Design and System Level Evaluation of a High Performance Memory System for reconfigurable SoC Platforms

Holger Lange*,[1], Andreas Koch*,[1]

*Tech. Univ. Darmstadt
Embedded Systems and Applications Group (ESA)
Darmstadt, Germany

ABSTRACT

We present a high performance memory attachment for custom hardware accelerators on reconfigurable SoC platforms. By selectively replacing the conventional on-chip bus wrappers by point-to-point connections, the full bandwidth of the DDR DRAM-based main memory is made available for the accelerator data path. At the same time, the system continues to run software at an almost undiminished execution speed, even when using a full-scale Linux as virtual-memory multi-tasking operating system. Despite the performance gains, the approach also reduces chip area by sharing the already existing bus interface to the DDR-DRAM controller with the hardware accelerator. System-level experiments measuring both software and accelerator performance on a representative rSOC platform show a speedup of two to four over the vendor-provided means of attaching accelerator cores, while still allowing the use of standard design flows. The results are expected to be portable to other platforms, as similar on-chip buses, protocols, and interface wrappers are employed in a great number of variations.

1 Introduction

Building Systems-on-Chip (SoCs) is complicated by their inherently heterogeneous nature. This necessitates efficient communication of their components both with external peripherals as well with each other. In this context, the memory subsystem plays a crucial role, as a growing number of SoC components require master-mode access (self-initiate transfers) to memories, and memory transfers account for the vast majority of overall bus traffic. Low-latency, high-bandwidth memory interfacing is thus highly desirable, especially in application domains where excess clock cycles and associated power consumption, incurred due to waiting for the completion of memory accesses, are not acceptable. For high-performance embedded systems, the CPU(s) alone already demands a significant share of the memory bandwidth to keep cache(s) filled, but the memory bottleneck becomes even tighter when custom hardware accelerators (abbreviated here as *HW*) are integrated into the system. These blocks, often realized by IP cores, frequently rely on a fast memory path to realize their efficiency advantages, both in speed and power, over software (*SW*) running on

[1]E-mail: {lange,koch}@esa.cs.tu-darmstadt.de

conventional CPUs.

For interconnecting SoC components, a variety of standards have been proposed [1][2]. In context of this work, we examined the IBM CoreConnect [3] approach more closely, which is also used in the widespread reconfigurable SoCs (*rSoC*) based on Xilinx Virtex II-pro (V2P) platform FPGAs.

These popular devices allow the creation of SoCs in a reconfigurable fashion, by embedding efficiently hardwired core components such as processors into the reconfigurable ressources. However, as we will demonstrate, they implement an even *less* performant subset of CoreConnect than the specification itself would allow. The supposedly high-performance means of attaching accelerators directly to the processor local bus (PLB) for memory access, as recommended by Xilinx development tools (Embedded Development Kit, EDK) [5], leads to a cumbersome high-latency, low-bandwidth interface instead.

Interestingly, our observations are not limited to the domain of reconfigurable SoCs, but also apply to hardwired ASIC SoCs such as the IBM Blue Gene/L compute chip [6]. Here, even the more advanced version 4 of the PLB had to be operated *beyond* the specification in order to provide a sufficiently fast link between the L2 and L1 caches, the latter feeding the PowerPC 440 used as processor on the chip at 700 MHz. In addition to the "overclocking" of the interface, it was also used as a dedicated point-to-point connection, instead of the bus fashion it was originally intended for (it would not have achieved the required performance then).

2 Base Platform Xilinx ML310

Since the simulation of an entire system comprising one or more processors, custom accelerators, memories, and I/O peripherals is both difficult and often inaccurate, we employ an actual HW platform for our experiments.

The Xilinx ML310 [4] is an embedded system development platform which resembles a standard PC main board (see Fig. 1). It features a variety of peripherals (USB, NIC, IDE, UART, AC97 audio, etc.) attached via a Southbridge ASIC to a PCI bus, which provides a realistic environment for later system-level evaluation. In contrast to a standard PC, the CPU and the usual Northbridge ASIC have been replaced by a V2P FPGA [7], which comprises two PowerPC 405 processor cores that may be clocked at up to 300MHz. They are embedded in an array of reconfigurable logic (RA). Thus, the "heart" of the compute system (CPUs, accelerators, buses, memory interface) is now reconfigurable and thus suitable for experiments in architectures. With sufficient care, this rSoC can implement even complex designs with a clock frequency of 100 MHz (a third of the embedded CPU cores' clock frequency).

