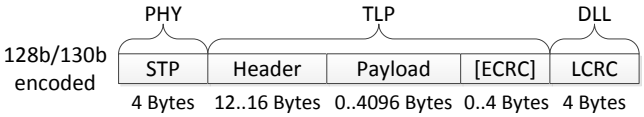

Figure 2: PCIe protocol layers

Packet (STP) data, a header of 12 B (16 B when using 64b addresses), and 4...8 B total of checksums (optional ECRC and mandatory LCRC), yielding a minimal overhead of 20 B per 0...4096 B of actual payload.

The Data Link Layer (DLL) ensures reliable transport between End Point and Root Complex, while the topmost Transaction Layer Packets transport memory and I/O accesses, as well as configuration data for the link itself (e.g., the maximum payload size, MPS). TLPs are distinguished into two kinds of requests: posted and non-posted. For posted transactions the transmitter does not expect a completion (reply), e.g., memory write, whereas a completion is expected for non-posted requests, e.g., memory read.

5. HARDWARE ARCHITECTURE

Figure 3 shows the on-chip architecture typical of systems using ffLink, which can be assembled in an automated fashion by Tcl scripting the Xilinx IP Integrator tool.

The AXI Bridge for PCIe Gen3 IP core [13] (Figure 3.b) translates PCIe transactions on the bus to corresponding AXI4 transactions. The target clock frequency of the design is 250 MHz, thus requiring a 256b wide AXI4 bus to cope with the theoretical maximum PCIe Gen3 throughput of 8 GB/s (eight serial lanes, 8 GT/s each). The bridge can operate with 32...64b bus addresses. We will use 32b addresses, which provides six Base Address Registers (BAR) to map distinct address ranges between PCIe bus addresses and on-board addresses on the accelerator card (e.g., memory banks, control/status registers etc.). PIO accesses from the CPU will thus use the bridge's M_AXI master port and the Register Interconnect (Figure 3.c) to reach the corresponding AXI4 slave on the card.

The bridge also offers a slave port S_AXI to allow AXI4 bus master-capable IP blocks (such as DMA engines or the accelerator itself) to generate PCIe transactions, e.g., to access host memory. An additional S_AXI_LITE interface is provided to read/write status/control data from/to the PCIe End Point IP block itself (e.g., actual link speed, interrupt mask etc.).

The system-on-chip (DMA engines, accelerator; Figure 3.d and .e) can use Message Signaled Interrupts (MSI) for communication with the host, selecting one of up to 32 interrupt vectors to activate. Note that for PCIe, interrupts are no longer transported as levels/edges on separate wires, but instead as special TLP packets in-band with the normal bus traffic. An on-chip Interrupt Controller (Figure 3.i) was developed to gather the interrupts from multiple sources (DMA engines, accelerators) and set the MSI vector in the PCIe IP core accordingly. If multiple interrupts arrive in parallel, a source is chosen in a priority-based manner. As soon as the PCIe block has accepted the interrupt, the next MSI can be issued. It is actually possible for MSIs to get lost due to starvation, if the bus is completely loaded with data traffic. The PCIe block will silently discard a pending MSI if it could not actually send the corresponding TLP

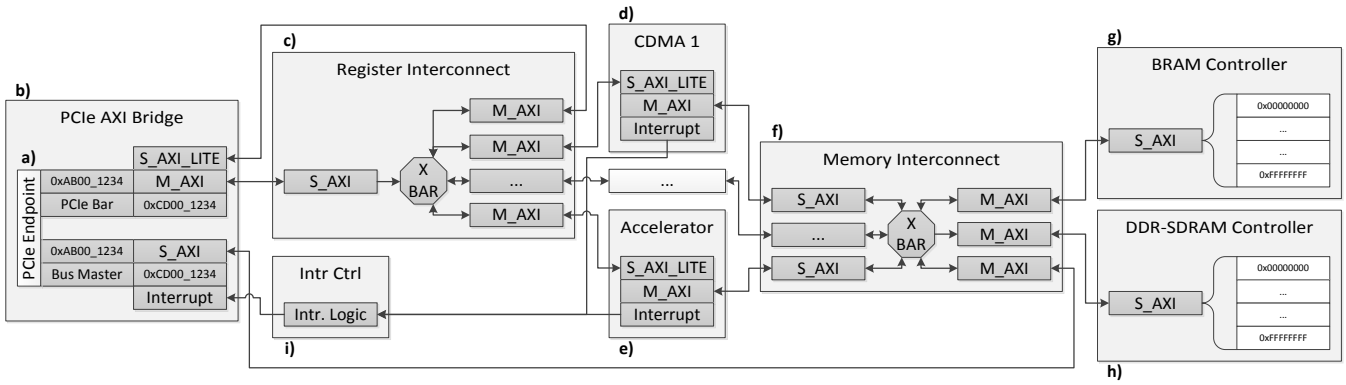

Figure 3: System-on-chip architecture

packet after an unspecified timeout. The on-chip Interrupt Controller has thus been extended to re-issue the MSI if it has not been acknowledged by the host within a set time interval, thus enabling reliable (if possibly delayed) interrupt delivery.

