First International Workshop on
HyperTransport™ Research and Applications

Proceedings

of the

1st International Workshop on HyperTransport Research and Applications

WHTRA 2009

Editors
Holger Fröning
Mondrian Nüssle
Pedro Javier García García

ISBN: 978-3-00-027249-3

February 12th, 2009, Mannheim, Germany

University of Heidelberg, Computer Architecture Group

Proceedings of the
First International Workshop on HyperTransport Research and Applications (WHTRA2009)
Feb. 12th, 2009, Mannheim, Germany

A general purpose HyperTransport-based Application Accelerator Framework

David Kramer Thorsten Vogel Rainer Buchty Fabian Nowak
and Wolfgang Karl
Institute of Computer Science & Engineering
Universität Karlsruhe (TH)
Zirkel 2
76131 Karlsruhe, Germany
{kramer, thorsten.vogel, buchty, nowak, karl}@ira.uka.de

Abstract

HyperTransport provides a flexible, low latency and high bandwidth interconnection between processors and also between processors and peripheral components. Therefore, the interconnection is no longer a performance bottleneck when integrating application specific accelerators in modern computing systems. Current FPGAs providing huge computational power and permit the acceleration of compute-intensive kernels. We therefore present a general purpose architecture based on HyperTransport and modern FPGAs to accelerate time-consuming computations. Further, we present a prototypical implementation of our architecture. Here we used an AMD Opteron-based system with the HTX Board [6] to demonstrate that common applications can benefit from available hardware accelerators. A cryptographic example showed that the encryption of files, larger then 50 kByte, can be successfully accelerated.

1 Introduction

We still see Moore's Law [10] being valid with the transistor count on a single chip doubling every 18 to 24 months. This further increase in technology density has two significant implications: for processor architectures, the further increase in integration does, due to technological constraints, not lead to increased individual processor speed, but rather the creation of multicore processor architectures. For reconfigurable logic, in term, it results in comparably large units being able to hold complex systems on chips or a multitude of dedicated hardware accelerators.

In the past, the use of such hardware accelerators was mainly hampered by the absence of appropriate interconnection technology. Since the demise of dedicated coprocessor interfaces, which enabled a fine-granular integration of hardware accelerators into the system, only peripheral buses remained as the only way of connection, imposing huge limitations on transfer speed and, therefore, data exchange between host system and accelerator.

HyperTransport is an example of current interconnection technology, serving for either CPU/CPU communication or the connection to peripheral subsystems. It therefore not only provides necessary speed and bandwidth numbers to not become a significant communication bottleneck, but furthermore enables tight integration of arbitrary computation units into the host system.

This interconnection technology combined with current FPGA technology enables the development and use of dynamically configurable hardware accelerators for vertical migration of algorithms, i.e. offloading dedicated or otherwise time-consuming computations such as e.g. numerical simulations, cryptographic algorithms, or processing of streaming media to specialized accelerator units.

In this paper we describe the design and use of an FPGA-based general-purpose hardware accelerator unit for HyperTransport-equipped systems. This accelerator unit comprises up to 6 individual accelerator cores and according circuitry providing interfacing with the HyperTransport bus and higher-level functions such as DMA data transfer and Monitoring.

This accelerator unit is part of an integrated platform consisting of a controlling host system, individual hardware accelerators, and a runtime system [3], enabling dynamic resolution of computation routines to be executed either in software on the host processor or offloaded to one or more accelerators. The platform also features a currently developed C/C++ language extension to generate control information enabling automation of the dynamic function resolution and general automatic optimization in the scope of Self-X and adaptive systems.

The remainder of this paper is therefore structured as follows: we will first present related work in Section 2,

shortly introducing other architectures and approaches with their advantages and drawbacks. In Section 3, we give an overview of our proposed accelerator architecture and how it integrates into HyperTransport systems. Section 4 present the first implementation of our architecture using the HTX Board from the University of Heidelberg. Section 5 contains an application case study demonstrating the usability and general applicability of our architecture. An outlook of ongoing and future is given in Section 6 and the paper is concluded with Section 7.

2 Related Work

In the past, different approaches for integrating hardware acceleration into existing systems were introduced, ranging from simple accelerator cards to dedicated co-processor solutions and completely dynamic processor architectures. Especially the latter employ FPGA technology to enable on-demand reconfiguration, therefore leading to increased use of the silicon area. The existing approaches can be roughly divided into two categories by granularity, i.e. fine-grained instruction set extension and coarse-grained extension as co-processors.