The ML310 is shipped with a V2P PCI reference design (shown on a gray background in Fig. 1). This design consists of several on-chip peripherals, which are attached to a single PowerPC core by CoreConnect buses. These peripherals comprise memory controllers (DDR DRAM and Block-RAM), I/O (PCI-Bridge, UART, the System ACE compact flash based-boot controller, GPIO, etc.), an interrupt controller, and bridges between the different CoreConnect buses.

On the SW side of the system, we run the platform under Linux. While the use of such a full-scale multi-tasking, virtual memory might seem overkill for the embedded area, the market share of Linux variants in that field is roughly 20% ([8], VxWorks 9%, Windows 12%). Furthermore, Linux stresses the on-chip communication network more than a light-weight

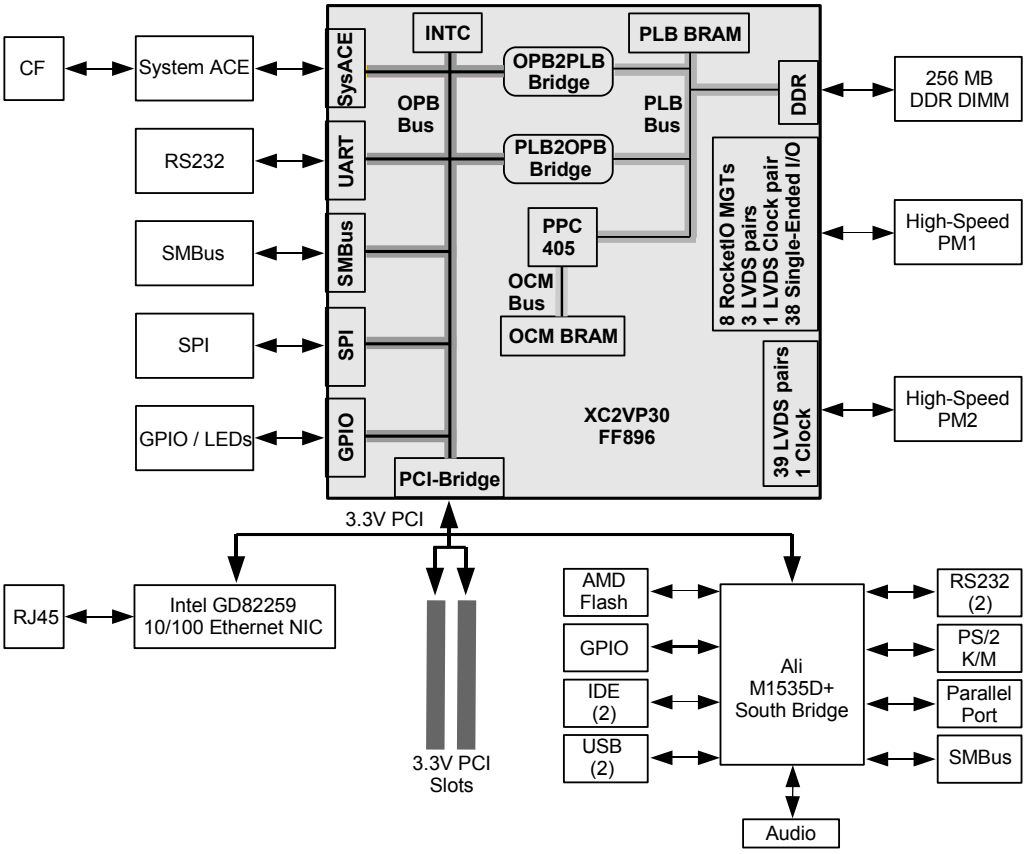

Figure 1: ML310 system (from Xilinx manuals)

RTOS, and is thus better suited for load-testing our architecture.