Following the idea of maximally re-using existing IP, to reduce development time and profit from vendor library updates, the AXI Central Direct Memory Access (CDMA [12]) block was chosen from the Xilinx catalog as DMA engine (Figure 3.e). It reaches up to 99 % bandwidth utilization of the AXI protocol. The CDMA block has a maximum address width of 32b, which also determines the address width of the PCIe block. The CDMA block has access to both on-chip (Figure 3.g) and off-chip memory (Figure 3.h). It uses the Memory Interconnect (Figure 3.f) to transfer memory data to/from the host using the AXI bridge slave port, which controls the bus-mastering of the PCIe End Point (Figure 3.a). After completing a data transfer, a CDMA block raises an interrupt to signal the host.

Note that the system can flexibly scale to multiple CDMA engines, which will be used to hide engine-setup or interrupt latencies (see Section 8.2.2). In the current structure of one Interconnect each for control/status register and memory accesses, a mix of up to 16 CDMA engines and accelerators can be implemented. For larger numbers, multiple Interconnects may be instantiated, allowing a higher clock frequency at the cost of access latency.

6. OPERATING SYSTEM INTEGRATION

6.1 Virtual and Physical Addressing

On the software side, Linux running on an x86_64 system uses two 47b (128 TB) *virtual* address spaces (Figure 4). One for each user process, the second one for the operating system kernel. The latter is subdivided to, e.g., directly map up to 64 TB of *physical* memory (all physical memory excluding zone_highmem), or directly map-in I/O devices from physical to kernel virtual memory using ioremap. The address ranges set by the BARs of a PCIe device are mapped into the physical address space at host boot time, and can then be selectively mapped into the kernel virtual space by the device driver. PIO can then be performed on these virtual address ranges to access the device (which acts as a slave in this case).

A different mechanism is used when the PCIe device (acting as bus master) wants to access data in host memory.

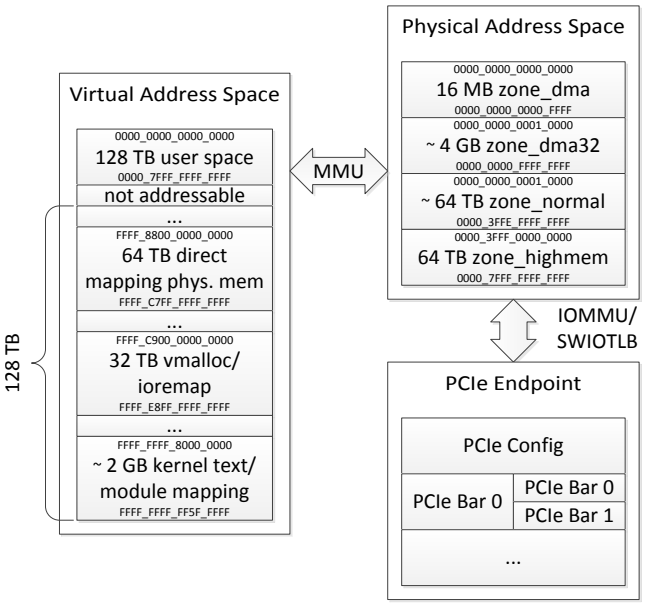

Figure 4: Linux x86-64 address map

Since not all PCIe devices are capable of 64b addressing (which includes ffLink with its use of stock 32b CDMA blocks), some care must be taken to ensure that data will actually be accessible to the device. A small number of hosts (mainly using CPUs by AMD) provide a general solution to this problem by employing a full-scale I/O Memory Management Unit (IOMMU) that allows arbitrary remapping of I/O accesses within the entire physical memory space. The more common Intel x86_64 processors lack such an IOMMU and instead emulate some of its functionality by Software I/O Translation Buffers (SWIOTLB). This technique relies on *bounce buffers* allocated at boot time in low physical address ranges, guaranteed to be accessible by the device (e.g., within the first 16 MB for 24b devices in zone_dma, within the first 4 GB for 32b devices in zone_dma32). When data is staged to be accessible by the device, it is actually *copied* to the bounce buffer if its physical address would be inaccessible to the device. For 32b devices, a typical size of the bounce buffer is 64 MB (larger buffers are possible, but have a higher set-up overhead).