An early research project for using reconfigurable logic for instruction set extension is PRISM (Processor Reconfiguration Through Instruction-Set Metamorphosis) [2]. PRISM consists of a general purpose processor and an FPGA, which is connected through a dedicated bus system. PRISM focuses on transparent hardware generation and acceleration of standard C code for single applications running on a general purpose processor. A configuration compiler splits the application into a software image and a hardware image. The software image is a regular binary which runs on the general purpose processor. This binary contains code that coordinates the execution of the generated hardware accelerators. The hardware accelerators are suggested by the compiler, generated by external synthesis tools, and included in the hardware image. This image runs on the connected FPGA. A similar concept is used by Garp [7], which also relies on compiler-generated predefinition of a reconfigurable logic section; in contrast to PRISM, however, a dedicated array enabling easy on-demand reconfiguration was used. According instructions were added into the processor instruction set to enable dynamic loading and unloading of hardware configurations.

While in PRISM the instruction set is fixed for a specific application, the instruction set in the DISC (Dynamic Instruction Set Computer) [16] is completely dynamic. Here, instructions are implemented as modules which can be loaded onto an FPGA at runtime. Since the FPGA has a limited size it can hold only a restricted number of instructions. In DISC, unused instructions can be replaced at runtime with the help of Partial Dynamic Reconfiguration. If not enough FPGA resources are available, LRU strategy is used to select instruction modules which are removed. A DISC application consists of instruction modules and software which defines their execution order.

The MOLEN Polymorphic Processor [15] is another approach for a processor which is capable of custom computing. Like Garp, it uses the reconfigurable co-processor scheme. In contrast to Garp, MOLEN allows parallel execution of several independent hardware operations. Additionally it uses standard FPGAs for the reconfigurable co-processor. Therefore, high-level hardware description languages can be used to develop the custom configured units (CCUs).

Although the clock frequency of current FPGA technology is a magnitude slower than that of recent CPUs, several approaches target integration of FPGAs as co-processors in high-performance computing systems. Common to all approaches is that they do not accelerate single instruction, but rather coarse-grained parts of the application as fine-grained acceleration of individual instructions is not feasible. This is due to latencies occuring from configuration, data transfer, and triggering computation which limit effectively limit the granularity of FPGA-based acceleration. Being usually just peripherals, they are rather loosely coupled to the remaining system so that e.g. it is not possible to stall the processor pipeline when executing a special instruction.

An example for such an architecture is the Cray XD1 [9] computing platform featuring AMD Opteron processors for general purpose processing and Xilinx Virtex FPGAs for accelerating compute intensive kernels. HyperTransport is used as the interconnection technology between CPU and FPGA. The bit-streams for the FPGAs are created using a high-level hardware description language and the standard Xilinx development tools. An API is provided for accessing the application accelerators from within the application.

SGI offers the *SGI RASC RC100 Blade* [12] for accelerating HPC applications. The blade features two Xilinx Virtex 4 LX200 FPGAs and 80 MB of SRAM. It is directly connected to the system's shared memory via the proprietary NUMALink interconnection. Intel Itanium CPUs are used as general-purpose processing units. The provided software solution allows to run the reconfigurable computing elements in a multi-user and multi-process environment. The application accelerators can be developed using a high level language like Impulse-C.

Our approach, as outlined in the next section, follows the latter design principles, i.e. enhancing a host system by a dedicated accelerator board, merging in the parallel aspects of MOLEN by providing several individual accelerator units which may be used and configured individually. Likewise, dynamic reconfiguration of individual accelerator units is possible. Using HyperTransport interconnection technology enables a tight integration into current state-of-

the-art workstations.

3 Overview

This section describes the structure of our architecture. Besides the hardware components, our architecture also includes a software stack for easy use of application accelerators form within normal C code and additional monitoring and steering components. The monitors provide status information of the hardware to the steering components, which can use these information, for example, to guide the reconfiguration process.

3.1 Hardware Components

The overview of the hardware components is depicted in Figure 1. The hardware consists of seven main parts: the HT Core, the Command- and Status-Bus (CSB), the Data Bus, the DMA Unit, the Monitoring Infrastructure, the Reconfiguration Controller, and the Application Accelerators itself.

3.1.1 HT Core

The HT Core connects our architecture to a HyperTransport-bus. It provides the application accelerators including the DMA Unit an efficient way to access the system's main memory. All protocol handling regarding HyperTransport is handled within this component. The HT Core provides an interface to the HyperTransport link signals and an uniform, queue-based interface to the application accelerators and the CSB. Further, the I/O area of the HT Core is memory-mapped into the virtual memory of the host system to enable easy usage of the application accelerators from within application code.