3 Vendor Flow for rSoC Composition

Regardless of the target technology, actually composing an SoC is a non-trivial endeavor. In addition to the sheer number of components (e.g., as demonstrated by the ML310), different components also have different interface requirements (e.g., an UART vs. a processor core) or may not be available using one of the supported standard interfaces at all. For our platform, standard interfaces would be either the PLB interface already mentioned above, or the simpler On-chip Peripheral Bus (OPB), which will be described below. Non-standard interfaces are common to HW accelerator blocks that often have applications-specific attachments, which are then connected to standard buses by means of so-called *wrappers*. These convert between the internal and external interfaces and protocols. In some cases, the different protocols are fundamentally incompatible and can only be connected with additional latencies or even loss of features (such as degraded burst performance).

The Xilinx EDK [5] SoC composition tool flow supports two standard means for integrating custom accelerators into the reference design. The simpler way is the attachment via OPB [3], shown in Figure 2. The idea here is to isolate slower peripherals from the faster processor bus by putting them on a dedicated bus with less complex protocols. OPB attachments thus implement a relatively simple bus protocol, having 32 bits transfer width at 100

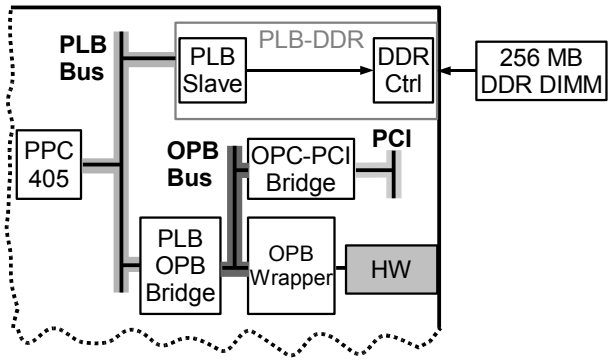

Figure 2: HW integration via OPB

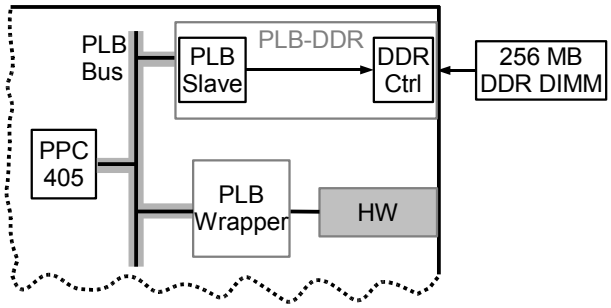

Figure 3: HW integration via PLB

MHz clock. The most important OPB operation modes are:

- Single beat transfers (one data item per transaction)
- Burst transfers of up to 16 data words (limited by processor bus)
- Master/slave (self/peer initiated) transfers

However, the simplicity comes at the price of higher access latencies compared to PLB attachment. These are due both to the OPB wrapper of the block as well as the PLB-OPB bridge which must be traversed when communicating with high-speed blocks on the PLB (such as the processor and the main memory).

The PLB attachment is the second proposed way of integrating HW accelerators, shown in Figure 3. Its protocol has more features aiming for high-performance operation, but is also more complex than the OPB one. Hence, bus wrappers are nearly always needed when connecting to the PLB. The most important PLB operation modes are:

- Single beat transfers (one data item per transaction)
- Burst transfers up to 16 data words
- Cacheline transfers (one cacheline in 4 data beats, cache-missed word first)
- Master/slave (self/peer initiated) transfers
- Atomic transactions (bus locking)
- Split transactions (separate masters/slaves performing simultaneous reads/writes)
- Central arbiter, but master is responsible for timely bus release

Additionally, the PLB interface also operates on 64 bits of data at 100 MHz (twice the bandwidth of OPB), accompanied by lower access latencies with just two bus wrappers left between HW and DDR-DRAM controller for the main memory. Note that the controller itself also requires a wrapper. However, since the memory never initiates transactions by itself, a slave-mode wrapper suffices for this purpose.

