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)**, held Feb. 12th, 2009, Mannheim, Germany
ISBN: 978-3-00-027249-3

EDITORS

Holger Fröning — Universität Heidelberg, Germany
Mondrian Nüssle — Universität Heidelberg, Germany
Pedro Javier García García — Universidad de Castilla-La Mancha, Spain

KEYNOTE SPEAKER

Prof. José Duato — Universidad Politècnica de Valencia, Spain

PROGRAM COMMITTEE

Francisco J. Alfaro — Universidad de Castilla-La Mancha, Spain
Ulrich Brüning — Universität Heidelberg, Germany
José Duato — Universidad Politècnica de Valencia, Spain
Hans Eberle — Sun Microsystems, USA
Holger Fröning — Universität Heidelberg, Germany
Pedro Javier García García — Universidad de Castilla-La Mancha, Spain
Brian Holden — HyperTransport Consortium, USA
Wolfgang Karl — Universität Karlsruhe, Germany
Mondrian Nüssle — Universität Heidelberg, Germany
Rich Oehler — AMD, USA
Sven-Arne Reinemo — Simula Research Lab, Norway
Jeff Underhill — AMD, USA
Sudhakar Yalamanchili — Georgia Tech, USA

INFORMATION ON PUBLICATION

To ensure a high level of academic content, a peer review process has been used. Each submission has been reviewed by a minimum of two separate reviewers on the Program Committee list.

The proceedings are available electronically at the website of the HyperTransport Center of Excellence as well as on HeiDOK, the Open Access document server of the University of Heidelberg (see links below). This publication platform offers free access to full-text documents and adheres to the principles of OpenAccess as well as the goals of the Budapest Open Access Initiative (BOAI). The papers are accessible through a special sub-portal and are fully citable.

The Open Access Document Server of the University library of Heidelberg also offers the possibility to order hardcopies of the proceedings.

Open Access Document Server:
http://archiv.ub.uni-heidelberg.de/volltextserver/portal/whtra09

Workshop Website:
http://whtra2009.uni-hd.de

HyperTransport Center of Excellence:
http://htce.uni-hd.de

WELCOME MESSAGE FROM THE EDITORS

As organizers of the First International Workshop on HyperTransport Research and Applications (WHTRA), it is our pleasure to present these proceedings, and we hope you will find them interesting and useful.

In response to the WHTRA call for papers, we received interesting submissions, covering either research on key aspects of HyperTransport technology or applications of HyperTransport in different systems. Each of these papers has been carefully and rigorously reviewed by three members of the Program Committee, which have provided not only detailed evaluations of the submissions but also valuable suggestions to enhance them. As a result of the review process, we have selected the seven papers which compose the present proceedings, and whose scope and high technical quality make them, in our opinion, very relevant for the HT community.

Of course, we would like to thank all the members of the Program Committee for their great amount of effort and time they devoted to support this first edition of WHTRA. All PC members are experts of the highest level, from both industry and academia, and their collaboration has been essential for the existence of this workshop.

We would also like to especially thank Prof. José Duato for accepting to deliver the opening keynote of the workshop. Taking into account the experience and brilliance of this prominent researcher, we are sure this keynote was one of the strongest contents we could add to the WHTRA program.

We have to thank the University of Heidelberg and the HyperTransport Center of Excellence for hosting this event, and the University Library of Heidelberg for publishing the proceedings.

Finally, we would like to thank authors and attendees for their interest in this first WHTRA. We hope all of them had a very nice and productive meeting.

Holger Fröning[*], Mondrian Nüssle[*] and Pedro Javier García García[†]

[*] Universität Heidelberg, Germany
[†] Universidad de Castilla-La Mancha, Spain

CONTENTS

A HyperTransport-Enabled Global Memory Model For Improved Memory Efficiency

Jeffrey Young, Sudhakar Yalamanchili*
Georgia Institute of Technology
jyoung9@gatech.edu, sudha@ece.gatech.edu

Federico Silla, Jose Duato
Universidad Politecnica de Valencia, Spain
{fsilla, jduato}@disca.upv.es

Abstract

Modern data centers are presenting unprecedented demands in terms of cost and energy consumption, far outpacing architectural advances. Consequently, blade designs exhibit significant cost and power inefficiencies, particularly in the memory system. We propose a HyperTransport-enabled solution called the Dynamic Partitioned Global Address Space (DPGAS) model for seamless, efficient sharing of memory across blades in a data center, leading to significant power and cost savings. This paper presents the DPGAS model, describes HyperTransport-based hardware support for the model, and assesses this model's power and cost impact on memory intensive applications. Overall, we find that cost savings can range from 4% to 26% with power reductions ranging from 2% to 25% across a variety of fixed application configurations using server consolidation and memory throttling. The HyperTransport implementation enables these savings with an additional node latency cost of 1,690 ns latency per remote 64 byte cache line access across the blade-to-blade interconnect.

1. Introduction

The current solution to satisfying increasing demand for memory on a blade server is to provision memory on each blade for the worst case demand. One recent study empirically measured memory footprints from non-virtualized applications across 3,000 servers under normal applications and found the average physical memory usage to be about 1 Gigabyte [3]. However, this study also found that memory requirements can vary greatly, with 50% of the applications requiring between 1 GB and 4 GB of memory at certain points during the five-week period of data collection. Thus, provisioning blade memory for the average case can prove to be inadequate with respect to the subsequent page fault rate while provisioning for the worst-case memory footprint can lead to servers that are substantially overprovisioned and consequently expensive and power inefficient. Furthermore, the cost of DRAM is a non-linear function of density and memory size, thus small increases in provisioned memory lead to disproportionate increases in cost.

We hypothesize that while memory demands of individual applications can vary substantially, rarely, if ever, do all applications make peak demands concurrently. The idea proposed by this work is to reduce the cost and power associated with memory by provisioning blades with less than worst-case memory demand and sharing memory across blades during periods of localized, high memory demand. Thus, the physical memory accessible to a blade can vary over time, increasing during periods of peak load by "borrowing" physical memory from an adjacent blade. This idea of shared memory is clearly not new. However, memory sharing via traditional means can exact significant performance penalties through the interconnect and operating system management functions rendering them infeasible in commodity server configurations.

What has changed is the recent introduction of fast interconnects integrated onto the multi-core die close to the memory controllers. The advent of HyperTransport technology reduced the distance from the "wire" to the on-chip memory controller providing low-latency access to remote memory controllers. Thus the hardware cost to access remote memory, e.g., adjacent blades, is no longer prohibitive. However, to productively harness this raw capability, a global system model must be defined to direct how the system-wide memory is allocated/accessed and thereby shared across the operating system domains of distinct blades. This is where our approach differs from prior non-uniform memory access (NUMA) architectures. Each blade is under the con-

*This research was supported in part by NSF grant CCF-0874991, and Jeffrey Young was supported by a NSF Graduate Research Fellowship

trol of a distinct OS. However a blade may periodically become a NUMA machine that has access to a portion of the physical memory of an adjacent blade. The advent of on-die integrated HT makes this feasible from a performance perspective.

This paper proposes a dynamic global address space model (DPGAS) by modifying the existing partitioned global address space model (PGAS) [4] to support a global, *noncoherent physical address space* where an application's virtual address space can be dynamically allocated physical memory located on local and remote nodes. Architectural support for address space management is tightly integrated into the HyperTransport interface to minimize the performance overhead of remote memory accesses and to permit fast, dynamic changes in physical address space mappings. Physical memory is dynamically shared by *spilling* memory demand on a blade to neighboring blades as necessary during peak periods. Consequently, the total amount of memory to be provisioned across the data center can be significantly reduced, leading to substantial cost and power savings with minimal loss of performance (an increase in the page fault rate).

Specifically, this paper contributes the following:

1. A physical address space model, Dynamic Partitioned Global Address Space (DPGAS), for managing system-wide physical memory in large-scale server systems.

2. Design, implementation, and evaluation of hardware support for the DPGAS model via a memory mapping unit that is integrated with a HyperTransport local interface and tunnels memory requests via commodity interconnect—in this case Ethernet.

3. An evaluation of DPGAS with 1) traces from memory-intensive applications, 2) an on-demand memory spilling policy to allocate off-blade memory when local demand exceeds available physical memory, and 3) an evaluation of the cost and power savings from more efficient DRAM usage.

The following sections present the model, its architectural support integrated into the HT interface, and a simulation-based evaluation of the potential for cost and power savings.

2. A Dynamic Partitioned Global Address Space model

The DPGAS model is a generalization of the partitioned global address space (PGAS) model to permit flexible, dynamic management of a physical address space at the hardware level—the virtual address space of a process is mapped to physical memory that can span multiple (across blades) memory controllers. The two main components of the DPGAS model are the architecture model and the memory model.

2.1. Architecture model

Future high-end systems are anticipated to be composed of multi-core processors that access a distributed global 64-bit physical address space. Cores nominally have dedicated L1 caches for instructions and data, but may share additional levels of cache amongst themselves in groups of two cores, four cores, etc. A set of cores on a chip will share one or more memory controllers and low-latency link interfaces integrated onto the die such as HyperTransport [15]. All of the cores also will share access to a memory management function that will examine a physical address and route this request (read or write) to the correct memory controller—either local or remote. For example, in the current-generation Opteron systems, such a memory management function resides in the System Request Interface (SRI), which is integrated on-chip with the Northbridge [6].

2.2. Memory model

The memory model is that of a 64-bit partitioned global physical address space. Each partition corresponds to a contiguous physical memory region controlled by a single memory controller, where all partitions are assumed to be of the same size. For example, in the Opteron (prior to Barcelona core), partitions are 1 TB corresponding to the 40-bit Opteron physical address. Thus, a system can have 2^{24} partitions with a physical address space of 2^{40} bytes for each partition. Although large local partitions would be desirable for many applications, such as databases, there are nonintuitive tradeoffs between partition size, network diameter, and end-to-end latency that may motivate smaller partitions. Further, smaller partitions may occur due to packaging constraints. For example, the amount of memory attached to an FPGA or GPU accelerator via a single memory controller is typically far less than 1 TB. Thus, the DPGAS model incorporates a view of the system as a network of memory controllers accessed from cores, accelerators, and I/O devices.

Two classes of memory operations can be generated by a local core: 1) *load/store* operations that are issued by cores to their local partition and are serviced per specified core-semantics, and 2) *get/put* operations that correspond to one-sided read/write operations on memory locations in remote partitions [22].

Coherence is separated from the issues central to defining the DPGAS model because large, scalable coherence is still an unsolved research problem, and many systems do not require full-scale coherence across large numbers of servers. Additionally, coherence can be enforced between the one to eight Opteron-based sockets on a server blade to provide local "islands" of coherence. In this case one can view the DPGAS model as dynamically increasing the size of physical memory (across blades) that is associated with a coherence domain although the specific protocols are beyond the scope of this paper.

A sample get transaction on a memory location in a remote partition must be forwarded over some sort of network to the target memory controller and a read response is transmitted back over the same network. The specific network is not germane to the DPGAS model implementation. However, being constrained by commodity parts, this study utilizes Gigabit Ethernet.

Once the DPGAS memory model is enabled, an application's (or process's) virtual address space can be allocated a physical address space that may span multiple partitions (memory controllers), i.e., local and remote partitions. The set of physical pages allocated to a process can be static (compile-time) or dynamic (run-time). Multiple physical address spaces can be overlapped to facilitate sharing and communication.

This paper is only concerned with a very specific application of DPGAS, namely sharing of memory across blades. Dynamic memory requests at a blade can be satisfied by *spilling*—allocating memory from a neighboring blade with spare capacity. We demonstrate in section 5 that this simple allocation policy can have a significant impact. The following section addresses the feasibility of a hardware implementation.

3. DPGAS: implementation

Hardware support for DPGAS has two basic components. The first is a memory function that distinguishes between local and remote memory requests. The second is a memory mapping unit that maps remote physical addresses to specific destination memory controllers. The former is available in modern processors such as the Opteron. The latter is contributed by this paper and is tightly integrated into the HyperTransport interface as shown in Figure 1. The proposed memory mapping unit or bridge performs several functions, including 1) managing remote accesses, 2) encapsulating remote requests into an inter-blade communication fabric (the demonstrator uses Ethernet), and 3) extending noncoherent HT packet semantics across nodes. This section describes the design and implementation of the bridge.