3.1.2 Command- and Status-Bus (CSB)

Components which are connected to the CSB receive commands from and provide status information to the software. Parallel read and write requests are enabled by using two separate buses. The CSB has an interface to the HT Core and the reconfiguration controller and interfaces to the monitors and the application accelerators.

Part of the CSB are two so-called Request Coder. The Request Coders act as a bridge between HT and CSB and are used to convert internal messages to HT messages and vice versa; this is necessary as both buses may be of different width.

3.1.3 Data Bus

Like the CSB, the Data Bus is also divided into a write bus and a read bus. This separation allows handling data reads and writes independently. The data bus has an interface to the DMA Unit and interfaces to the application accelerators. An arbiter is used to grant access rights to the individual components.

3.1.4 DMA Unit

Our architecture features a Direct Memory Access-Unit to avoid Programmed Input/Output (PIO). Accounting for the independent read and write buses, read and write requests are handled by two distinct components, not only simplifying the design but also allowing concurrent read and write operations.

The DMA Unit hides HyperTransport-specific details. HyperTransport does only allow memory access aligned to a 64 byte boundary [1]. Unaligned accesses or accesses of more than 64 byte are split by the DMA Unit into multiple accesses adhering to the bus restrictions.

3.1.5 Monitoring Infrastructure

To support software-based control daemons, a *monitoring infrastructure* was introduced. This infrastructure consists of several, independent monitors, more precisely, one monitor for each application accelerator. The monitor itself consists of multiple 32-bit counters. It monitors the state of the accelerator. The counter corresponding to the current state is increased every clock cycle. The monitors provide an interface to the CSB and, upon request, deliver information to higher control instances.

3.1.6 Reconfiguration Controller

The *reconfiguration controller* can be used to reconfigure independent accelerator slots. For this purpose, it has an interface to both, on-chip bus systems and, if available, an interface to a reconfiguration port of the FPGA.

3.1.7 Application Accelerators

The *application* accelerators consists of two main parts, the *Accelerator Interface* (AI) and an *Accelerator Wrapper* (AW). The latter consists of a *Parameter and Result Bridge* (PRB) and the application accelerator itself.

The *AI* has two main objectives: it provides a uniform interface to CSB and Data Bus, and forwards commands, parameters, and data to the AW. The uniform interface ensures compatibility among different accelerators, as every accelerator uses the same interface. The uniform interface is achieved by parameter serialization, i.e. all parameters for an application accelerator are serialized on software side. As a result of this approach, only one parameter per clock cycle can be passed to the accelerator.