As data will often be copied (usually, see below for a zero-

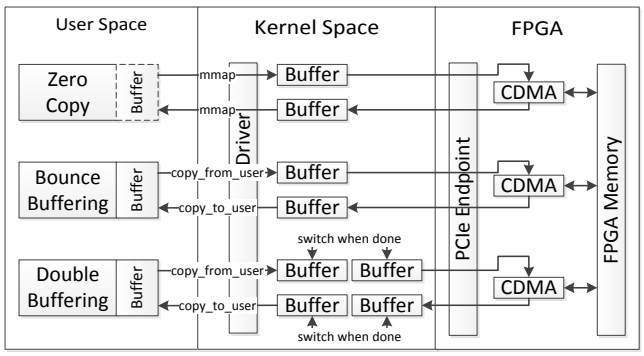

Figure 5: Driver transfer methods

copy approach) when using a bounce buffer, adding scatter/-gather capability to a 32b DMA engine is not very efficient. It would increase its complexity, but not help in avoiding the copying in the general case.

Instead, data to be transferred is copied by the CPU from user space to a contiguous address range within the bounce buffer, which can then be accessed in a single transfer using a simple (and small) DMA engine. The default memory allocator in the Linux kernel can easily provide such contiguous regions on request, but is limited to a maximum size of 4 MB per allocation, which in turn limits the maximum size of a single DMA transfer to 4 MB. However, as will be shown in Section 8.2.1, this restriction does not impede the maximal transfer rate, and thus makes more complicated means to lift it (e.g., the use of huge pages) unnecessary.

6.2 Data Transfer Mechanisms

The ffLink device driver is highly flexible. It can operate in three different transfer modes (Figure 5) and also allows scaling the number of CDMA engines. Due to the small size of each simple engine, even configurations of four engines remain highly area efficient. Each engine appears as a separate mutex-protected device node in the Linux file system, ensuring serialization if multiple processes attempt concurrent accesses.

Zero Copy: This approach can always avoid the extra copying of data from user space to a bounce buffer. Instead, device-accessible memory is directly `mmap`ed into user space. Once the user program has finished operating on the data, the actual transfer to the accelerator is initiated using an `ioctl` call, which starts the DMA transfer for the given number of bytes and also flushes/invalidates the CPU cache for the buffer. This transfer mode has the lowest latency, but requires multiple transfers for sizes exceeding 4 MB and requires a separate buffer per process using the accelerator in this manner.

Bounce Buffering: This is the conventional SWIOTLB approach, using two separate buffers for reading/writing, both located in device-accessible memory. The buffers are present only in kernel space (and can thus be shared between user processes), with actual CDMA data transfers initiated by `read`/`write` system calls after the accelerator address has been set using `ioctl`. The driver automatically copies data between the buffers and user space, and splits larger transfers into 4 MB chunks.

Double Buffering: This employs the same API as the previous mechanism, but uses *two* pairs of buffers. The driver can thus perform the copying of data from/to user space on one buffer in parallel with the DMAing of data from/to the accelerator. This method should be selected for larger transfers, where the overhead of copying data can be hidden behind the DMA operation, thus justifying the additional allocation of more device-accessible memory.

Table 1: Utilization of Xilinx XC7VX690T FPGA for different CDMA and memory configurations at Gen3 x8

Configuration	**LUT** [%]	**FF** [%]	**BRAM** [%]
1 CDMA/ BRAM memory	3	5	9
1 CDMA/ DDR3 memory	5	9	6
4 CDMA/ BRAM memory	5	8	11
4 CDMA/ DDR3 memory	7	11	7
Commercial *Loopback*	11	31	6
Commercial *Base*	20	44	18

7. CHALLENGES

A key difficulty in designing ffLink were the significant changes Xilinx made to the interfaces from the Gen2 to the Gen3 interface block, which rendered much prior work inapplicable. Without very costly measurement equipment (e.g., external bus protocol analyzers), in-system debugging of the core had to rely on the Xilinx Integrated Logic Analyzer core, which, however, does not have sufficient signal-capturing depth to effectively diagnose intermittent failures (such as the loss of MSI packets) that only occur during large transfers.