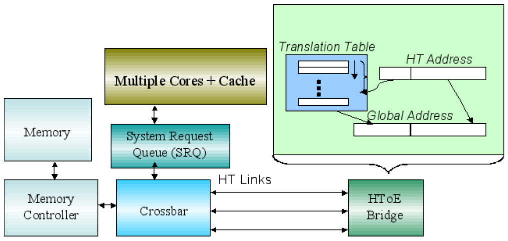
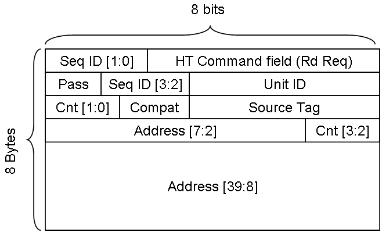


Figure 1. HToE Bridge with Opteron Memory Subsystem



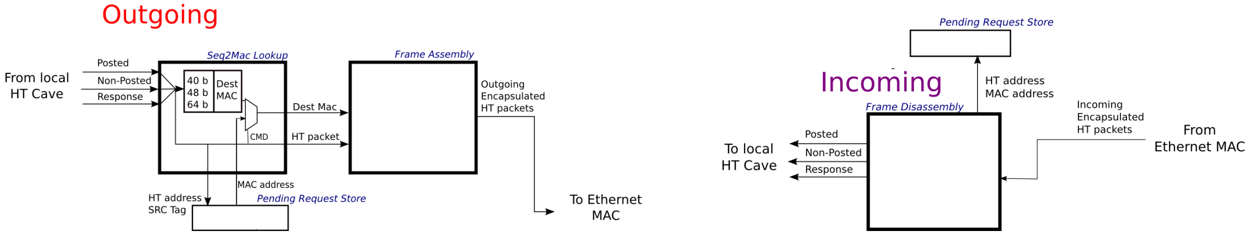
Seq ID [1:0]	HT Command field (Rd Req)	
Pass	Seq ID [3:2]	Unit ID
Cnt [1:0]	Compat	Source Tag
Address [7:2]		Cnt [3:2]
Address [39:8]		

Figure 2. HT read request packet format

3.1. HyperTransport overview

HT is a point-to-point packet switched interconnect standard [15] that defines features of message-based communication, including 1) the use of groups of virtual channels, 2) read/write transactions with posted and non-posted semantics, 3) naming and tracking of multiple outstanding transactions from a source, and 4) specification of ordering constraints between messages. In addition, the HT specification defines flush and fence commands to manage updates to memory on a node. Our model extends the flush command to a remote version while conforming to normal HT ordering and deadlock avoidance protocols.

A typical command packet is shown in Figure 2, where the fields specify options for the read transaction and preservation of ordering and deadlock freedom. Our implementation specifically relies on the UnitID, SrcTag, SeqID, and address fields. The UnitID specifies the source or destination device and allows the local host bridge to direct requests/responses. The SrcTag and SeqID are used to specify ordering constraints between requests from a device, for example, ordering between outstanding, distinct transactions. Finally, the address field is used to access memory that is mapped to either main memory or HT-connected devices. An extended HT packet can be used that builds on this format



(a) Outgoing datapath

(b) Incoming datapath

Figure 3. HToE Bridge Components

to specify 64-bit addresses [15].

3.2. HyperTransport over Ethernet—address translation and Ethernet encapsulation

Our demonstrator is based on the use of Ethernet as the commodity inter-blade interconnect primarily due to ready availability of hardware implementations. The bridge design itself does not rely on Ethernet and is easily replaced with other commodity or specialized interconnects. We refer to this demonstrator bridge as the *HT-over-Ethernet (HToE)* implementation. The HToE bridge implementation uses the University of Heidelberg's HyperTransport Verilog implementation [25], which implements an noncoherent HT cave (endpoint) device. Our bridge interfaces with the Heidelberg core so that we can demonstrate functionality with a realistic HT cave implementation. Figures 3(a) and 3(b) show the outbound and inbound components of our HToE bridge along with interface signals for the Heidelberg core and Ethernet MAC.

The HToE implementation is based on a system with Opteron nodes where each Opteron node has an Ethernet-enabled FPGA card available in the HTX connector slot, such as the University of Heidelberg HTX card [2]. Several nodes are connected via an inexpensive Ethernet switch, and it is assumed that HyperTransport messages sent to remote addresses via the HToE bridge are routed using one of two methods: 1) access to the northbridge address mapping tables (via the BIOS) in order to specify the physical address space mappings for the HToE bridge device, or 2) an intelligent MMU that distinguishes between accesses to the local memory and the I/O address space and HT packets that are sent for non-local addresses through the HToE bridge.

Consider a system that has been properly initialized and consider an application that generates a read operation to an address that is in a remote partition. There are three stages in each individual communication operation (e.g., a read request command) at a given source host and attached devices: 1) extension from the 40-bit physical address in the Opteron to the 64-bit physical address, 2) creation of a HT packet that includes a 64-bit extended address, and 3) mapping the most significant 24 bits in the destination address to a 48-bit MAC address and encapsulation into an Ethernet frame. An efficient implementation could pipeline the stages to minimize latency, but retaining the three stages has the following advantages: 1) It separates the issues due to current processor core addressing limitations from the rest of the system, which will offer a clean, global shared address space, thus allowing implementations with other true 64-bit processors, and 2) it will be easy to port to other platforms that do not encapsulate by using Ethernet frames, but use other link layer formats such as Infiniband. Thus, some efficiency was sacrificed for initial ease of implementation and for a cleaner, modular design.

First, the HT packet type is decoded into a request or response command packet in the module called Seq2Mac in Figure 3(a). For request packets the two most significant bits of the 40-bit address are decoded to select one of four partition registers to access the 24-bit partition address—the two most significant bits in the 40-bit address used to address the partition register are reset in parallel with the access to the partition register. Now three pieces of information are needed: 1) the extended 24-bit address to form an HT read request packet with extended address, 2) the MAC address of the destination bridge to encapsulate the extended HT packet into Ethernet, and 3) the local MAC address, according to Ethernet frame format to enable the response. Item 3 has been set during initialization, and access to the source MAC address is not in the critical path. Items 1 and 2 have a direct correspondence among them—given a destination node ID or the remote partition address, there is a unique MAC address associated with both data fields. Therefore, the partition register can store both the 24-bit partition address and the destination MAC address together, thus reducing access time when forming the Ethernet frame. Once the remote MAC address and the 64-bit address have been found in the partition table, the new HT packet is constructed and encapsulated in a standard Ethernet packet, illustrated in the figure as the Ethernet Frame Assembly module. The encapsulated packet is then buffered until it can be sent using the local node's Ethernet MAC and the physical Ethernet interface. For packets that send a set amount of data, the control and data packets must be buffered until all the data has been encapsulated into Ethernet frames.

The receive behavior of the bridge on the remote node will require a "response matching" table where it will store, for every non-posted HT request (request that requires a response), all the information required to route the response back to the source when it arrives. This table is required since HT is strictly a local interconnect and response packets have no notion of a destination 40-bit (or extended 64-bit) address. Since the formats of HT request and response packets differ and this implementation desires not to change local HT operation, the SrcTag field of each packet is used to match MAC addresses from an incoming request packet with an outgoing response packet. Note that each request packet contains the source MAC address, and this is the address stored in the "response matching" table and later used as the destination MAC address for the corresponding response. Encapsulation and buffering occur once again until the response and data can be transmitted over Ethernet. In the HToE bridge, this module is listed as the Pending Request Store in Figure 3(b) and is shared between incoming and outgoing packets.

It should also be noted that since HT SrcTags are 5 bits, a maximum of 32 outstanding requests can be handled concurrently using the Pending Request Store. This limitation means that additional requests must be queued in the bridge until space is free in the Pending Request Store. If two request packets arrive with the same SrcTag, then the latter packet is remapped before being stored in the table. When the corresponding response leaves the HToE bridge, the SrcTag is mapped back to its original value to ensure proper HT routing on the requesting local node. Once the response reaches the local HToE bridge that initiated the read request, the HT packet is removed from its Ethernet encapsulation. The UnitID is changed again to that of the local host bridge and the bridge bit is set to send the packet upstream. This allows the local host bridge to route responses to the originating HT device. Other transactions, such as a posted write or a non-posted write, involve similar sequences of events. The differences in these transactions are that for posted writes, no data is stored to create a response; for non-posted writes, only a "TargetDone" response is returned and no data needs to be buffered before the response is sent over Ethernet. Similarly, atomic Read Modify Write commands can be treated as non-posted write commands for the purposes of this model.

Table 1. Latency results for HToE bridge

DPGAS operation	Latency (ns)
Heidelberg HT Core (input)	55
Heidelberg HT Core (output)	35
HToE Bridge Read (no data)	24
HToE Bridge Response (8 B data)	32
HToE Bridge Write (8 B data)	32
Total Read (64 B)	1692
Total Write (8 B)	944

4. DPGAS: evaluation of hardware support

Memory mapping is on the critical path for remote accesses. This section reports on the evaluation of a hardware implementation of DPGAS support, the bridge, and the integration into the HyperTransport interface and remote extensions to the HyperTransport protocol required to support DPGAS.

4.1. Bridge implementation

Xilinx's ISE tool was used to synthesize, map, and place and route the HToE Verilog design for a Virtex 4 FX140 FPGA. Synthesis tests using Xilinx software have indicated that the four major modules that make up the bridge are individually capable of speeds in excess of 160 MHz—combined, unoptimized results indicate that the HT bridge is more than capable of feeding a 1 Gbps or faster Ethernet adapter with a 125 MHz clock speed. Evaluations for each of the request and reply critical paths suggest that the latency overhead of the bridge is on the order of 24 to 72 ns (for a control packet with no data and a read request response with eight doublewords of data, respectively). In a Xilinx Virtex 4 FX140 FPGA, an unoptimized placement of the bridge uses approximately 1,300 to 1,500 slices, or approximately 5% to 6% of the chip. Overheads that reduced performance included the use of a serial Gigabit Ethernet MAC interface and the use of only one pipeline to handle packets for each of the three available virtual channels. The latency results for our bridge, the Heidelberg core (used to interface with our bridge) [25], and total latency for the entire path from local to remote memory are listed in Table 1. The bridge latency numbers assume a 125 MHz clock and discount any serialization latency normally associated with Xilinx Ethernet MAC interfaces.

4.2. Bridge and memory subsystem latencies

While our synthesis results proved that the HToE bridge is low-latency, it is also important to understand the overall latency penalty that the memory subsystem contributes to remote memory accesses. The latency values for the HToE bridge component and related Ethernet and memory subsystem components were obtained from statistics from other studies [6] [25] [17] and from the above place and route timing statistics for our bridge implementation. An overview is presented in Table 2. Our HToE implementation was based on a 1 Gbps Ethernet MAC included with the Virtex 4 FPGA, but latency numbers were not available for this IP core. 10 Gbps Ethernet numbers are shown in this table to demonstrate the expected performance with known latency numbers for newer Ethernet standards.

Table 2. Latency numbers used for evaluation of performance penalties

Interconnect	Latency (ns)
AMD Northbridge	40
CPU to on-chip memory	60
Heidelberg HT Cave Device	35 - 55
HToE Bridge	24 - 72
10 Gbps Ethernet MAC	500
10 Gbps Ethernet Switch	200

Utilizing the values from Tables 1 and 2 for using the HToE bridge to send a request to remote memory, the performance penalty of remote memory access can be calculated using the formula:

$$t_{rem\_req} = t_{northbridge} + t_{HToE} + t_{MAC} + t_{transmit}$$

where the remote request latency is equal to the time for an AMD northbridge request to DRAM, the DPGAS bridge latency (including the Heidelberg HT interface core latency), and the Ethernet MAC encapsulation and transmission latency. This general form can be used to determine the latency of a read request that receives a response:

$$t_{rem\_read\_req} = 2{*}t_{HToE\_req}\ 2{*}t_{HToE\_resp} + 2{*}t_{MAC} + 2{*}t_{transmit} + t_{northbridge} + t_{rem\_mem\_access}$$

These latency penalties compare favorably to other technologies, including the 10 Gbps cut-through latency for a switch, which is currently 200 ns [23]; the fastest MPI latency, which is 1.2 μs [21]; and disk latency, which is on the order of 6 to 13 ms for hard drives such as those in one of the server configurations used below for the evaluation of DPGAS memory sharing [26]. Additionally, this unoptimized version of the HToE bridge is fast enough to feed a 1 Gbps Ethernet MAC without any delay due to encapsulating packets. Likely improvements for a 10 Gbps-comptable version of the HToE bridge would include multiple pipelines to allow processing of packets from different virtual channels and the buffering of packets destined for the same destination in order to reduce the overhead of sending just one HT packet in each Ethernet packet in the current version.