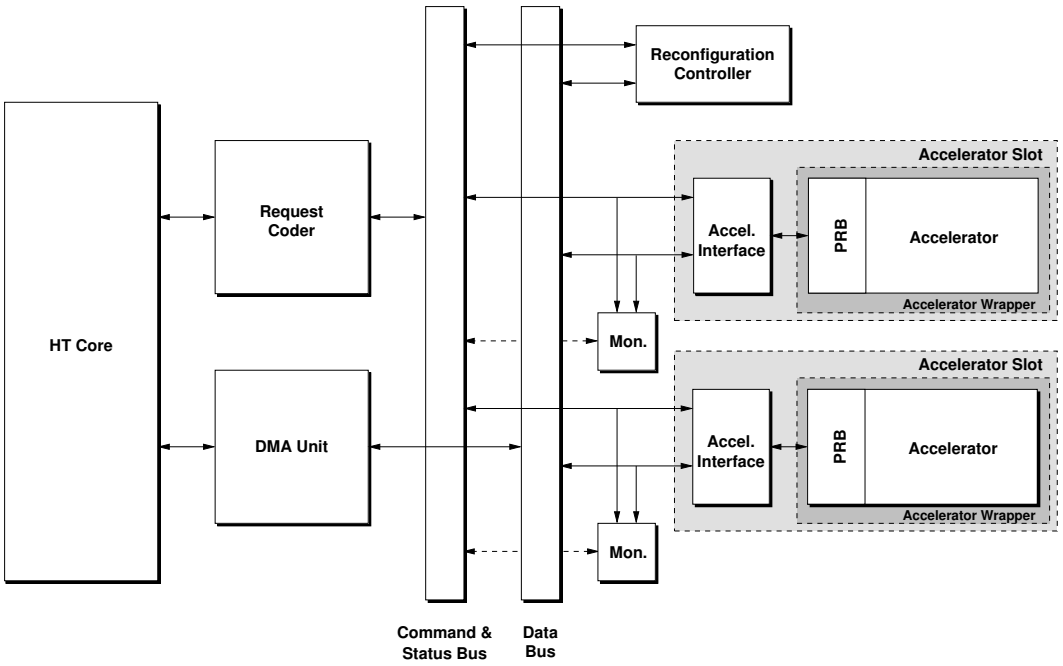

Figure 1. Hardware Composition

```
typedef struct acc_mgnt_struct
{
  controller_t    controller;
  read_access_t   read_access;
  accel_slot_t    slots[SLOT_COUNT];
  accel_monitor_t mon[SLOT_COUNT];
} acc_mgnt_struct
```

Figure 2. Accelerator Management Structure

The *PRB*, in term, is the interface specific to the individual application accelerator, transforming serialized uniform representation into the accelerator's native format. The PRB hence deserializes the parameters and stores them internally, providing all required parameters simultaneously to the accelerator.

The *AW* packages the accelerator and its according PRB into an exchangeable modular entity.

3.2 Software Components

3.2.1 Accelerator Management Structure

To avoid error-prone pointer arithmetic, we provide a convenient interface for accessing the application accelerators from within C Code. We therefore map a supporting accelerator management structure (see Figure 2) into the memory area of the HT Core.

This structure reflects the order of the hardware components connected to the CSB. The first item of this structure is the structure for steering the reconfiguration controller. The second item can be used to configure the number of concurrent reads of the DMA Unit. The third structure is an array for controlling the individual accelerators. The last array is for status queries to the accelerator monitors.

To access the individual accelerators the application developer must use the `accel_slot_t` structure. This structure contains variables like a parameter sink for passing the parameter to the accelerator or a command structure for triggering the computation.

3.2.2 Driver

A *kernel driver* is required to permit mapping of the HT Core memory range into the address space of user processes. This driver creates two character devices which can be opened and used with the `mmap` system call. The first device represents the HT Core memory range, the latter the DMA memory range. Additionally, the DMA memory range can be read for obtaining the overall size and its location. Before user processes may map both types of memory into their address space, they must already be mapped into the kernel space.

In order to be able to easily use a dedicated DMA memory area, we must prevent the kernel from managing all available physical memory. This simplifies the communication between HT Core and application as no translation

between virtual and physical memory address must be performed. This is done by passing the `mem` parameter to the kernel at boot time. The remainder of the main memory must be made available for handing out to user space. This is achieved through the `ioremap` system call.

Furthermore, the driver provides runtime information and a command interface through the virtual proc file system (procfs), supplying monitoring and controlling possibilities to the Control Daemon. All information provided by the Monitoring Infrastructure can be obtained via *procfs*.

3.2.3 Control Daemon

The Control Daemon is a central resource manager. It manages both, accelerators and DMA memory. It consists of a device handler, an accelerator manager, a memory manager and a notification broker.

The *device handler* is used to abstract the kernel driver interface and handles communication with the driver via its device nodes. It provides functions for mapping and unmapping the Accelerator Management Structure and DMA memory area into the user address space.

The *accelerator manager* is responsible for finding and reserving accelerators of a specific type. The `get_accelerator()` function iterates through the accelerator slots and compares their loaded accelerator type to the requested. The first accelerator with the correct accelerator type and unoccupied status is assigned to the thread. To ensure a clean initial starting environment, a reset followed by a request command are sent to newly acquired accelerators. Likewise, appropriate functions for releasing accelerators are provided.

The *memory manager* handles accesses to the DMA memory area, i.e. requests with sizes being multiples of the systems page size. This restriction is introduced as only complete pages can be mapped into the user address space. The memory manager performs bookkeeping regarding memory areas being already mapped or being free for further request. Two functions are provided for allocating and freeing memory. To minimize fragmentation, an approach similar to free list [11] is used.

Notifying threads when an appropriate application accelerator is finished is the task of the *notification broker* (NB). It monitors the status of the accelerators slots and calls the corresponding thread upon status changes to `finished` or `error`. Being a pure software solution, the NB polls the accelerator status registers periodically. As such a PIO approach fully utilizes one host processor for busy-waiting, we therefore instantiated a POSIX message queue between the NB and the calling thread. These queues support a blocking mode, i.e. reading from an empty or writing to a full queue results in the calling thread being blocked. The NB iterates periodically through the status of each accelerator slot and identifies status changes. For each status change, an event is submitted into the corresponding notification queue. Threads can subscribe to their notification queue and receive according notifications. If no notification is available, the thread will be blocked.