4 Practical Limitations

As discussed in the previous section, attaching a HW accelerator to the PLB offers performance increased over an OPB attachment. However, there are still practical limitations: The

	IBM PLB Spec v3	Xilinx V2P PLB
Clock	133 MHz	100 MHz
Single cycle transactions	yes	no (wrappers)
Burst length	unlimited	16 words

Table 1: PLB specification vs. implementation

Wrapper for	Min. Size [Slices] (slave only)	Max. Size [Slices] (full master-slave)
OPB	27	544
PLB	180	2593

Table 2: Area overhead for bus wrappers in vendor flow

original CoreConnect specification [3] allows for unlimited PLB burst lengths, with transaction latencies being as short as a single cycle (if the addressed slave allows it). Unfortunately, as shown in Table 1, the actual implementation of the specification on the V2P-series of devices does not support all of the specified capabilities.

In addition to being clocked at only 100 MHz, PLB is further hindered by its relatively complex protocol and an arbitration-based access scheme, both leading to long initial latencies. Furthermore, in the Xilinx V2P implementation, the maximum burst length is limited to just 16 words. The bus wrappers employed in the vendor tool flow impose additional latency and can also require considerable chip area (see Table 2). Even the performance-critical controller for the DDR-DRAM main memory is also connected to the PLB by means of a wrapper (see Figure 3). This combination of restrictions renders the memory subsystem insufficient for 64 bit, DDR-200 operation (1600 MB/s theoretical peak performance, the maximum supported by the actual DDR DRAM memory chips used on the ML310 mainboard).

5 *FastLane* Memory System

However, since we are experimenting with a *reconfigurable* SoC, we can choose an alternate architecture. To this end, we designed and implemented a new approach to interface the processor, custom HW accelerators and the main memory.

The main concept behind the *FastLane* high performance memory interface is the *direct* connection of the memory-intensive accelerator cores to the central memory controller without an intervening PLB. By also using a specialized, light-weight protocol, we can avoid the arbitration as well as the protocol overhead associated with PLB. This leads to a greatly reduced latency, with no wrapper left between accelerator core and RAM controller, as opposed to two wrappers in the Xilinx reference design. We can now also make the full bandwidth of the RAM controller available to the accelerator, eventually enabling true 64 bit double data-rate operation. Figure 4 shows the new memory subsystem layout.

For even further savings, the accelerator(s) now also share the PLB slave attachment of the DDR controller wrapper. Thus, no additional chip area is wasted on wrappers (which have now become redundant). The master mode side of the accelerator is connected via FastLane directly to the interface of the DDR controller, but can accept data transfers from

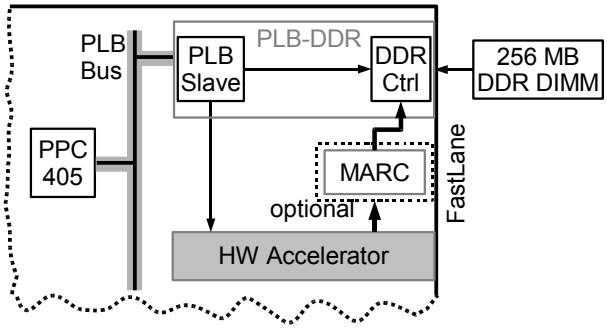

Figure 4: FastLane: Attaching HW accelerator directly to DDR controller

the processor (e.g., commands and parameters) via the shared PLB slave. Both interfaces internally use a simple double handshake protocol, streamlined for low latency and fast burst transfers.

Many algorithms implemented on HW accelerators can profit from higher-level abstractions in their memory systems, such as FIFOs/prefetching for streaming data, and caches for irregular accesses. To this end, the Memory Architecture for Reconfigurable Computers (MARC, shown in Figure 5) can be inserted seamlessly in the FastLane, using a compatible protocol. MARC offers a multi-port memory environment with emphasis on caching and streaming services for (ir)regular access patterns. MARC, which is described in greater detail in [9], consists of 3 parts:

- The *core* encapsulates the functionality for caching and streaming services, the cache tag CAM (Content Addressable Memory), cacheline RAM and stream FIFOs. The core also arbitrates the *back ends* and *front ends*, aiming to keep all of them working concurrently but resolving conflicts when accessing the same resource.
- The *front ends* provide standardized, simple interface ports for both streaming and caching using a simple double-handshake protocol.
- The *back ends* adapt the core to several memory and bus technologies. New backends can be easily added as required.