5. DPGAS: evaluation of memory sharing

In the absence of a full hardware testbed, we employ a trace-driven analysis of the potential savings offered by a DPGAS implementation. Virtual address traces were acquired using an instrumented SIMICS 3.0.31 model [20] and fed through an internally developed C++ page table simulator to determine the number of page faults as a function of physical memory footprints ranging from 32 MB to 1 GB. Five benchmarks were selected: Spec CPU 2006's MCF, MILC, and LBM [11]; the HPCS SSCA graph benchmark [1]; and the DIS Transitive Closure benchmark [7]. These benchmarks had maximum memory footprints ranging from 275 MB to 1600 MB. A 2.1 billion address trace (with 100 million addresses to warm the page table) was sampled from memory intensive program regions of each benchmark.

5.1. Memory allocation

We analyze the impact of DPGAS by simulating a workload allocation across a multiblade server configuration using a simple greedy bin packing algorithm. An application is randomly selected and its maximum memory footprint is allocated on a random blade. This process is repeated until some termination criterion is met, e.g., allocation failure, fixed workload, etc. The workload is recorded and the same set of memory footprints is allocated across the same server configuration using DPGAS as follows. When an application cannot be allocated on a blade due to a lack of memory, additional memory is allocated on an adjacent blade, i.e., *spilling* the memory request. This is repeated until all application footprints have been allocated.

Two HP Proliant server configurations were selected for analysis, representing high-end and low-end performance points. Both configurations are expected to execute at least two instances of a benchmark application trace per core. These server configurations are detailed in Table 3. All associated system and memory costs and power statistics were derived from [13] and [14].

Table 3. HP Proliant server configurations

Model (HP)	CPU Cores (Opterons)	Max. Memory	Base Cost/Power
DL785 G5	8 quad-core 2.4 GHz	512 GB	˜$42,000/1110 W
DL165 G5	2 quad-core 2.1 GHz	64 GB	˜$2,000/197 W

5.2. Cost and power evaluation

Results from two classes of experiments are shown here based on results from experiments (as described in Section 5.1) averaged over 50 iterations.

5.2.1. Fixed workload and scale out. These experiments considered fixed workloads where a workload is a fixed number of applications. We based our workload model results from Intel's study of candidate applications for virtualization [3] where an average number of applications per core were identified. We extrapolated this number to a data center with 250 servers (which translates to 2,000 or 500 processor sockets for our server configurations) that could support either 19,500 applications using high-end servers or 4,700 applications using low-end servers. Additionally, we investigated the effects of scaling the number of blades while keeping the workload fixed.

This set of experiments used a baseline configuration with a fixed 64 GB of memory per blade and standard bin packing allocation where application memory footprint had to reside within a blade. The resulting fragmentation left unused memory across blades although several blades exhibited very high memory utilization (in excess of 60 GB). For comparison purposes, we considered a DPGAS-enabled server configuration where half the blades were provisioned with 64 GB and half with less memory. The aggregate difference in memory is roughly equal to the unutilized memory in the first configuration. This latter configuration corresponds to a data center with half of the servers overprovisioned (receivers in our model) and half of the servers minimally provisioned (spill memory to other nodes). Finally we repeated the experiment with 56 GB per blade rather than 64 GB, which reduced memory fragmentation.

The total cost savings for the low- and high-end server configurations are shown in Figures 4 and 5 with savings between standard and DPGAS allocation graphed as the third column of each group. As we see in the base (250-server) case, DPGAS has the potential to save 15% to 26% in memory cost when the initial provisioning is high (64 GB), which translates into a $30,736 savings for the low-end servers and $200,000 for the high-end servers. On the other hand, with lower initial memory (56 GB), the savings in Figure 5 are 13%, or $103,365

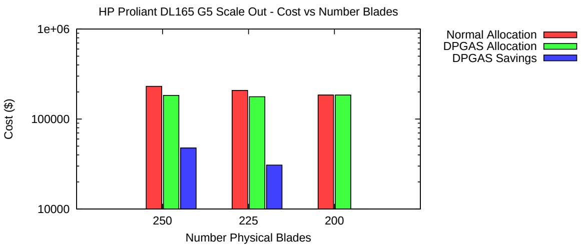


Figure 4. Scale out cost for Proliant DL165 G5

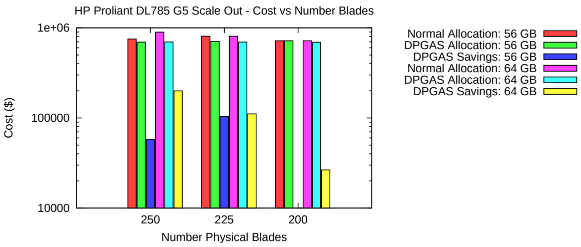


Figure 5. Scale out cost for Proliant DL785 G5

It is also important to notice that savings with DPGAS allocation drops as applications are consolidated onto fewer servers. This is likely due to the fact that there is less fragmentation with no sharing and therefore less inefficiency to be recovered.

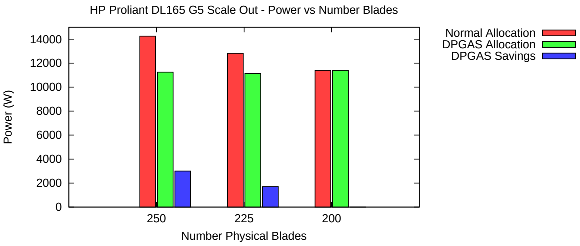


Figure 6. Scale out power for Proliant DL165 G5

Similarly, the power savings using DPGAS allocation (Figures 6 and 7) is substantial in the base case, with savings of 3,625 (25%) and 5,875 (22%) watts of input power for the low-end and high-end server configurations, respectively. When server consolidation onto 200 servers is used, power savings drops substantially to 800 and 500 watts for the same configurations. The smaller memory configuration results for the high-end server also demonstrate smaller savings of 2500 watts in the 250 server case. Both the cost and power results indicate that DPGAS memory allocation is most effective when fragmentation is normally high and when variance in workload memory footprint is high.

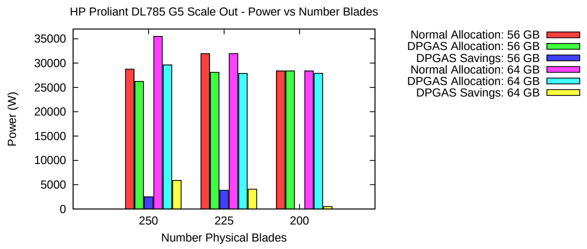


Figure 7. Scale out power for Proliant DL785 G5

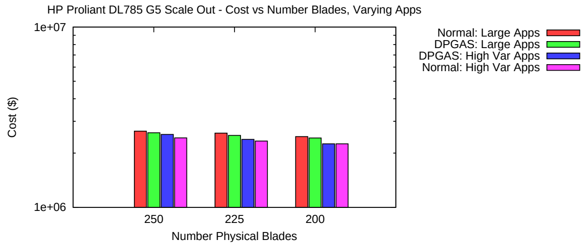


Figure 8. Scale out cost for Proliant DL785 G5 - varying workload sizes

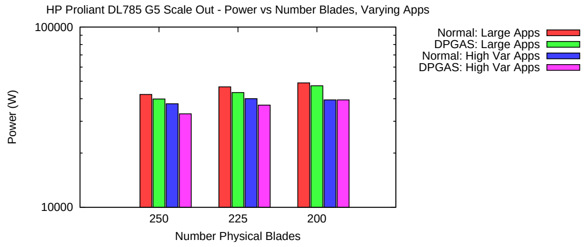


Figure 9. Scale out power for Proliant DL785 G5 - varying workload sizes

To further investigate the effects of memory fragmentation on cost and power, we also ran two separate sets of allocations using workloads drawing from 1) a pool of three applications with large memory footprints and 2) a pool of two applications with small and very large footprints. This experiment included the use of a synthetic benchmark with a memory footprint of 2275 MB that represented a large, unknown enterprise workload similar to those in [3]. The results can be seen in Figures 8 and 9 with cost savings of 2% to 3% for the large applications and 2% to 4% for the second application set. Power savings range from 6% to 7% for large applications and 8% to 12% for the second set of applications. The dropoff in performance can be explained as follows. When application footprints are of similar size, the bin packing behavior of allocation produces little fragmentation, but when applications have small footprints, they can fill unallocated memory and reduce fragmentation. DPGAS seems to work best when the dynamics are such that a wide range of footprints are likely, leading to fragmentation that can be otherwise recovered by DPGAS.

5.2.2. Memory throttling. Memory throttling is where the allocated footprint per application is less than the maximum footprint at the expense of an increased page fault rate. We compared two additional cases with 250 servers: 1) Each server had 50% of the original memory and each application was allocated 50% of its maximum memory footprint, and 2) each server had 25% or the original memory, and each application received 25% of its maximum footprint. The results for cost and power in the high-end server are shown in Figures 10 and 11. The effects of memory throttling are significant. For instance, reducing memory from 64 GB to 32 GB in each server reduces memory cost by $478,000 and memory power by 17,750 watts (from a base cost of $897,000 and base power of 35,500 watts). The usage of DPGAS allocation with 50% memory throttling with the high-end server configuration can reduce the total memory cost by $570,000 and total memory power by 21,125 watts.

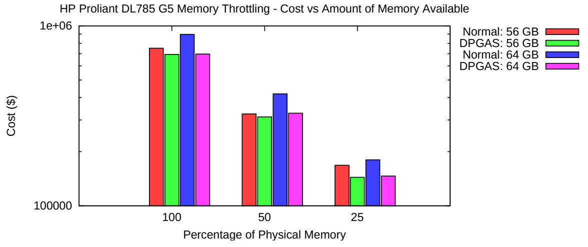


Figure 10. Memory throttling cost for Proliant DL785 G5

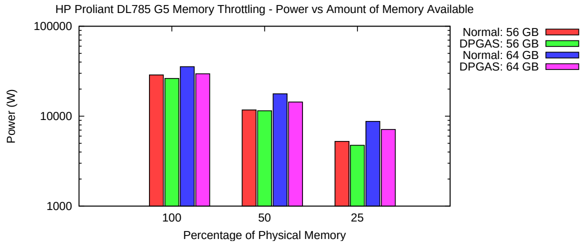


Figure 11. Memory throttling power for Proliant DL785 G5

At the lower bound of savings, reducing memory in the high-end server from 56 GB to 28 GB or 14 GB results in similar savings due to memory throttling, but the savings from using DPGAS is somewhat lower with cost savings of 4% to 14% and power savings of 2% to 10%. This translates to cost savings of $12,000 to

Table 4. HP Proliant 165 G5 cost and power with memory throttling

Allocation	No Throttling	50% Throttling	25% Throttling
Normal ($)	$230,750	$111,250	$51,500
DPGAS ($)	$183,000	$97,125	$51,500
Normal (W)	14,250	7,000	3,500
DPGAS (W)	11,250	5,875	3,500

$24,000 over the normal case and power savings of 250 to 500 W, using 50% and 25% memory throttling.

Additional statistics for the low-end server configuration are shown in Table 4. These experimental results concur with the high-end server configuration, except that power and cost savings are smaller due to less memory fragmentation and less memory overall for remote sharing. In the 25% memory throttling case, there is not enough leftover memory to be utilized with DPGAS, so no savings are incurred. Overall, DPGAS enables a 4% to 22% reduction in memory cost and a 2% to 25% reduction in memory power when compared to normal allocation for both the low- and high-end servers.

When using memory throttling, performance must also be taken into account. The results from our trace-driven analysis of the benchmark applications provide data on page fault rates that directly correspond to the amount of memory a benchmark is allocated. These results are used to generate Figures 12 and 13 that demonstrate the effects of memory throttling on random allocations of each of our benchmark applications. In general, the usage of memory throttling leads to an order-of magnitude increase in the number of page faults for all applications, but some applications with small memory footprints or random access patterns (poor spatial reuse) are affected much more by using memory throttling with normal allocation.

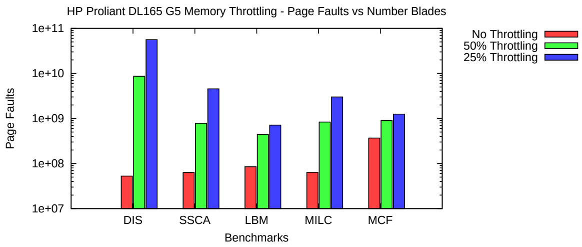


Figure 12. Memory throttling performance for Proliant DL165 G5

6. Related Work

Other researchers have also been focused on the growing power and cost implications of large clusters and server farms. Feng, et al. [5] discussed the efficiencies associated with large servers and proposed a power-efficient supercomputer called Green Destiny. Other strategies have included dynamic voltage scaling for power-aware computing [10] with a focus on CPU power. Raganathy, et al. [24] has also suggested that power-management should take place at the server enclosure levels so that individual systems are not overprovisioned. This study also focused mainly on high-level CPU power management, not memory power.