4 Prototypical Implementation

In this section we describe the prototype implementation of our framework based on the HTX Board [6], using the HT Core [14, 13] provided by the University of Heidelberg. We present implementational details of our framework as well as first latency and bandwidth measurements.

4.1 Hardware

The testbed for our prototypical implementation is a system comprising an AMD Opteron 870 dual-core processor, 2 GB of main memory, and a HTX slot[8]. The HTX slot enables the usage of HyperTransport interconnection technology to extend systems which are based on a processor that is HyperTransport-capable. This slot is used to connect a HTX Board with the system, featuring a Xilinx Virtex-4 FX100 FPGA, 128 MB SDRAM, Gigabit Ethernet connectivity, and up to 6 transceiver sockets. An EEPROM is used to store the initial bitstream for initialization after power-on. The FPGA's PowerPC cores are not used in this design.

The protocol handling of the HyperTransport channel is done by the HT Core [14]. The HT Core, provided by the University of Heidelberg, is a soft-core and written in the Verilog hardware description language. In our current synchronous design the HT Core runs at 100MHz clock frequency and provides a 16 bit wide link to the AMD Opteron processor, resulting in a peak bandwidth of 800MB/s for each direction. The HT Core uses 4868 slices, equalling 11.5% of the available slices. Due to I/O constraints, the HT Core is located in the FPGA's lower right corner (see Figure 3). Our current implementation is completely synchronous to the 100 MHz board clock, simplifying communication between individual components.

The floorplan of our current implementation is depicted in Figure 3. The six available hardware accelerators are located on top. The PRBs are already included in this area. Due to the physical design of the FPGA, the accelerators slots vary in size and available additional resources. The size and available resources are listed in Table 1.

Beneath the accelerators are their AIs and AWs. Both are connected to the CSB, which has a width of 8 bits, and the Data Bus, which has a width of 64 bits. The DMA Unit, bus arbiter, and reconfiguration controller are located in the lower left corner of the floorplan. All hardware components, except HT Core and the accelerators themselves,

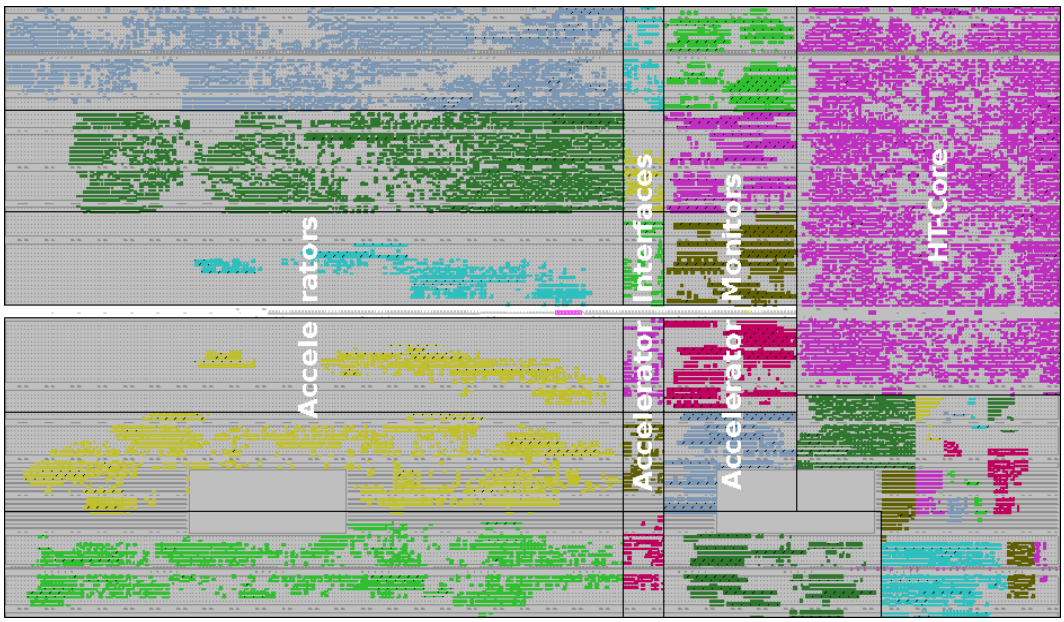

Figure 3. Floorplan

Table 1. Available Resources per slot

Slot	Slices	BRAM	FIFOs	DSPs
1	3944	40	40	-
2	3656	40	40	46
3	4136	23	23	-
4	4136	23	23	46
5	4136	46	46	-
6	4136	46	46	-

Table 2. Occupied Resources

Component	Slices	FF	Carry	Mux	LUTs
Request Coder	201	234	-	34	252
DMA-Unit	1468	943	1043	312	2249
Accel. Interface	600	618	372	204	762
Accel. Monitor	2928	3138	6132	1296	5466
Reconf. Controller	16	15	-	-	19
Overall	5213	4948	7547	1846	8748
HT Core	4868	5257	719	209	7382
Complete Design	10081	10205	8266	2055	16130

occupy 5213 slices, equalling 12.3% of the available slices. Table 2 shows the resource usage of each component.