While the FastLane approach aims to provide optimal conditions for the compute-intense HW accelerators, it must also consider that the rest of the system, specifically the processor(s), also require access to the main memory and may be intolerant to longer delays in answers to their requests. For example, interrupts, timers and the process scheduler cause memory traffic even on an idle system, and a too-slow response leads to system instabilities. Bus master devices (capable of initiating transfers on the bus) may experience buffer over- or underruns if the transfer is not completed in time due to bus contention caused by a HW accelerator.

This implies that the CPU and other bus master devices must always have *priority* over the accelerator block (which can be explicitly designed to tolerate access delays). The required arbitration logic is completely hidden from the accelerator within the FastLane interface. The CPU (and other bus master devices) may interrupt master accesses of the accelerator at any time, while the accelerator cannot interrupt the CPU, and has to wait for the completion of a CPU-initiated transfer.

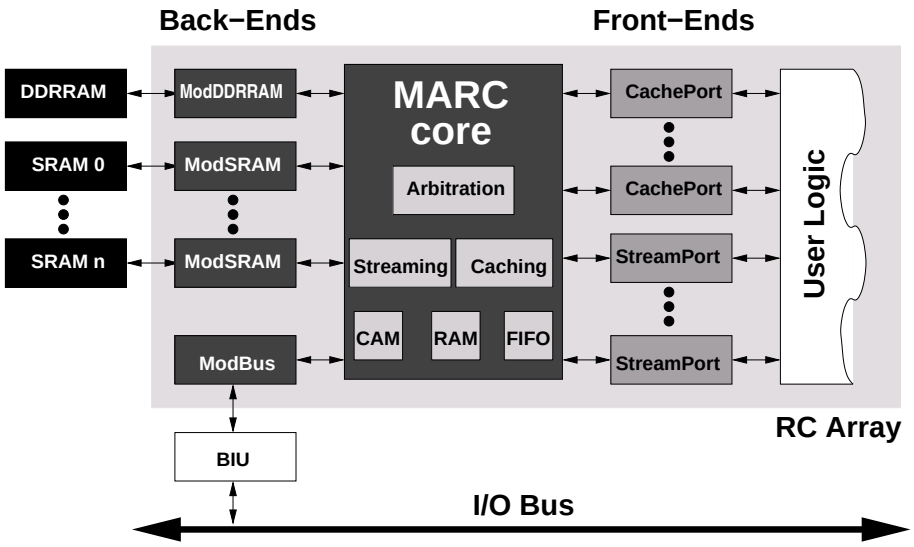

Figure 5: MARC overview

6 Operating System Integration

When considering accelerators at the system level, it is obvious that these HW cores must also be made accessible to application SW in an efficient manner. Given their master-mode access to main memory, this is non-trivial in an OS environment supporting virtual memory. The memory management unit (MMU) translates the virtual *userspace* addresses as seen by SW applications into physical bus addresses, which are sent out from the processor via the PLB. Address translations and the resolution of page faults are transparent for SW. Since the accelerators do not have access to the processor MMU with its page address remapping tables, this implies that hard- and software communication in a virtual memory environment must use *both* userspace and physical addresses. Furthermore, since the HW is neither aware of virtual addresses, nor can it handle page faults, the memory pages accessed by the accelerator *must* be present in RAM before starting the accelerator.

The solution to this requirement is a so-called *Direct Memory Access buffer* (DMA buffer). In the Linux virtual memory environment, a DMA buffer is guaranteed to consist of contiguous physical memory pages that are locked down and always present in physical RAM, they can never be swapped out to disk. As described previously, there are now *two* addresses pointing to the buffer, the first being the physical bus address as seen by the accelerator, the second being the virtual userspace address representing the same memory area for application SW. In the example given in Figure 6, a SW program has allocated a DMA buffer and passes its physical address to the accelerator. The SW can access this buffer via userspace address 0x01004000, which is translated to physical address 0x12345678 by the MMU. The accelerator directly uses this physical address to access the same DMA buffer.