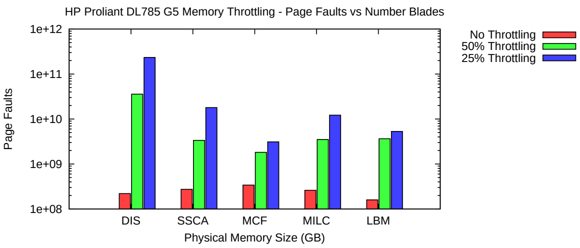


Figure 13. Memory throttling performance for Proliant DL785 G5

However, Lefurgy's 2003 study [18] cited important reasoning behind why DRAM cost and power should be considered as a major component in improving overall server efficiencies. Several other researchers have also begun focusing on memory power management at the architecture level, including [16], which proposes using adaptive power-based scheduling in the memory controller, and [9], which uses power "shifting" driven by a global power manager to reduce power of the overall system based on runtime applications.

At the operating system level, [12] proposed a power-aware paging method that utilizes fast MRAM to provide power and performance benefits. Tolentino [27] also suggested a software-driven mechanism to limit application working sets at the operating system level and reduce the need for DRAM overprovisioning.

An evaluation of power and cost trends similar to the ones in this paper was conducted in [19], concluding that separate PCI Express-based memory blades could be used to reduce overall memory usage and memory cost and power. [8] investigated real-world statistics for some of the large "warehouse-sized" server farms that Google runs.

7. Conclusion

With increasing server power and cost outpacing related performance gains, a focus on making data centers and clusters as efficient as possible is vital from a business perspective. We present a new address space model, the Dynamic Partitioned Global Address Space, and define an associated dynamic hardware-based address translation scheme for efficiently utilizing remote memory with low-latency interconnects such as HyperTransport. An implementation of this model has been developed by encapsulating HyperTransport packets in Gigabit Ethernet via our HT over Ethernet bridge, and initial synthesis results indicate that remote read and write operations are low-latency and comparable to fast message-passing implementations. The impact of low-latency remote access on the ability to share memory is significant, and future plans include the pursuit of a HW/SW testbed to evaluate a complete solution. Additional future work is described in [28].

References

[1] David A. Bader and Kamesh Madduri. Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. In *HiPC*, pages 465–476, 2005.

[2] Ulrich Bruening. The htx board: The universal htx test platform. `http://www.hypertransport.org/members/u_of_man/htx_board_data_sheet_UoH.pdf`.

[3] S. Chalal and T. Glasgow. Memory sizing for server virtualization. 2007. `http://communities.intel.com/docs/`.

[4] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05*, pages 519–538, New York, NY, USA, 2005. ACM.

[5] Wu chun Feng. Making a case for efficient supercomputing. *ACM Queue*, 1(7):54–64, 2003.

[6] Pat Conway and Bill Hughes. The amd opteron northbridge architecture. *IEEE Micro*, 27(2):10–21, 2007.

[7] Dis stressmark suite, updated by uc irvine. 2001. `http://www.ics.uci.edu/~amrm/hdu/DIS_Stressmark/DIS_stressmark.html`.

[8] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *"ISCA 2007: Proceedings of the 34th annual international symposium on Computer architecture"*, pages 13–23, New York, NY, USA, 2007. ACM.

[9] Wes Felter, Karthick Rajamani, Tom Keller, and Cosmin Rusu. A performance-conserving approach for reducing peak power consumption in server systems. In *ICS '05*, pages 293–302, New York, NY, USA, 2005. ACM.

[10] Rong Ge, Xizhou Feng, and Kirk W. Cameron. Improvement of power-performance efficiency for high-end computing. In *IPDPS '05*, page 233.2, Washington, DC, USA, 2005. IEEE Computer Society.

[11] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[12] Y. Hosogaya, T. Endo, and S. Matsuoka. Performance evaluation of parallel applications on next generation memory architecture with power-aware paging method. *IPDPS '08*, pages 1–8, April 2008.

[13] Hp proliant dl servers - cost specifications. 2008. `http://h18004.www1.hp.com/products/servers/platforms/`.

[14] Hp power calculator utility: a tool for estimating power requirements for hp proliant rack-mounted systems. 2008. `http://h20000.www2.hp.com/bc/docs/support/SupportManual/c00881066/c00881066.pdf`.

[15] Hypertransport specification, 3.00c, 2007. `http://www.hypertransport.org`.

[16] Ibrahim Hur and Calvin Lin. A comprehensive approach to dram power management. In *HPCA '08*, 2008.

[17] Intel 82541er gigabit ethernet controller. `http://download.intel.com`.

[18] Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler, and Tom W. Keller. Energy management for commercial servers. *Computer*, 36(12):39–48, 2003.

[19] Kevin Lim, Parthasarathy Ranganathan, Jichuan Chang, Chandrakant Patel, Trevor Mudge, and Steven Reinhardt. Understanding and designing new server architectures for emerging warehouse-computing environments. In *ISCA '08*, pages 315–326, Washington, DC, USA, 2008. IEEE Computer Society.

[20] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[21] Mellanox connectx ib specification sheet, 2008. `http://www.mellanox.com`.

[22] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: a portable "shared-memory" programming model for distributed memory computers. In *Supercomputing '94*, pages 340–349, New York, NY, USA, 1994. ACM.

[23] Quadrics qs ten g for hpc interconnect product family. 2008. `http://www.quadrics.com/`.

[24] Parthasarathy Ranganathan, Phil Leech, David Irwin, and Jeffrey Chase. Ensemble-level power management for dense blade servers. In *ISCA '06*, pages 66–77, Washington, DC, USA, 2006. IEEE Computer Society.

[25] David Slogsnat, Alexander Giese, Mondrian Nüssle, and Ulrich Brüning. An open-source hypertransport core. *ACM Trans. Reconfigurable Technol. Syst.*, 1(3):1–21, 2008.

[26] Storagereview.com drive performance resource center. 2008. `http://www.storagereview.com/`.

[27] Matthew E. Tolentino, Joseph Turner, and Kirk W. Cameron. Memory-miser: a performance-constrained runtime system for power-scalable clusters. In *CF '07*, pages 237–246, New York, NY, USA, 2007. ACM.

[28] Jeffrey Young, Sudhakar Yalamanchili, Federico Silla, and Jose Duato. A hypertransport-enabled global memory model for improved memory efficiency (tech report), 2008. `http://www.cercs.gatech.edu/tech-reports/index08.shtml`.

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

HyperTransport 3 Core: A Next Generation Host Interface with Extremely High Bandwidth

Benjamin Kalisch, Alexander Giese, Heiner Litz, Ulrich Brüning
University of Heidelberg
Computer Architecture Group
{benjamin.kalisch, alexander.giese, heiner.litz, ulrich.bruening}@ziti.uni-heidelberg.de

Abstract

As the amount of computing power keeps increasing, host interface bandwidth to memory and input-output devices (I/O) becomes a more and more limiting factor. High speed serial host interface protocols like PCI-Express and HyperTransport (HT) have been introduced to satisfy the applications' ever increasing demands for more bandwidth. Recent applications in the field of General Purpose Graphic Processing Units (GPGPUs) and Field Programmable Gate Array (FPGA) based coprocessors are an example. In this Paper we present a novel implementation of an FPGA based HyperTransport 3 (HT3) host interface. To the best of our knowledge it represents the very first implementation of this type. The design offers an extremely high unidirectional bandwidth of up to 2.3 GByte/s. It can be employed in arbitrary FPGA applications and then offers direct access to an AMD Opteron processor via the HT interface. To allow the development of an optimal design, we perform a complexity and requirements analysis. The result is our proposed solution which has been implemented in synthesizable Hardware Description Language (HDL) code. Microbenchmarks are presented to show the feasibility and high performance of the design.

1. Introduction

Following Moore's Law, computing power has doubled every 18 months over the last years. While scaling the operating frequency of high end processors has come to an end, exponential gains in computing power are still anticipated through the use of parallel processing. In either case, providing the processor with enough duty will require the increase of I/O bandwidth significantly. In fact, the processor - I/O bandwidth performance gap has increased in the last years [1] making it even more crucial to improve I/O performance.

This fact was first realized by AMD which replaced the outdated front size bus (FSB), that is used to interconnect the CPU, memory and I/O devices, with a novel packet based point-to-point interconnect called HyperTransport (HT) in their Opteron processors. HT offers high bandwidth, extremely low latency [2] and can support cache coherency which makes it ideally suited for communication between CPUs, memory and IO. The high performance of the HT interconnect is the main reason for the much better scalability of Opteron based symmetric multiprocessor (SMP) with non unified memory architecture (NUMA) machines in comparison to Intel Xeon based SMPs [3].

HT supports variable link widths and up to 2 Gbit/s on each lane in protocol version 2.x, also referred to as Gen1. This leads to a maximum unidirectional bandwidth of 4 GByte/s for a 16 bit link. Apart from the Opteron CPUs, HT 2.x is successfully implemented by peripheral hardware devices like the Pathscale network interface [4], Cray's Seastar [5] and the Field Programmable Gate Array (FPGA) based rapid prototyping board [6]. HT devices can directly communicate with the processors without any intermediate bridges using the HyperTransport Extension (HTX) connector [7]. HTX is a PCI-Express slot like standard defined by the HyperTransport Consortium (HTC).

Recently, the HTC introduced HT 3.1, also referred to as Gen3, which increases the supported speeds to 6.4 Gbit/s on a lane equalling a theoretical unidirectional peak bandwidth of 12.8 GByte/s for a 16 bit link. The first integrated circuits (ICs) that will support HT 3.x are the Shanghai Opteron processors, however, no non Opteron implementations are currently available. The reason for this is, that currently no HTX3 capable mainboards are available and the lack of an open source HT3-Core like the HT2-Core [8]. To solve the latter issue, in this paper we present the very first high performance HT3-Core for FPGA implementations. The core provides very high bandwidth even for FPGA implementations, and therefore presents the ideal building block for high performance next generation I/O devices. Our solution promises to deliver a bidirectional bandwidth of up to 9.2 GByte/s for a 16 bit link. To the best of our knowledge this makes it the fastest host interface implementation currently available for FPGAs.

The rest of the paper is organized as follows. Section 2 will provide background information and define the requirements for an FPGA based HT3 core. Section 3 will present a complexity analysis and describe the challenges of such an implementation. Our proposed architecture is presented in Section 4. It is followed by an evaluation in Section 5 and we draw a conclusion in Section 6.

2. Background

To define the requirements of an HT3 core a short introduction to the HT protocol will be given. The HT specification defines the entire protocol stack ranging from the physical layer up to the transaction level layer. The physical layer defines the electrical parameters which have to be adhered by HT device implementations and include jitter, slew rate and common mode characteristics. Physical layer compliance is already provided by the physical layer device (PHY) and therefore out of scope of this paper. The PHY also takes care of serialization/deserialization (SERDES) of the high-speed serial data stream. For signalling HT defines 2, 4, 8, 16 and 32 bit command-address-data (CAD) busses which are accompanied by a set of control (CTL) lanes and clock (CLK) lanes. Most common are 8 or 16 bit configurations, whereby multiples of 8 CAD lanes, one CTL and one CLK lane are considered as a link. A link connects exactly two endpoints whereas switches have to be employed to realize topologies of multiple endpoints.

The transaction layer defines the packets which are transmitted over HT links. A transaction consists of a command packet and an optional data packet carrying 1-16 doublewords (32 bit) which allows to send maximum sized transactions of 64 Byte. This size is equivalent to a cacheline on current x86 systems. In Gen3 mode each transaction is also appended with a CRC. The specification defines a large number of commands with the main purpose of data movement. Therefore, write, read and response operations are defined. To avoid deadlocks, which may be caused by cyclic dependencies from split phase transactions, the different commands are assigned to different virtual channels (VC). This allows reordering of the data stream and breaking up cyclic dependencies. Furthermore, the commands implement low level functionality like flow control and fault tolerance. The required functionality which has to be provided by an HT3-Core implementation is therefore as follows:

- Packetization: Extracts Transactions from the data stream and sorts them into their according virtual channel queue and vice versa.
- Flow control: HT defines a credit based flow control which has to be supported by the core.
- Fault tolerance: HT3 defines an advanced CRC mechanisms for increased reliability
- Scrambling: To support clock data recovery in Gen3 mode, the data stream is scrambled.
- Low level initialization methods

As the HT2-Core presented in [8] implements the same functionality according to the HT2 specification and HT3 is downwards compatible, it is reasonable to analyze, whether a modified HT2-Core would be a sufficient solution. Therefore, it is useful to examine the novel features which have to be supported by Gen3 devices. The most important modifications are the increased frequency support of up to 3.2 GHz and the enhanced fault tolerance mechanic. Additionally to the periodic CRC, which can be used to detect, but not to correct errors, HT3 introduces a retry mechanism with per-transaction CRC. Every transaction sent out by the transmitter is appended with a CRC and stored into a retry buffer. On reception the receiver calculates the CRC again and in the case of a successful match sends an acknowledge back. In the case of a mismatch a nack packet is generated which leads to a retransmission of the original transaction. Implementation of the retry mechanism requires heavy modification of the HT2-Core. Even more significant, however, is the increased bandwidth that has to be supported internally. The HT2-Core supports an internal data width of 64 bit which requires an internal core clock frequency of 600-1600 MHz for a 16 bit link at Gen3 frequencies, and a frequency of 300-800 MHz for a 8 bit link.