4.2 Software

For easing the use of the application accelerators, we developed a library containing individual functions for system initialization, acquiring and releasing the accelerators and DMA memory, starting the accelerators, and for NB communication. In its current implementation, the library can be used with C and C++ applications.

4.3 Bandwidth and Latency Measurements

We developed a simple accelerator for measuring the bandwidth from and to main memory. This accelerator reads and writes a predefined number of QWords (64 Byte) from/to the main memory. The accelerator monitor is used to measure the duration of the operations. Because HyperTransport requests are limited to eight QWords, the DMA Unit prevents sole sequential read requests, but rather performs multiple concurrent read requests. Figure 4 depicts the achieveable read bandwidth when using multiple concurrent requests. A sole request achieves a bandwidth of 103MB/s. The maximum is reached when using five or more concurrent requests. A sustained bandwidth of about 311MB/s can be achieved. The write bandwidth is much higher, as no response from main memory is needed. Using only one write request, a bandwidth of 763MB/s can be achieved.

The accelerator monitors are also used for measuring the read request latency. In our current implementation, each read request shows a latency of 52 cycles. The DMA Unit needs three cycles to initiate a write request. But as HyperTransport has no acknowledge signal for indicating a suc-

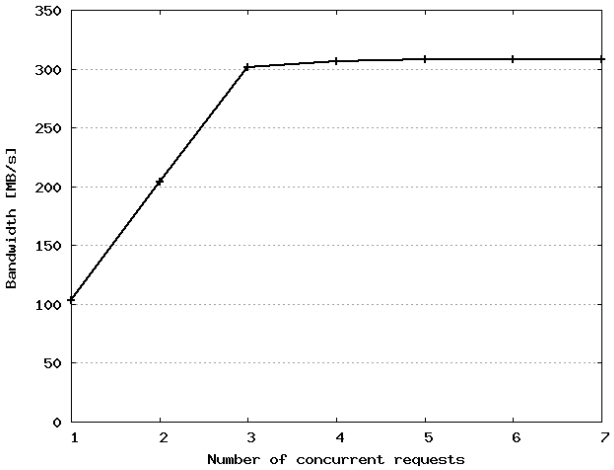

Figure 4. Achieved read bandwidth

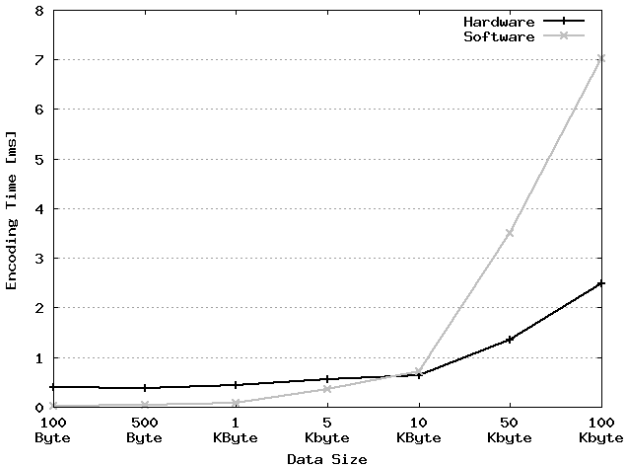

Figure 5. 3DES: Encoding time for software or hardware encryption

cessful write, the entire write latency cannot be measured.

5 Evaluation

Our framework is designed for acceleration of compute-intensive kernels, we therefore present in this section an illustrative example. This example shows that even common and widely used applications can benefit from hardware acceleration.

We implemented a dedicated hardware accelerator performing 3DES, an improved version of the standard DES algorithm, performing three individual DES rounds, using a different key for each round.

Along with the hardware accelerator, we implemented a complete software application enabling en- and decryption of files. Besides en- and decryption with one or multiple hardware accelerators, the application is capable of performing multi-threaded en- and decryption in software using the OpenSSL [5] library.