Tuning the ffLink for high-performance was also non-trivial. As an example, the default configuration of the AXI interconnect allows at most two outstanding requests, which leads to the DMA engines not being able to saturate the PCIe-AXI bridge, limiting transfer rates to less than 5 GB/s. Only by manually configuring the AXI interconnect IP the number of in-flight requests can be increased to four, which alleviates the problem. It is not solved completely, as multiple DMA engines can saturate even the widened interconnect.

8. EXPERIMENTAL RESULTS

8.1 FPGA Area Utilization

Table 1 shows the area utilization of different ffLink configurations and of the commercial IP [4], which has similar performance for a Gen3 x8 use-case. The BRAM-based versions omit the memory controller for access to external memory (this is similar to the *Loopback* configuration of the commercial solution), while the DDR3 versions do support external memory instead (similar to the commercial *Base* configuration). Note that ffLink is much smaller than the commercial IP, leaving more logic on the FPGA for the actual accelerator(s).

8.2 Throughput Measurements

We evaluate ffLink throughput at different transfer sizes using 1, 2, and 4 CDMA engines in parallel, each controlled by a separate software thread (using the pthread library). To obtain accurate results, each test executed transfers for

at least 5 ms, and was repeated 100 times, with the best result used for further analysis.

All tests were run on Linux kernel 3.17 and compiled by gcc 4.8.3 with the -O3 flag. The software is executed on a Intel Xeon E5-1620 v2 CPU, consisting of four cores, activated Hyper-Threading, and a base clock frequency of 3.7 GHz (boosted up to 3.9 GHz). The hardware was built using Vivado 2014.4 for the Xilinx VC709 Evaluation Board. The circuits were clocked at 250 MHz, only the DDR3-SDRAM controller uses 200 MHz. However, since the controller supports 512b wide accesses, it will never be a bottleneck for throughput.

Even with the theoretical throughput tp_{theo} of 8 GB/s for Gen3 x8, the practically achievable throughput in this scenario can never exceed tp_{prac}:

$$\begin{aligned} tp_{\text{prac}} &= tp_{\text{theo}} * \text{encoding} * \frac{\text{MPS}}{\text{STP} + \text{Header} + \text{MPS} + \text{LCRC}} \\ &= 8\,\text{GB/s} * \frac{128}{130} * \frac{256}{4 + 12 + 256 + 4} \\ &= 7.306\,\text{GB/s} \end{aligned}$$

However, even that will be reduced in real-world usage due to difficult-to-model effects such as PCIe flow-control, ACK/NACK, and inter-layer latencies. In addition, software has to perform a context switch to kernel space for the driver system calls, which takes 100...300 ns, and an additional 2...10 us are required for acknowledging the interrupt raised by hardware. The MPS value used of 256 B is the maximum supported by the hardwired PCIe End Point and a typical upper bound of many server-class host systems.

8.2.1 Single CDMA Engine

In the initial scenario, we select a minimal ffLink configuration providing just a single CDMA engine. We measure the achievable throughput, shown in Figure 6, for different transfer sizes, using each of the three different transfer mechanisms discussed in Section 6.2.

The measurements differentiate between reading from device memory (which writes to host memory over the bus) and writing to device memory (which reads from host memory over the bus). The throughput will be determined by the operations crossing the bus. Since PCIe writes are non-posted requests (no response required), they will always be faster than bus reads (posted requests). Thus, in our scenario of the device performing the transfers as DMA bus master, we expect reads from device memory to be faster than writes to device memory.

For small transfer sizes, the achieved throughput is similar for all mechanisms, as copies to the single bounce buffer are still fast, and the setup overhead of each transfer dominates the execution time (even for the zero-copy approach). For increasing transfer sizes, however, the larger amounts of data to copy lead to the bounce buffer approach falling behind zero-copy. This growing overhead for copying actually leads to bounce buffer performance deteriorating for sizes of 1...4 MB, leveling out after the maximum single-transfer size of 4 MB is reached.