Last but not least is the introduction of a new data sampling scheme for Gen3 devices. Instead of the source synchronous mechanism sampling incoming data with the link clock, HT3 devices use a clock data recovery (CDR) technique. The CDR circuit recovers a dedicated clock for each lane and uses it to sample the data. As static data patterns occurring in IDLE phases prohibit reliable clock recovery, a data scrambling mechanism is used in Gen3 mode.

The required change to a 128 bit internal interface and the additional modifications regarding retry mode demand for a complete redesign of the HT2-Core.

3. Complexity analysis

As mentioned before HT3 is a packet-based point to point interconnect, which operates on a minimum packet size of 32 bit, one doubleword (DW). To support the provided bandwidth of HT3 an analysis of the data stream is required.

The data stream of HT3 can be distinguished into three different DW types which are command (CMD), data (DATA), and a cyclic redundancy check (CRC) checksum. To keep the decoding of the data stream as simple as possible, an internal data width of 32 bit would be ideal, so every clock cycle one of only three different types of DWs must be interpreted. To support higher bandwidth on the HT link, the data stream has to be parallelized which leads to wider internal data buses, as the maximum frequency is the limiting factor in FPGAs. Due to the fact that the HT3 protocol does not allow all combinations of different DW types, the complexity does not increase quadratically, but as can be seen in Figure 1, the increase in complexity is significant. Multiple consecutive command DWs may belong to two separate command packets without a separating CRC due to command packet insertion. The number of combinations for a 256 bit wide data path is not depicted due to the large number of possible combinations.

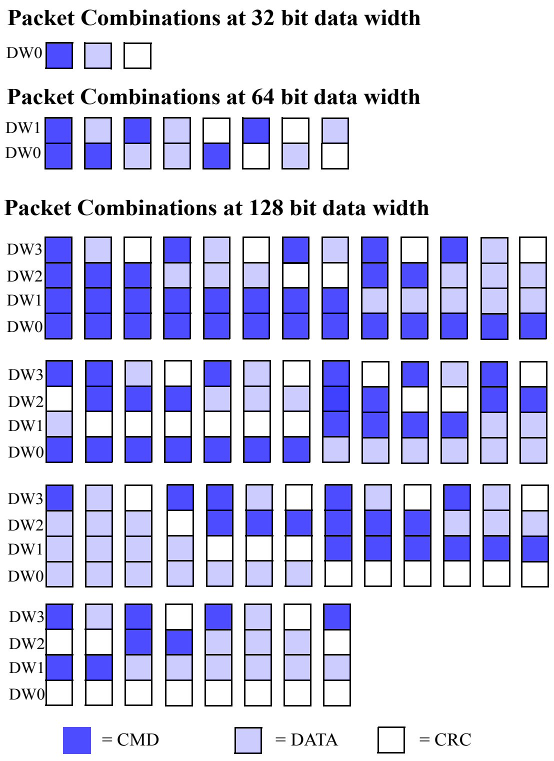

Packet Combinations at 32 bit data width
DW0
Packet Combinations at 64 bit data width
DW1
DW0
Packet Combinations at 128 bit data width
DW3
DW2
DW1
DW0
= CMD
= DATA
= CRC

Figure 1: Complexity growth

HT3 has a minimum link frequency requirement of 1.2 GHz. Depending on link width and parallelization degree this results in different possible core frequencies shown in Table 1.

Table 1: Internal clock frequencies

At 1.2GHz link frequency		External Link Width	
		8bit	16bit
Internal Link Width	32bit	600MHz	1200MHz
	64bit	300MHz	600MHz
	128bit	150MHz	300MHz
	256bit	75MHz	150MHz

The target device is a state of the art Virtex 5 FPGA which can be clocked at a theoretical maximum frequency of 550 MHz for the core logic. For a design that contains complex logic as the HT3-Core a core frequency of 300 MHz is difficult to achieve but possible. This reduces the possible combinations of link width and parallelization degree that can be realized.

In the HT3-Core design an internal data width of 128 bit is implemented, as it provides the best combination between feasible core frequency and logic complexity. The core logic mainly consists of multiplexing structures which sort the DWs to form complete packets. Analysis of these multiplexing structures has shown that two different factors influence the reachable frequencies of such multiplexers. One is the number of the input bits of the multiplexer, the other is the number of control signals of the multiplexers. Figure 2 shows that increasing the multiplexer width reduces the achievable core frequency significantly. Doubling the width from two to four doublewords reduces the operating frequency by almost 100 MHz.

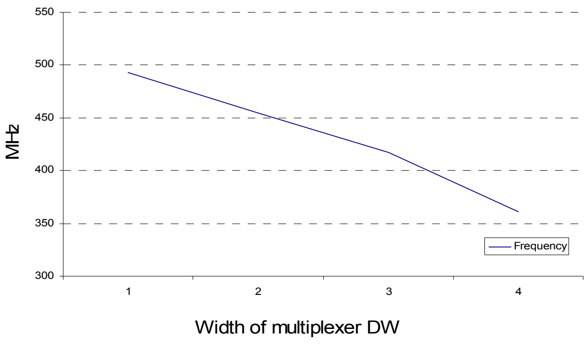


Figure 2: Multiplexer width influence

Increasing the number of control signals of the multiplexer also reduces the maximum operating frequency. This is shown in Figure 3, where all other parameters besides the control signals remain unmodified.

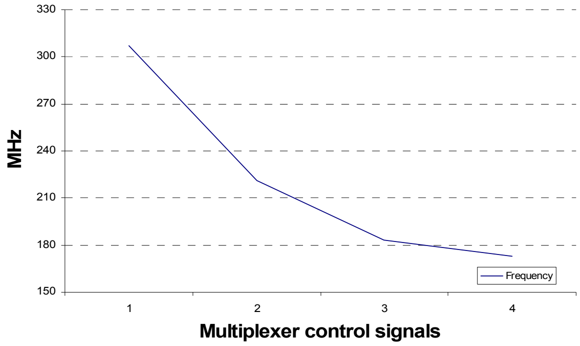


Figure 3: Multiplexer control signal influence

To handle all different traffic types, a multiplexer width of 128 bit which selects between two sets of seven DW wide registers is needed. The input and output path is four DWs wide, and if only one DW can be forwarded three DWs must be stored. If the multiplexer width would be increased to 256 bit the above explained factors for frequency decrease would take effect. Obviously, the input width would have to be doubled and also the number of control inputs would have to increase, as the decoding complexity increases due to different cases that have to be handled. These two combined factors result in a much larger amount and deeper hierarchy of multiplexers inside the design. Thereby the routing of the control signals to all multiplexers becomes so difficult that routing delay and fanout get extremely high and reduces the reachable frequency. This reduction outweighs the advantage gained through doubling the data path, which is a reduction of the necessary core frequency by a factor of two.

These results show that an internal data width of 64 bit is not sufficient to reach a clock frequency which is feasible in today FPGAs. A multiplexer width of 256 bit increases the complexity of the logic nearly quadratically, which is a point where no advantage of the lower internal frequencies can be achieved due to the routing overhead. Therefore a multiplexer width of 128 bit was chosen for the design.

4. Proposed architecture

Due to the addition of a retry mode implementation for HT3 devices, as well as the increased internal data path width, a new architecture has been created to fulfill these needs. The increased data path width, necessary to handle the complexity, also results in an increased pipeline depth to reach timing closure.

Due to the nature of the HT protocol, it is necessary to support Gen1 operation as well as Gen3 operation. As the goal of the architecture is to operate in HT3 mode, Gen1 operation is only intended for configuration access following cold reset, to transition the controller into Gen3 operation.

The controller can be separated into two functional main components. One is handling the reception of incoming traffic (RX), while the other is responsible for creating and transmitting an outbound transaction stream (TX). These two entities largely operate independently from one another. Only the exchange of flow control credits links both components. An overview is given in Figure 4.

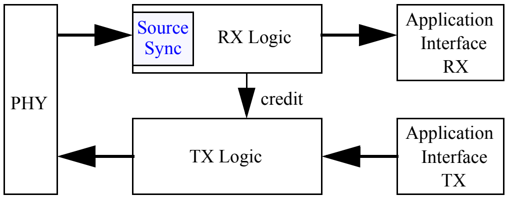


Figure 4: Top-level HT3-Core overview

The application interfaces consist of a number of traffic buffers, and support fully asynchronous clocking. This enables the application to run at an arbitrary frequency, independent of the link frequency. The interfaces contain separate command and data packet buffers for each VC. All contents of these buffers start at naturally aligned borders, whereby command packets are reordered to gain a continuous address field for transactions with address extension (64 bit addresses).

The PHY operates with a deserialization factor of 8. So for a 16 bit link, this results in 128 bits of CAD and 16 bits of CTL information each cycle. The core always operates on the same amount of data, independent of link width. This means that an 8 bit link only requires half the internal frequency of a 16 bit link.

The RX side architecture imposes no restrictions of command throttling, and permits command insertion. The TX architecture is more restrictive. Command insertion is not performed, and the number of command packets in each octaword is limited to one each cycle. If data transactions travel in the same VC, they can be streamed back-to-back.

RX and TX paths will be discussed in more detail in paragraphs 4.1 and 4.2 respectively. Two additional paragraphs highlight some of the more interesting implementation details. Paragraph 4.3 details the implementation of the user application interface, and paragraph 4.4 describes the CRC implementation.

4.1 RX path

The RX path reorders and decodes the incoming transaction stream, so that it can forward octaword aligned command and data packets. Such an alignment is rarely given in the data stream itself. It is further responsible for handling some low-level signaling (initialization), and must handle no-operation (NOP) packets.

All RX functionality can be divided into five major functional blocks, shown in Figure 5, and further described below. Each block contains multiple pipeline stages.

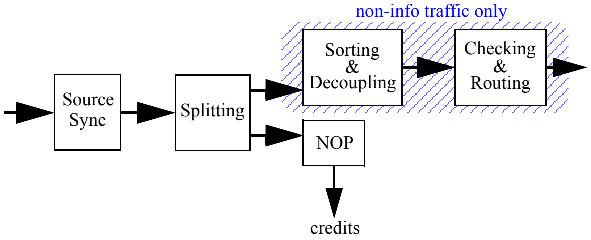


Figure 5: Functional pipeline of the RX path

The *Source Sync block* operates at the frequency of the recovered link clock. Its functionality includes handling the Gen1 initialization and the Gen3 training sequences. These sequences are used to communicate the start of the first DW between two participants in an HT chain. Due to the deserialization factor of 8, it might further be necessary to align the 8 bit received from each lane to reflect the DW alignment. During Gen3 operation the individual lanes are deskewed to return the same link bit-time, and the data stream is also descrambled. The last function this block fulfills is to check the periodic CRC DW and remove it from the data stream. The data stream is then stored in an asynchronously clocked FIFO to leave the source synchronous clock domain.

The *Splitting block* separates the incoming transaction stream into info and non-info traffic. Info traffic refers to NOPs and credits, whereas non-info traffic consists of all other transactions. This is necessary, as the following block buffers the non-info transactions. Buffering is enabled by the fact that VC traffic is flow controlled and therefore limited, whereas info traffic is not limited. Info packets are handled in the NOP block in parallel to the non-info transaction processing.

The *NOP block* evaluates received info packets. This includes extraction of flow control credits, as well as evaluation of other info packet fields, such as LDT-STOP/retry indication. During Gen3 operation the per-transaction CRC of the NOP packets is also checked and the acknowledge count included in the NOP is used to remove the acknowledged transactions from the retry logic.

The *Sorting and Decoupling stage* contains three major blocks. The first separates the incoming transaction stream into the basic transaction building blocks, which are command packets, data packets, and CRC packets. These packets are then stored in independent buffers. This buffering allows the remaining controller logic to operate at reduced bandwidth in cases where the input stream contains more than one command packet every octaword. A worst-case maximum of three command packets can be located inside of an octaword. The logic can compensate the reduced bandwidth if data transactions are processed. This is possible as commands and data, of up-to 128 bit size each, are processed simultaneously.