The basis of the hardware accelerator is the freely available 3DES core provided by CoreTex Systems [4]. This core is implemented in synthesizeable VHDL, performs operations on blocks of data with a size of 64 bit as required by DES, and has a maximum bandwidth of 581MBit/s at 162 MHz. We implemented a custom PRB for interfacing this core to our hardware setup.

The software application can be controlled via command line arguments at start time or through the control daemon at runtime. Some parameters have to be specified at start time, e.g. source and destination file, the individual keys, or the number of concurrent threads or accelerators to be used.

In our first experiment we measured the time needed for encryption using one thread or hardware accelerator for files of varying size resulting in the runtimes shown in Figure 5. As we can clearly see, for smaller file sizes, the software implementation is faster than the hardware accelerators due to data transfer overhead resulting from copying data into the accelerator and back to main memory.

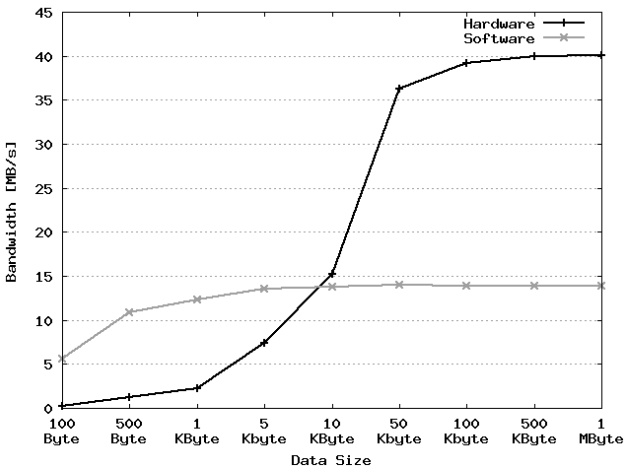

Figure 6. 3DES bandwidth

Figure 6 shows the achieved bandwidth. The pure software implementation has a peak bandwidth of about 14MB/s, the hardware accelerator of about 40MB/s.

We also conducted an experiment with multiple threads or hardware accelerators. The results are shown in Figure 7 and Figure 8. To avoid interference from slow hard drive accesses, all data is read from main memory and written back to main memory. A file of 500 MB was used in this experiment. As we can see, the hardware implementation scales almost perfectly with the number of used ac-

celerators. With six accelerators a peak bandwidth of about 241MB/s is achieved. As our test system has only two processor cores, the software implementation scales only with up to two threads. When using six threads the bandwidth naturally increases only marginal to 29.7MB/s at most.

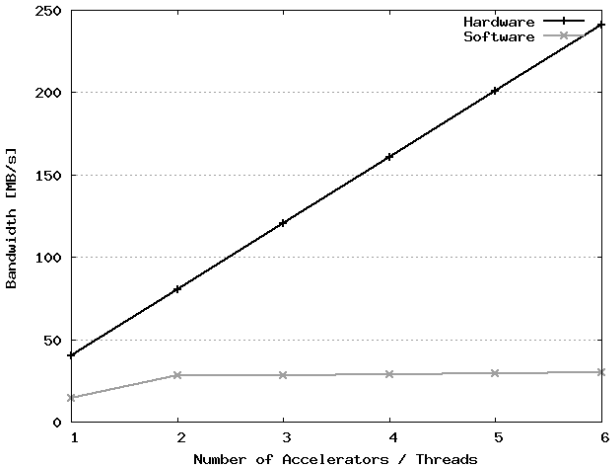

Figure 7. 3DES: Bandwidth – concurrent encryption

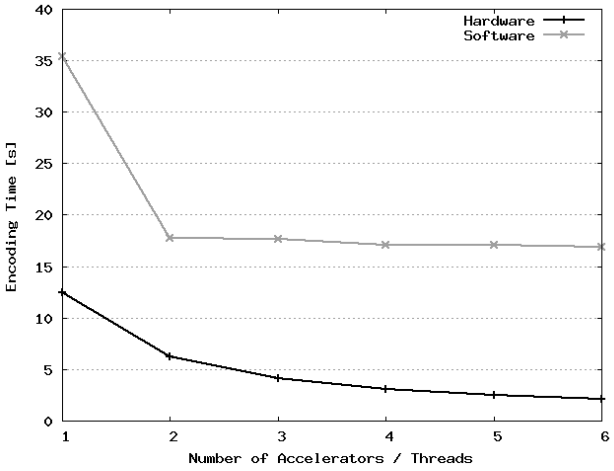

Figure 8. 3DES: Duration – concurrent encryption

6 Outlook

The achieved results in Section 5 showed the usefulness of our architecture. Ongoing work focuses on security aspects and partial dynamic reconfiguration.