The algorithms running on the accelerator often require a set of parameters for their operation, which are transferred from the SW application by performing writes to the memory-mapped accelerator registers. These are actually handled by the PLB slave shared with the memory controller and forwarded to the accelerator HW. From the SW perspective, the memory mapped registers are simply accessed via a pointer to a suitable data structure. Bulk data (e.g., image pixmaps etc.) is also prepared by the SW within the previously al-

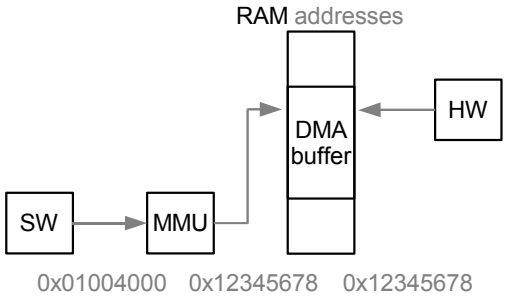

Figure 6: HW and SW addressing of DMA memory

located DMA buffer (e.g., read from a file), which can be manipulated by SW as any other dynamically allocated memory block. Then, offsets from the start of the DMA buffer where various data structures are located (both for input and output) are simply passed as parameters to the accelerator. After starting the accelerator by writing to its command register, the accelerator fetches the bulk data directly from memory, preferably in efficient burst reads, without involvement of SW running on the processor. In a similar fashion, it also deposits the output data in the DMA buffer, and then indicates completion of the operation via an interrupt to the processor. Now SW can take over again.

On the OS side, this functionality is completely encapsulated in a character device driver that performs the appropriate memory mappings for the slave-mode registers and the DMA buffer.

7 Experimental Results

To demonstrate the effectiveness of our approach, we exercised several system load scenarios. The basic setup is identical in all cases: We mimic the actions of an actual HW accelerator by a hardware block that simply repeatedly copies a 2 MB buffer from one memory location to another as quickly as possible, totaling to 4 MB of reads and writes per turn, and measure the transfer rate in MB/s. In addition to the accelerator, we run different software programs, chosen for their specific load characteristics on the processor. We then measure the HW execution time (the time it takes to copy memory data at full speed) and the SW execution time (the time it takes for a given program to execute on the processor) for both the original vendor-provided as well as our FastLane memory interface. The extreme cases (HW accelerator and processor idle) are also considered.

The suite of software programs was chosen to represent an everyday mix of typical applications that also perform I/O and calculations in main memory (instead of just running entirely within the CPU caches). The *scp* program from the OpenSSH suite [10] was instructed to copy a local file, sized 4 MB, via network to a remote system. The same is done without encryption by *netcat* [11]. The GNU *gcc* [12] C compiler was evaluated while compiling the *netcat* sources.

To also cover the embedded system domain where SoC platforms similar to V2P are often employed, the ETSI GSM enhanced full rate speech codec [13] and an image processing pipeline as often found in printers were included ([14], JPEG RGB to CMYK conversion as part of the HP Labs Vliw EXample development kit), both representing typical embedded

Application	V2P ref design		FastLane	
	Exec. Time [ms]	Mem. Rate [MB/s]	Exec. Time [ms]	Mem. Rate [MB/s]
idle system	18.81	213	5.67	705
scp	55.11	73	12.82	312
netcat	53.07	75	19.27	208
gcc	32.14	124	17.42	230
GSM	19.05	210	6.35	630
imgpipe	44.67	90	19.33	207

Table 3: HW accelerator run times and available bandwidth using original and FastLane memory subsystem implementations

Application	HW inactive [ms]	V2P ref design [ms]	FastLane [ms]
scp	4831	61052	5828
netcat	3130	55938	3901
gcc	40686	166655	52908
GSM	25981	40045	27767
imgpipe	3545	5109	4018

Table 4: SW run times on idle system and using accelerator attached by original and FastLane memory subsystem implementations

applications. The various programs can be characterized as follows:

- **scp** provides a mix of CPU- and I/O load
- **netcat** exercises network I/O exclusively
- **gcc** interleaves short I/O- and long calculation phases
- **GSM** provides codec stream data processing
- **imgpipe** implements multi-stage image processing

The first set of measurements shown in Table 3 considers the memory throughput of the HW accelerator under different CPU load scenarios. Here, we show the time for a single 2 MB block copy (four mega-transfers), as well as the resulting memory throughput, when using the original vendor-provided PLB interface as well as our FastLane for connecting the HW accelerator. It is obvious that FastLane significantly increases the throughput in all load scenarios, in some cases by a factor of up to 4.3.