The *Checking and Routing block* of the RX path implements forwarding of the decoded transactions to their corresponding VC buffer within the application interface. During Gen3 operation, it calculates the per-transactions CRC of the forwarded transaction and indicates a successful check to the VC buffers. The VC buffers are implemented in such a way that a stored transaction only becomes visible to the application after it has been validated.

4.2 TX path

The TX path creates a HT compliant transaction stream from the command and data packets provided by the application interface. If no transactions are available in the user buffers, NOP packets are transmitted. The retry functionality required for Gen3 operation is not implemented with an explicit retry buffer, but reuses the TX application interface buffers to reduce complexity and area in terms of SRAM.

TX functionality can be divided into four major functional blocks, shown in Figure 6, which are further described below. Each TX block contains multiple pipeline stages.

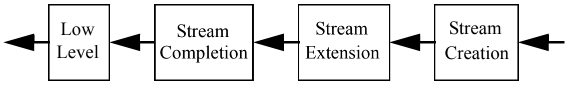


Figure 6: Functional pipeline of the TX path

The *Stream Creation block* merges the command and data packets from the application interface buffers and creates a rudimentary transaction stream for each VC, which is limited to one command packet per octaword. These streams do not yet contain the per-transaction CRC or any info packets. The VC transaction-streams are multiplexed into a single stream via round robin arbitration. The order in which transactions from different VCs are transmitted is tracked as well. This allows to correctly assign received acknowledges to the corresponding VCs. The arbitrated transaction stream is stored into a decoupling buffer to ease implementation of backward flow control between the complex pipeline stages.

The *Stream Extension block* adapts the transactions retrieved from the decoupling buffer to the actual link width. During this adjustment, it also appends a per-transaction CRC placeholder after each transaction, independent of the operation mode, and is filling possible gaps between transactions with NOP placeholders.

The *Stream Completion block* is filling the placeholders inserted by the previous block with the required information. This means that it is computing the per-transaction CRC during Gen3 operation and inserting it into the CRC placeholder. NOP placeholders are filled with valid information, including the release of flow-control credits. During Gen1 operation all CRC placeholders are replaced with empty NOP packets, as this helps reduce the amount of necessary complexity for Gen1 operation in prior pipeline stages.

The *Low Level stage* implements a multiplexer between the assembled transaction stream and special low-level signaling schemes. The low-level signaling includes Gen1 initialization, Gen3 training, as well as sync-flooding. During Gen3 operation the transaction stream is also scrambled in this block, before it is forwarded to the PHY.

4.3 Application interface buffers

During Gen3 operation the core must support the retry mode defined by HT. This retry mode secures every transaction with a per-transaction CRC, and introduces two requirements to the architecture:

a) Received transactions are only forwarded after their CRC has been successfully verified

b) Transmitted transactions, barring info traffic, must be stored to allow a retransmission

Point *a)* is solved by the use of a special buffer that stores unverified transactions until their CRC has been checked. Entries in the buffer only become visible to the application after the CRC check was successful. This allows the decoding of the transaction to continue concurrently with the verification of the transaction's CRC, as even unverified transactions can already be added to the buffer and get validated later on. It furthermore, reduces area as transactions do not have to be intermittently stored in registers, but can be forwarded to the buffers immediately. To keep the interface simple and easy to use, while still fulfilling all needs of the retry mode, the RX application buffers are implemented as FIFO buffers with an additional *validated* write pointer (*ValWr*). Stored values are written to the write pointer (*Wr*) address, whereas the output is read from the read pointer (*Rd*) address. The operation of this FIFO is illustrated in Figure 7. Entries located between the *ValWr* pointer and the write pointer are unvalidated (shaded dark) and not visible to the application. Entries located between the read and the *ValWr* pointer are validated entries that the application can access through the defined mechanism. If a retry is executed, the write pointer will be set to the current *ValWr* pointer address, thereby removing all unvalidated entries.

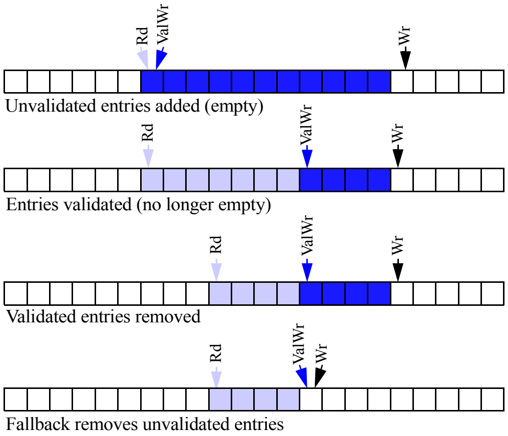

Rd
ValWr
Wr
Unvalidated entries added (empty)
Rd
ValWr
Wr
Entries validated (no longer empty)
Rd
ValWr
Wr
Validated entries removed
Rd
ValWr
Wr
Fallback removes unvalidated entries

Figure 7: RX application buffer operation

A solution to issue *b)* would be the addition of a retry buffer that stores all transmitted non-info transactions. Our proposed implementation avoids this additional retry buffer by reusing the already existing TX application buffers. This is possible as HT makes no assumption about the order in which unacknowledged transactions are replayed in case of an error.

The TX application buffers are implemented as FIFO buffers with a second *unacknowledged* read pointer (*URd*). Whenever a transaction gets acknowledged by the remote device, the *URd* pointer is incremented and thereby the addressed transaction is effectively removed from the retry buffer. Figure 8 illustrates the operation of this FIFO. All transactions located between the *URd* pointer and the read pointer resemble the retry buffer, as they are unacknowledged (shaded dark). Entries located between the read and the write pointer resemble the application interface buffer with transactions that still have to be transmitted (lightly shaded). During a retry, the read pointer is reset to the current value of the *URd* pointer. As the VC multiplexing in TX is done behind the application buffers, this means that retried transactions are not sent in the same order they were initially sent.

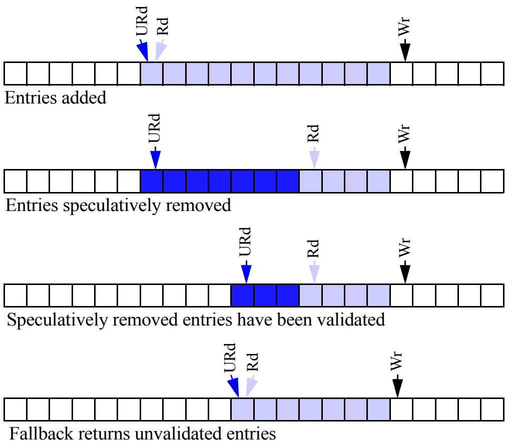

URd
Rd
Wr
Entries added
URd
Rd
Wr
Entries speculatively removed
URd
Rd
Wr
Speculatively removed entries have been validated
URd
Rd
Wr
Fallback returns unvalidated entries

Figure 8: TX application buffer operation

To maintain the simple FIFO interface, both additional pointers can be incremented in single steps via an additional input signal to the buffers. Incrementing of the additional pointers is done after a successful packet-CRC check for RX, and the additional TX pointer is incremented whenever a new acknowledge counter is received from the remote device.

4.4 CRC calculation of non-info transactions

Calculation of the 32 bit per-transaction CRCs used for the retry mode during Gen3 operation is dependent on the degree of used data parallelism. The CRC calculation is commonly implemented as linear feedback shift registers (LFSR) for the polynomial division. Figure 9 depicts an example of a CRC calculation implemented as LFSR, where in each cycle one bit of data is serially added to the checksum. The calculation shown is based on the polynomial x^4+x^2+x+1.

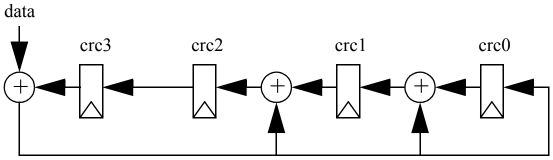


Figure 9: CRC LFSR example

The operation performed by the LFSR can be expressed through the following formulas, where t identifies time (cycles).

$$\begin{aligned} crc3_{t+1} &= crc2_t \\ crc2_{t+1} &= crc1_t + crc3_t + data_t \\ crc1_{t+1} &= crc0_t + crc3_t + data_t \\ crc0_{t+1} &= crc3_t + data_t \end{aligned}$$

These formulas describe the addition of one bit of data to a checksum. It can also be seen that each new result directly depends upon the previous cycle's result. More practical formulas, describing how multiple bits are added to a checksum in parallel, can be produced by recursively iterating these formulas.

The complexity of the formulas increases with more data to be included into the calculation. They can always be expressed as XOR combinations of the input data and the state of the CRC register from the previous cycle. This determines the maximum number of parameters one formula can include to be $parameter\_limit = CRC\_size + data\_size$ and the worst case number of necessary XOR operations is $xor\_limit = parameter\_limit - 1$. So each formula grows linear with both CRC and data size. All formulas together grow quadratically with the CRC size and linear with the data size, because there are as many formulas as there are bits in the CRC.

In HT, the minimum transaction unit (mTU) is one DW of CAD plus 4 bit of CTL, which are all covered by the per-transaction CRC. The maximum size of an HT transaction that is supported by the proposed architecture is 19 mTUs, excluding the per-transaction CRC. Such a transaction contains a 3 mTU command packet with address extension plus a 16 mTU data packet. Any received HT transaction can have an arbitrary size ranging from 1 to 19 mTUs. The CRC calculation must be capable of calculating the per-transaction CRC for all possible transaction sizes. For the given architecture operating on four mTUs per cycle, this means that in each clock cycle 0 to 4 mTUs may be added to the calculation of one per-transaction CRC. This results in four sets of different CRC formulas, for adding 1 to 4 mTUs that all operate on the same 32 bit CRC register.

A formula f used to calculate one bit of a CRC for one parallel data input combination can be divided into a recursive function g and a non-recursive function h.

$$crc_{t+1} = f(crc_t, data) = g(crc_t) + h(data)$$

This reduces the impact of the cycle-to-cycle dependency on the CRC calculation and relaxes timing, as functions g and h can be implemented in different pipeline stages. This is especially attractive for large amounts of data, as function h can be further pipelined. Function g however contains the cycle-to-cycle dependency of the CRC calculation and cannot directly be pipelined further, which also means that it includes the critical timing path.

This approach was used to implement the CRC calculation of non-info transactions in the proposed architecture. As the architecture must handle 1 to 4 mTUs, four different g and h formulas exist for each CRC bit. These are multiplexed in the last pipeline stage which then contains all recursive logic. Figure 10 gives an overview of the CRC calculation pipelining.

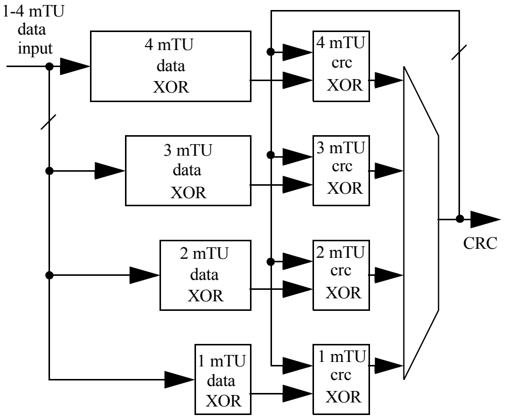


Figure 10: CRC pipelining overview

5. Evaluation

The bandwidth results shown in this paragraph were gathered from simulation of the synthesizable HDL description of the core. An implementation of the proposed architecture, using an 8 bit link operating at 2.4 Gbit/s. The bandwidth was measured through the transmission of 2,000 write transactions which introduces 3 DWs of overhead to the data payload. One DW per-transaction CRC and two DWs command packet containing a 40 bit address. The simulation have been repeated for all sizes of data packets to show the overall performance of the core.

Figures 11 and 12 show the measurement results of the RX and TX path. Both paths were simulated separately to avoid performance influences. In these figures Transaction Bandwidth refers to the bandwidth being used for non-info transactions, including command packets, data packets and per-transaction CRCs, calculated as *(((DATA DW + CMD DW + CRC DW) * 2000 * 4 * (1 / time)) / 10^9) GByte/s*. Transaction Bandwidth without CRC does not count the CRCs, calculated as *(((DATA DW + CMD DW) * 2000 *4 * (1 / time)) / 10^9) GByte/s*. Lastly Payload Bandwidth shows the effective bandwidth that is used for data forwarding, excluding command packets and CRCs, calculated as *((DATA DW * 2000 * 4 * (1 / time)) / 10^9) GByte/s.*

For sufficient payload sizes the architecture reaches a total bandwidth of 2.38 GByte/s. The periodic CRC slot accounts for a bandwidth loss of about 0.775% or 0,0186 GByte/s on an 8 bit link as it is recommended by HT so every 512 bit times a 4 bit CRC has to be transmitted. If this bandwidth loss is added to the Transaction Bandwidth it results in a total utilized bandwidth of 2.3986 GByte/s which is extremely close to the theoretical maximum of 2.4 GByte/s for an 8 bit link.