In our current implementation of the software stack, the accelerator management structure, as shown in Figure 2, is mapped completely into each user process. That implies that all processes can access each available accelerator, leading to incorrect results. Hence, we are currently evaluating the possibility of mapping an accelerator specific management structure into the user process, granting only access to the assigned accelerator. Other security concerns arise with granting main memory access to the accelerators. Each accelerator has a direct access to the complete main memory. Incorrect memory access could lead to incorrect results, or even worse to a corrupt system. Therefore, we are currently extending our monitoring infrastructure to observe the main memory accesses of the accelerators and prevent them of accessing memory regions which are not assigned to them.

The extension of the reconfiguration controller and generation of partial bitstream is another ongoing work. The reconfiguration controller will be extended by an interface to the FPGAs Internal Configuration Access Port (ICAP). ICAP can be used for partial reconfiguration of the FPGA. In combination with the DMA Unit, the reconfigration controller should be able to load upon request partial bitstream from main memory and forward them to the ICAP port.

7 Conclusion

With the emergence of new interconnection technology like HyperTransport, the interconnection between application specific accelerators and the general-purpose processor is no longer a bottleneck. Older bus systems such as PCI could not provide the required bandwidth or direct access to the systems main memory to successfully accelerate compute-intensive kernels. HyperTransport provides a flexible, low-latency and high-bandwidth interconnection between both, invidiual processors as well as processors and peripheral components.

In this paper, we presented a versatile HyperTransport-based architecture providing application-specific hardware accelerators. The concept itself, while implemented using standard PC technology and the HTX Reference Platform as introduced in Section 4, is generally applicable and may be applied to arbitrary HT-equipped embedded or high-performance computing systems.

References

[1] HyperTransport™ I/O Link Specification Revision 3.10. 2008. http://hypertransport.org/docucontrol/HTC20051222-00046-0028.pdf.

[2] P. Athanas and H. Silverman. Processor reconfiguration through instruction-set metamorphosis. *Computer*, 26(3):11–18, Mar 1993.

[3] R. Buchty, D. Kramer, M. Kicherer, and W. Karl. A Lightweight Approach to Dynamical Run-time Linking Supporting Heterogenous, Parallel, and Reconfigurable Architectures. In *Proceedings of the 22st International Conference on Architecture of Computing Systems (ARCS 2009)*, pages 60–71. Springer, 2009.

[4] L. CoreTex Systems. Triple-DES Encryption+Decryption Core, November 2006. `http://www.opencores.org/projects.cgi/web/3des\_vhdl/overview`.

[5] Eric Young. *OpenSSL Crypto Library Manual*. The OpenSSL Project. `https://www.openssl.org/docs/crypto/des.html`.

[6] H. Fröning, M. Nüessle, D. Slogsnat, H. Litz, and U. Brüning. The HTX-Board: A Rapid Prototyping Station. In *3rd annual FPGAWorld Conference*, 2006.

[7] J. Hauser and J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:12, 1997.

[8] HTX3™Specification for HyperTransport 3.0 Daughtercards and ATX/EATX Motherboards. June 2008. http://www.hypertransport.org/docs/uploads/HTX3_ Specifications.pdf.

[9] C. Inc. Cray XD1 Supercomputer, 2004. `http://www.cray.com/downloads/Cray\_XD1\_Datasheet.pdf`.

[10] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, Volume 38, Number 8, 19. April 1965.

[11] Robert J. Baron and Linda G. Shapiro. *Data Structures and Their Implementation*. PWS Publishing Co., Boston, MA, USA, 1983.

[12] Silicon Graphics, Inc. SGI RASC RC100 Blade (Datasheet). 2008.

[13] D. Slogsnat, A. Giese, and U. Brüning. A versatile, low latency HyperTransport core. In *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pages 45–52, New York, NY, USA, 2007. ACM.

[14] D. Slogsnat, A. Giese, M. Nüssle, and U. Brüning. An open-source HyperTransport core. *ACM Trans. Reconfigurable Technol. Syst.*, 1(3):1–21, 2008.

[15] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte. The MOLEN Polymorphic Processor. *IEEE Transactions on Computers*, 53(11):1363–1375, 2004.

[16] M. Wirthlin and B. Hutchings. A dynamic instruction set computer. *FPGAs for Custom Computing Machines, 1995. Proceedings. IEEE Symposium on*, pages 99–107, Apr 1995.