Compared to bounce buffering, double buffering becomes efficient for transfer sizes of 256 KB and above. At that point, the time required for copying to the single bounce buffer reaches the time required for performing a *second*

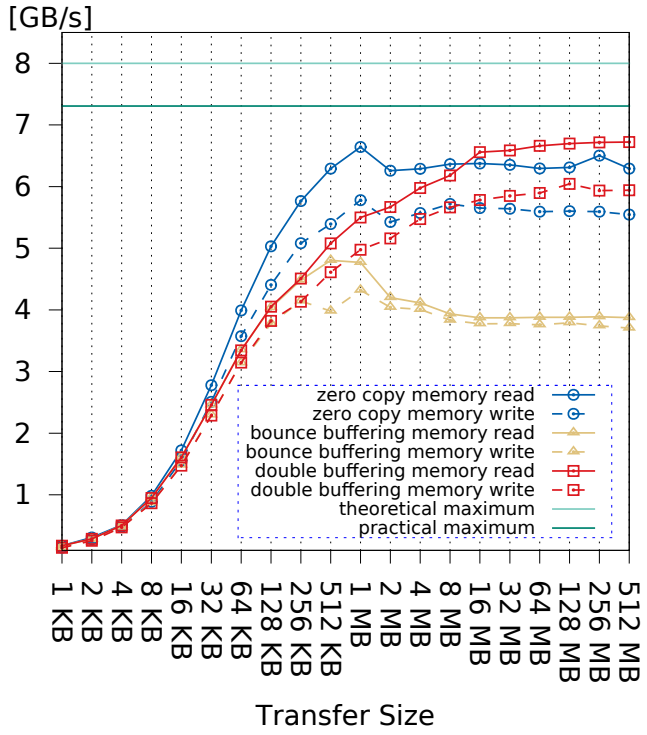

Figure 6: Throughput using single DMA engine

CDMA transfer. Since the behavior of zero-copy transfers indicate a sweet-spot for the throughput at 1 MB transfer sizes, double buffering is also internally configured to transfer larger amounts of data in 1 MB chunks. However, until the total size of a transfer exceeds 8 MB, the second buffer is not reused sufficiently often to completely hide the user→buffer copying time behind the actual DMA transfer time.

The 1 MB chunk size is indeed a sweet spot for double-buffering, as larger chunk sizes can be shown to no longer hide the growing copying times behind the DMA transfers, and smaller chunks require more time to set-up each transfer relative to the actual transfer times (reducing throughput).

As foreshadowed in Section 6.1, this behavior for both zero-copy as well as double buffering is the reason, why complex measures to increase the maximum DMA transfer sizes beyond their default Linux-imposed limit of 4 MB are not necessary in practice.

For the single CDMA engine case, zero-copy achieves the best throughput (up to 6.29 GB/s) for transfer sizes below 8 MB, while double buffering performs best for larger transfers (6.72 GB/s). The conventional approach of just using a single bounce buffer is not efficient, as it tops out 3.87 GB/s.

8.2.2 Multiple CDMA Engines

The same experiments were repeated for the three transfer mechanisms, but now using 1, 2, or 4 CDMA engines. For space reasons, Figure 7 graphs these results only for the zero-copy approach, the other approaches will be discussed below in text.

The key idea remains the use of multiple engines to hide the latencies of reacting to interrupts, or programming an engine for the next data transfer. For smaller transfer sizes,

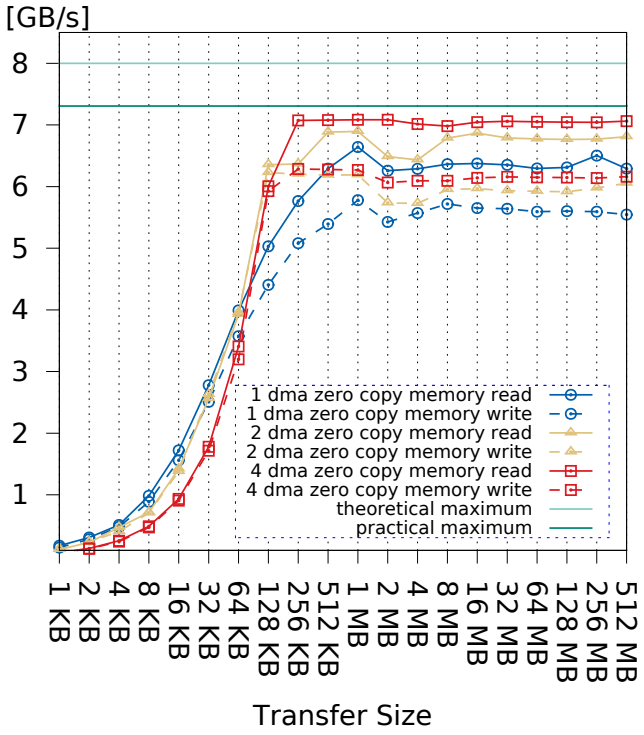

Figure 7: Throughput using multiple DMA engines

the effort of programming an additional engine is not efficient with regard to the very short actual transfer times. But at transfer sizes above 64 KB, the use of two engines becomes worthwhile. Finally, at 256 KB, the addition of two more engines improves throughput to a top performance of 7.06 GB/s for zero-copy.