Due to the increased number of pipeline stages the bandwidth drops with lower data payloads. This happens because of credit starvation. Then all available credits can be in use. But if high bandwidth is required, sending smaller data payloads is counterproductive because the further command overhead will decrease the usable bandwidth additionally.

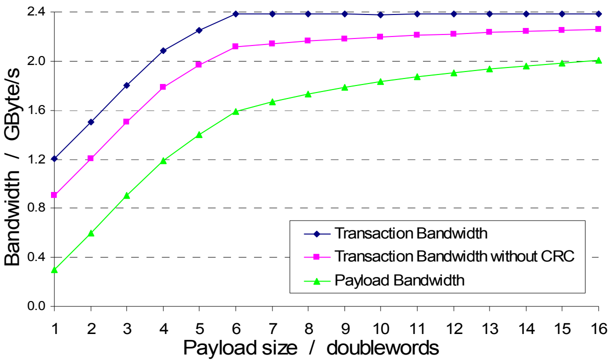


Figure 11: RX bandwidth for data transfers

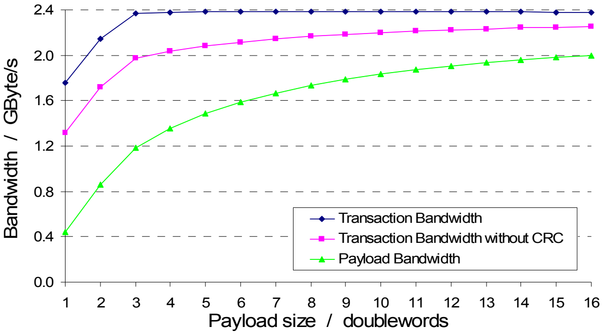


Figure 12: TX bandwidth for data transfers

Due to the increased datapath of the core and the fact that this increase also adds complexity, the resource usage of the HT3-Core is higher than it was for the HT2-Core. Table 2 shows the total and percentage resource usage of the core on a Xilinx Virtex-5 LX110T FPGA device.

Table 2: Resource usage on LX110T

Resource	Used	Percentage
Slice Registers	18,905	27%
Slice LUTs	37,094	53%
Occupied Slices	11,098	64%
Block RAMs	78	52%

6. Conclusion and outlook

We have proposed a novel architecture of an HT3 controller for FPGAs. To the best of our knowledge this is the first implementation for such reconfigurable platforms. We have performed a complexity analysis and shown the issues of modern high bandwidth I/O technologies like HT. Solutions for these problems and a very efficient implementation of the core is provided. The benchmarks show the excellent performance of the architecture with maximum achievable bandwidth of 2.3 GByte/s, which is close to the theoretical optimum.

Our future work will focus on the bringup of the design in real world systems using various FPGA technologies. The architecture will also be further improved as more real world data can be gathered. A 16 bit link version, currently in development, promises to double the achievable bandwidth to 4.6 GByte/s. There are also plans to push the 8 bit link version beyond the current lane rate of 2.4 Gbit/s.

7. References

[1] Mahapatra, N. R. and Venkatrao, B.: The processor-memory bottleneck: problems and solutions. Proc. Crossroads 5, 3, 1999

[2] Bees, D. and Holden B. 2004. HyperTransport reduces delays in some applications.

[3] C. Guiang, K. Milfeld, A. Purkayastha, J. Boisseau. "Memory Performance on Dual-Processor Nodes: Comparison of Intel Xeon and AMD Opteron Memory Subsystem Architectures," Proceedings of the ClusterWorld Conference and Expo, San Jose, CA, June 24-26, 2003.

[4] R. Brightwell, D. Doerfler, K.D. Underwood, "A preliminary analysis of the InfiniPath and XD1 network interfaces," ipdps,pp.311, Proceedings 20th IEEE International Parallel & Distributed Processing Symposium, 2006

[5] Ron Brightwell, Kevin T. Pedretti, Keith D. Underwood, Trammell Hudson, "SeaStar Interconnect: Balanced Bandwidth for Scalable Performance," IEEE Micro, vol. 26, no. 3, pp. 41-57, May/Jun, 2006

[6] H. Fröning, M. Nüssle, D. Slogsnat, H. Litz, U. Brüning: The HTX-Board: A Rapid Prototyping Station. Proc. of 3rd annual FPGAworld Conference, Nov. 16, 2006, Stockholm, Sweden.

[7] HyperTransport Consortium: The Future of High-Performance Computing: Direct Low Latency CPU-to-Subsystem Interconnect. HTC whitepaper, 2008

[8] David Slogsnat, Alexander Giese, Mondrian Nüssle, Ulrich Brüning: An Open-Source HyperTransport Core. ACM Transactions on Reconfigurable Technology and Systems (TRETS), Vol. 1, Issue 3, p. 1-21, Sept. 2008.

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

Exploiting the HTX-Board as a Coprocessor for Exact Arithmetics

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

Abstract

Certain numerical computations benefit from dedicated computation units, e.g. providing increased computation accuracy. Exploiting current interconnection technologies and advances in reconfigurable logic, restrictions and drawbacks of past approaches towards application-specific units can be overcome. This paper presents our implementation of an FPGA-based hardware unit for exact arithmetics. The unit is tightly integrated into the host system using state-of-the-art HyperTransport technology. An according runtime system provides OS-level support including dynamic function resolution. The approach demonstrates suitability and applicability of the chosen technologies, setting the pace towards broadly acceptable use of reconfigurable coprocessor technology for application-specific computing.

1 Introduction

One major requirement of certain numerical applications is computation accuracy. Depending on the application, using standard floating-point implementation does not provide necessary resolution, especially due to accumulating rounding errors. This may lead to more required iterations, thus degrading the overall application performance. The solution to such problems is the use of dedicated precise arithmetic routines, ideally provided as a dedicated application-specific hardware accelerator.

Such approaches were already targeted in the past, but were typically hampered by insufficient hardware resources and, most notably, the used interconnection technology: peripheral buses usually do not provide significant bandwidth to enable a tight cooperation between an application running on the host CPU and the according hardware accelerator.

With the advances in programmable logic and interconnection technology, using so-called FPGAs as application-specific computation devices tightly integrated into the host system via current interconnection technology such as HyperTransport delivers a promising concept for arbitrary coprocessor solutions. Suiting the programmers' needs, such a solution should be easily accessible, not requiring a new programming model or dedicated, heavyweight execution environments.

We therefore chose the original work of Kulisch et al. for exact arithmetics [21], which delivered impressive numbers on raw computation throughput. Unfortunately, the back then available state-of-the-art interconnection bus turned out to be severely limiting. Using current HyperTransport and FPGA technology, we adopted the original design principle. Based on the UoH HTX-Board [2], a dedicated unit supporting exact arithmetics was developed. We furthermore provide a runtime system supporting dynamic resolution and mapping of function calls that enables seamless switching between standard and exact computation. Such a runtime system not only fulfils the requirements for advanced, parallel arithmetics as mentioned before, but also for irregularly memory-demanding applications such as numerical applications on adaptive meshes and graph analysis problems [14] as it allows redirecting resource accesses, which is substantial to tackling these issues.

In the following section, we will give an overview over the field of dedicated application accelerators and their integration into the host system, outlining the individual peculiarities of each approach. Section 3 then introduces the concept of exact arithmetics as proposed by Kulisch, and its implementation on the UoH HTX-Board using the Open-Source HyperTransport Core [27]. An overview of our runtime system and its general usage is given in Section 4. Our experimental setup is presented in Section 5, its results are shown in Section 6. Finally, we give an outlook over future work in Section 7 after drawing some conclusions.

2 Related Work

A significant amount of research on the arithmetic part, dynamically reconfigurable application accelerators, and according runtime systems and programming models exists. For this paper, we will therefore restrict to some focal points, outlining the intended problem solution and eventual drawbacks.

Targeting the domain of error estimation, interval arithmetic was developed in the 1950s and 1960s [23] and is still actively researched today [25, 8].

Several approaches targeting hardware implementations of more exact arithmetics exist, ranging from multi-precision fixed-point vector MAC [29] to quad-precision floating-point units [11] or fixed-point computations. A major problem with floating-point operations is the accumulation of rounding errors: for this, the extension of the radix avoiding time-consuming rounding operations is a potential solution [26]. Using reconfigurable logic lays the foundation for arbitrary-precision arithmetic units on FPGAs for rational numbers [7]. While the cited approach lacks floating-point support, it demonstrates feasibility and achievable speed-up in comparison to software emulation by the GMP library [12].

As of now, a vast number of also commercially available hardware accelerators exists for different goals, among them acceleration of computation and algorithms on dedicated hardware. The ClearSpeed series of accelerators [5] might serve as an example of current state-of-the-art accelerator architectures, providing 96 floating-point processing cores. An even bigger number of processing cores is provided by current graphics cards being used as floating-point accelerators, also employing a multi-level memory hierarchy to overcome bus bandwidth limitations resulting from the used PCI express bus (PCIe).

All these hardware approaches have in common that interconnection bandwidth and therefore data transport between host system and accelerator becomes a bottleneck. This makes the use of such units only sensible when the gained speedup outweighs transfer time.

In order to avoid transport delay issues when tackling precision, the available precision can be extended inside floating-point units: in the IBM P6 [30], this technique is well-employed with buffering of intermediate results and providing further rounding operations in addition to those specified in the 754 standard. Still, precision is not enough for numerically unstable algorithms.

Approaches such as IRAM [24] and PIM [28] target the problem of data transport and transport latencies directly by calculating inside memory, what requires their own programming models as well. Both exact arithmetic and interval arithmetic are also available as software libraries for a large number of programming environments and systems, even bindings for high-level programming languages are available [20, 12, 9].

Reconfigurable logic offers the possibility of providing required arithmetic operations as demanded by the computation, therefore offering the same flexibility as software libraries but at significantly higher speed. With the advent of high-performance interconnection buses such as HyperTransport [16], this approach has gained widespread acceptance ranging from reconfigurable accelerator cards like Nallatech's FPGA Computing platforms [17] to reconfigurable supercomputing systems like Cray XD-1 [6]. On these accelerator cards, the available FPGAs are big enough to hold a couple of acceleration units.

Our approach follows the reconfigurability concept, making use of reconfigurable FPGA hardware to model a dedicated, high-performance acceleration unit focusing on exact arithmetics. To avoid limitations from communication bottlenecks, we employ HyperTransport as state-of-the-art interconnection technology. The case study for our approach is Kulisch et al.'s first implementation of a hardware unit for exact arithmetic. As mentioned before, this work suffered from bandwidth limitations of the PCI bus while the unit was sufficiently fast [18].

The use of application-specific and potentially reconfigurable hardware from within the application is targeted by several approaches ranging from description languages to integrated environments, consisting of dedicated compilers and runtime systems. The scope ranges from programming languages such as Handel-C [4], abstraction via ISA extensions such as MOLEN [31] and EXOCHI [32], and runtime environments such as LIME [15], and combined API/Runtime systems like the recent OpenCL [13].

The above solutions typically require either additional software layers for accessing accelerator hardware, are focused on a dedicated system setup, or both. What is missing from these approaches is an easy and lightweight method to dynamically resolve function calls at runtime so that a different implementation or a different library be used for the same function call. This is very desirable, as it offers the possibility to hide transport and computation latency by switching to different implementations as long as the hardware is occupied, and it is also important for dynamic systems where hardware resources are allocated and freed at runtime for the best mapping of an application onto the respective hardware, enabling more precise, faster or less power-consuming execution of the application.

We therefore developed a lightweight extension to the Linux OS's runtime system providing a method for dynamic control of function mapping as a convenient means to change between different arithmetic implementations as required by the running computation.

In the following, we will present the implementation and integration of the accelerator into HT-equipped systems, focusing on the hardware design, and will include software and runtime implications where appropriate.

3 Exact Arithmetics

As basic design of an exact accumulator, we implemented the approach proposed by Kulisch et al. [21]. The idea behind their approach is to avoid rounding results as much as possible, because rounding leads to accumulation of rounding errors after a couple of computations in conventional FPUs, such as adding small values multiple times to a rather big value. For double precision, the computation window is much larger than for single precision operation, but breaks with three more orders of magnitude as well. Such computation schemes however, are common to a wide range of numerical applications.

As already mentioned, their implementation in CMOS technology was fast enough, but did not keep pace with advances in processor technology as the interconnection relied on the PCI bus.