The set of measurements shown in Table 4 quantifies the influence of the different memory attachments on the execution time of software running on the processor that also access main memory in different load patterns. The results show that, despite its high throughput to the HW accelerator, FastLane does not significantly impair the processor: SW execution times are almost unaffected by the HW memory transfer, owing to the absolute priority of the CPU (cf. Section 5) over the HW accelerator. In contrast, the original vendor-provided reference design exhibits a steep SW performance decline, increasing execution times by a factor of up to 14x over that of SW running with the FastLane-attached accelerator. FastLane thus enables the accelerator to access memory bandwidth that appears to be completely unused by the original memory interface.

One might assume that the FastLane approach of giving the CPU override priority for bus access will cancel out the theoretical performance gains of FastLane over the original

PLB-based accelerator attachment. However, we measured that even under these conditions, FastLane is able to provide the accelerator with roughly half of the theoretically available memory bandwidth (which is 800 MB/s in the single-data rate mode used here): Practically achievable are 32b data words at a rate of 352 MB/s and 64b words at 705 MB/s, yielding a bus efficiency of 88%. Going to double data rate mode (planned as a refinement) would double these rates again. At the same time, the multi-tasking OS and the SW application continue to run at almost full speed.

In scenarios where fast SW interrupt response is not required, for example, and it is possible to freeze the processor entirely (e.g. by stopping the clock signal), FastLane makes the full memory bandwidth available to the accelerator. This is not achievable using the original PLB attachment, which even with a frozen processor is only able to exploit just 25% of the theoretically available read bandwidth and 33% when writing.

8 Conclusion and Future Work

We presented a high performance memory attachment for custom HW accelerators. Our approach can increase the usable memory throughput by more than 4x over the original vendor-provided PLB attachment (included in the Xilinx EDK [5] design suite). Additionally, it required less chip area and left performance of SW applications running on the on-chip processors almost unaffected.

From a practical perspective, FastLane integrates into the standard EDK design flow and is as easy to use as the original attachment. Although the results are not directly transferable to platforms other than Virtex 2 Pro system FPGA, it is clear that reduced bus and wrapper overhead will always result in smaller logic and lower latencies. Hence, other platforms may also benefit from this approach.

Our current and future work focuses on improving the OS support (transparent addresses between HW and SW) as well as the supporting full 64b double-data rate operation in FastLane.

References

[1] AMBA home page, **http://www.amba.com**, 2006
[2] Open Core Protocol International Partnership, **http://www.ocpip.org**, 2006
[3] IBM, "The CoreConnect Bus Architecture", *White Paper*, 1999
[4] Xilinx, "ML310 User Guide" *(UG068)*, 2005
[5] Xilinx, "Embedded System Tools Reference Manual" *(UG111)*, 2006
[6] M. Ohmacht et al, "Blue Gene/L compute chip: Memory and Ethernet subsystem", *IBM Journal of Research and Development*, Vol. 49, No. 2/3, pp. 255-264, 2005
[7] Xilinx, "Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet" *(DS083)*, 2005
[8] J. Turley, "Operating systems on the rise", **http://www.embedded.com**, 2006
[9] H. Lange, A. Koch, "Memory Access Schemes for Configurable Processors", *Proc. Workshop on Field-Programmable Logic and Applications (FPL)*, Villach, 2000
[10] **http://www.openssh.com**, 2006
[11] **http://netcat.sourceforge.net**, 2006
[12] **http://gcc.gnu.org**, 2006
[13] ETSI EN 300 724 V8.0.1 (2000-11), **http://www.etsi.org**, 2006
[14] J. Fisher, P. Faraboschi, C. Young, "Embedded Computing: A VLIW Approach to Architecture, Compiler and Tools", chapter 11.1, Elsevier, 2005