In a similar fashion, the addition of more engines also improves the other transfer mechanisms, leading to a best-case performance of 6.15 GB/s for the conventional bounce buffer approach, and 7.06 GB/s for double buffering. Bounce buffering and zero-copy profit most from the additional engines, while double buffering realizes only a modest improvement.

9. CONCLUSION AND FUTURE WORK

ffLink achieves its goals of being lightweight (at most 11% device utilization), yet offering high data transfer performance of up to 7.06 GB/s. It thus exceeds the throughput of even commercial offerings. ffLink is highly configurable and allows to trade-off chip area and Linux DMA buffer memory with transfer throughput, as required by the application.

The entire system is composed using the Xilinx Vivado design flow and easily integrates custom logic (e.g., the actual accelerator block(s)) into memory access and host signaling mechanisms. By relying on standard functions from the Xilinx IP catalog (e.g., the CDMA block), ffLink can profit from future vendor updates without requiring user design effort.

An interesting future refinement would be the use of CDMA engines capable of 64b, which would eliminate the need for the SWIOTLB approach on processors without an IOMMU entirely.

The source code of ffLink *is available from the Download section of* `http://www.esa.cs.tu-darmstadt.de`.

Acknowledgment

This work was performed in the context of "REPARA – Reengineering and Enabling Performance and poweR of Applications" [3], a *Seventh Framework Programme* project of the European Union.

10. REFERENCES

[1] Demo Bundle Virtex7 Gen3. `http://xillybus.com/pcie-download`, 2014. Accessed: 02/15.

[2] NVIDIA NVLink High-Speed Interconnect: Application Performance. Whitepaper, 2014. Accessed: 02/15.

[3] REPARA - Reengineering and Enabling Performance and poweR of Applications. `http://www.repara-project.eu`, 2014. Accessed: 01/15.

[4] PCI Express Solution. `http://nwlogic.com/products/pci-express-solution/`, 2015. Accessed: 03/2015.

[5] R. Bittner. Speedy bus mastering pci express. In *Field Programmable Logic and Applications, Intl. Conf. on*, pages 523–526, Aug 2012.

[6] J. Gong, T. Wang, J. Chen, and et al. An efficient and flexible host-fpga pcie communication library. In *Field Programmable Logic and Applications, Intl. Conf. on*, pages 1–6, Sept 2014.

[7] M. Jacobsen and R. Kastner. Riffa 2.0: A reusable integration framework for fpga accelerators. In *Field Programmable Logic and Applications, Intl. Conf. on*, pages 1–8, Sept 2013.

[8] H. Kavianipour, S. Muschter, and C. Bohm. High performance fpga-based dma interface for pcie. *Nuclear Science, IEEE Trans. on*, 61(2):745–749, April 2014.

[9] A. Tumeo, M. Monchiero, G. Palermo, and et al. Lightweight dma management mechanisms for multiprocessors on fpga. In *Application-Specific Systems, Architectures and Processors, ASAP Intl. Conf. on*, pages 275–280, July 2008.

[10] K. Vipin, S. Shreejith, D. Gunasekera, and et al. System-level fpga device driver with high-level synthesis support. In *Field-Programmable Technology, Intl. Conf. on*, pages 128–135, Dec 2013.

[11] Q. Wu, J. Xu, X. Li, and et al. The research and implementation of interfacing based on pci express. In *Electronic Measurement Instruments, Intl. Conf. on*, pages 3–116–3–121, Aug 2009.

[12] Xilinx Inc. *LogiCORE IP AXI Central Direct Memory Access*, v4.1 edition, December 2013.

[13] Xilinx Inc. *AXI Bridge for PCI Express Gen3 Subsystem*, v1.0 edition, November 2014.

[14] Xilinx Inc. *Virtex-7 FPGA XT Connectivity Targeted Reference Design for the VC709 Board*, v3.0 edition, December 2014.

[15] C. Zinner and W. Kubinger. Ros-dma: A dma double buffering method for embedded image processing with resource optimized slicing. In *Real-Time and Embedded Technology and Applications Symposium, Proc. on*, pages 361–372, April 2006.