3.1 Concept

In order to avoid rounding, a different presentation than the conventional one consisting of the well-known sign-characteristic-mantissa encoding where numbers are composed like

$$number = 2^{exponent} * 1.mantissa \tag{1}$$

with m the length of the characteristics field and n the length of the mantissa field and $exponent = characteristic - 2^{m-1} + 1$ (i.e. the characteristic is the exponent plus the required bias), is needed for intermediate representation of the values, because with the exponent representation, only a “window” of a float number is precise.

The flat two’s complement encoding suits very well as rational numbers can be stored in fix-comma representation easily enabling both very large and very small numbers to be represented at once:

$$number = whole\text{-}number.fraction \tag{2}$$

with a length of k and l respectively where $whole\text{-}number$ ranges from -2^{k-1} to $2^{k-1} - 1$ and $fraction$ is in between 2^{-l} and $\sum_{i=1}^{l} 2^{-i}$ for both positive and negative numbers as well as zero is possible. Considering IEEE 754 single-precision format with $m = 8$ and $n = 23$, $0x00000001 = 2^{-2^{m-1}+2-n} = 2^{-149}$ is the smallest representable number; whereas the largest number is $0x7FFFFFFF = 2^{2^{m-1}} - 2^{-n} < 2^{128}$. Thus, for accumulating these precisely, the following inequation must be held:

$$k >= 128; l >= 149 \tag{3}$$

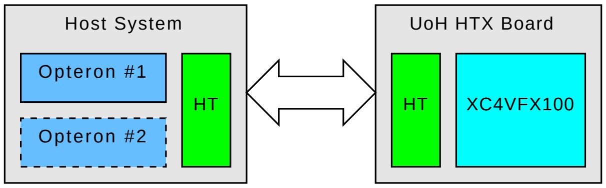


Figure 1. HTX testsystem

For double-precision with $m = 11$ and n=52, the requirements are huge:

$$k >= 1024; l >= 1074 \tag{4}$$

Adhering to these requirements does however only allow accumulation within this range, but accumulation of the largest possible values would result in infinity. Thus, additional bits are recommendable; and for multiplication of float-format encoded numbers, we need to double the amount of bits at least. Accommodating the possibility for a billion accumulations only does not seem very reasonable as numerical applications frequently iterate over meshes with some million elements per dimension. Hence, Kulisch proposes 86 additional bits for single-precision and 92 bits for double-precision accumulations.

With this in mind, when implementing an exact accumulator for single-precision arithmetics, $2 * 277 + 86 = 640$ seem to suffice.

3.2 Implementation

For accessing the exact arithmetics unit (EAU), we used the HyperTransport (HT) evaluation design of the University of Heidelberg with a Xilinx Virtex-4 FX100 in an AMD Opteron system (cf. Figure 1).

The EAU as an accelerator unit is wrapped in a memory-mapping HT interface, which is linked to the IO buffer wrappers for the evaluation board and the HTX socket, as is shown in Figure 2. Note that the reset and clock wires are not drawn for simplicity. A simplification unit has been inserted to merge the posted and non-posted requests. The memory-mapping interface allows addressing of up to 16 EAUs; however, only 14 MAC-equipped EAUs fit onto the FPGA, while 16 simpler accumulation-only units fit very well. The resource usage is discussed in more detail in Section 6.1. The design runs at a clock frequency of 100 MHz, hence allowing a theoretical unidirectional throughput of 800MB/s with 16 bits per each clock edge of the 200 MHz HT clock.

Accumulation of the decoded IEEE-754 represented values happens in small blocks of 32 bits each. The exponent determines both the block’s position inside the big accumu-

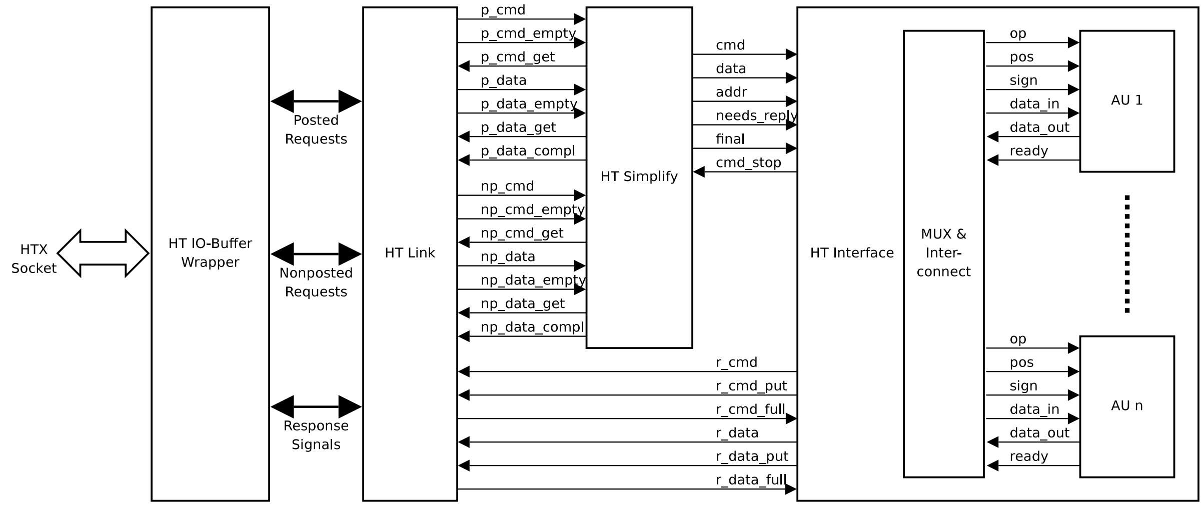


Figure 2. Device architecture

lation register and the proper shifting of the mantissa value before accumulating.

When adding the possibility for multiplication of two floating-point format data values, the sum of parts of their exponents is used for addressing the register position, with each data being shifted by the lower parts of its own exponent only. Using dual-port registers, reading the data to be accumulated to while writing the last sum to the last position is possible during the same cycle.

By using bit masks, fixing carry-resolution becomes an easy task: it requires only one additional cycle. Speaking of cycles, a regular accumulation of a single value requires up to 6 cycles after transmitting the data, where the last cycle may even overlap with the first cycle of another computation; so already with 6 accumulators we could hide the total latency completely if the calculation was split and data transfer lasted one cycle only. Multiply-accumulate, however, is finished no earlier than 18 cycles after starting the transfer of the first data word. The data transfers account for 8 of these cycles, among which 6 are due to the internal communication structure between the HT Link and the HT interface. These 6 additional transport cycles cannot be hidden by simply addressing other accumulations units as only one connection is possible during theses transfers. But one cycle can be saved when starting the next computation already during the last cycle. This leads to a latency of at least 17 cycles. With data transfers for a multiply-accumulate operation taking 3 and 5 cycles respectively, it is clear that three accumulation units can already make up for at least the computation latency; a higher number does not provide any additional advantage with regard to processing speed and hiding this latency. Throughput is therefore limited to less than 50MB/s because of the aforementioned 8 internal transport cycles when multiplying-accumulating. 4 cycles are needed for computing the addendum, and another three for writing the shifted sum in blocks to the accumulation register. This way, a clock rate of more than 100MHz can still be obtained, making the implementation suitable for the HTX bus specification, but currently suffering from high latency.

Speaking of addressing the accumulation units via the HT bus, all the units on the HT-core enabled HTX Board are memory-mapped into the processor's address space with 4kB page size, where different areas of a memory-mapped page denote different operations such as value-reading, writing flags or adding a product to the current accumulator value. This is illustrated in Figure 3. For example, writing to a page offset of 136 will prepare a multiply-accumulate operation and require a subsequent write to offset 140 for the next single-precision value to be multiplied. When reading from the first five words (i.e. addresses 0, 4, 8, 12, 16), the register value is returned as single-precision floating-point number rounded to 0, away from 0, towards negative infinity, towards positive infinity or towards nearest number, respectively.

All the accumulators are completely independent from each other except for the data transfer, which is multiplexed by the HT Memory-Mapping Interface to only one of the EAUs based on the target address. This interface passes the bundles of command, data, and address to the arithmetic units as operation code and associated data, thereby also handling replies.

The HT Simplify module then merges posted and non-posted requests, preferring the posted requests, passes them to the memory-mapping interface and cares for fetching and buffering new commands or data.

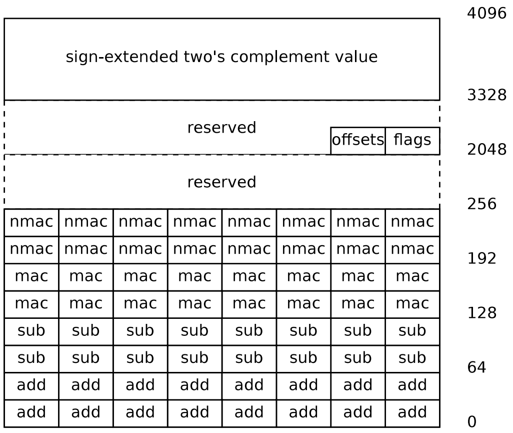

4096
sign-extended two's complement value
3328
reserved
offsets
flags
2048
reserved
256
nmac nmac nmac nmac nmac nmac nmac nmac
nmac nmac nmac nmac nmac nmac nmac nmac
192
mac mac mac mac mac mac mac mac
mac mac mac mac mac mac mac mac
128
sub sub sub sub sub sub sub sub
sub sub sub sub sub sub sub sub
64
add add add add add add add add
add add add add add add add add
0

Figure 3. Memory-mapping of EAUs

4 Runtime System

When executing numerical programs exploiting external hardware units, it is crucial to allow regular program continuation during calculation in hardware while also maintaining flexibility for choosing which software, hardware or hybrid implementation of a function to use. We developed a runtime system [3] coming in two different flavours, a Global Linking System (GLS) and a Dynamic Linking System (DLS). Both can be controlled via the Proc file system.

4.1 Global Linking System

The Global Linking System, also referred to as GOT-based Linking System for the Global Offset Table in the Linux kernel's task management, extends the lazy-linking technique of the kernel by means of the Executable and Linkable Format (ELF) [19]: the kernel extracts the function names from an application's ELF file and resolves each function symbol when the respective function is accessed for the first time. Using the Proc file system [22], the GLS resets a function symbol's structures and target function so that upon the next access onto the symbol, the dynamic linker resolves the symbol again. Function alternatives both from inside an application and from the linked libraries can be used for function switching.

The GLS is completely independent from compilers and programming languages used and also suits closed-source, proprietary software. However, although providing flexibility for mapping functions to alternate implementations, different threads will always use the same function; so regular program continuation cannot be granted when access to hardware resources used in a function implementation is exclusive-only.

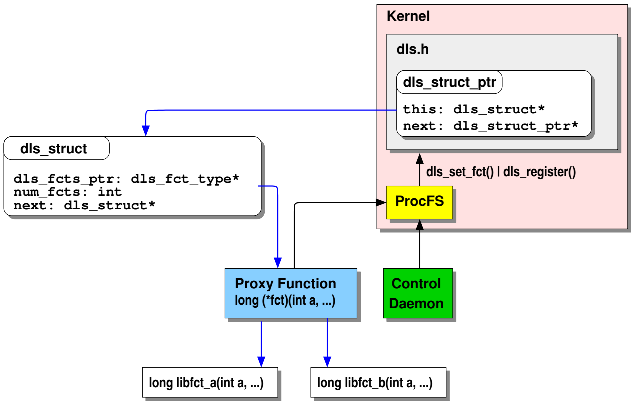


Figure 4. Switching of proxy functions

4.2 Dynamic Linking System

To overcome the missing support for individual per-thread function mapping, the DLS changes an application's task state segment (TSS) by adding function proxies per thread for each reconfigurable function. Hence, apart from linking to an additional library, the program code must contain statements indicating which functions may be exchanged at runtime, and must then indicate which functions of which libraries are appropriate implementations for the proxy. An application may hence start a software and a "hardware" thread, each thread adding a software or hardware implementation function to the list of possible alternatives. A quick overview of the kernel extensions, the Proc file system interface and the proxy functions is given in Figure 4.

The DLS can be used in two different ways, either the static version or the dynamic version.

In the static version, right from the beginning the function proxy points to a valid function implementation. The pointer may be changed via the Proc file system to a different implementation. In contrast, in the dynamic version the unresolved proxy calls a fix-up function, which in return starts the dynamic loader and adjusts the function pointer. When changing that proxy's target at runtime, a fix-up function must be given first that again resolves the proxy's pointer.

In both versions, functions can be exchanged even from inside the application itself.

4.3 Employment

Targeting matrix multiplication using exact arithmetics as a first example for numerical applications, there are two levels at which the runtime system can be employed: at the coarse algorithmic level for selecting which of the specific implementations should the proxy function be resolved to, and at the fine-grained operation level for deciding whether calculation on regular FPU, in enhanced-precision library or on external hardware unit is preferred.

The runtime system offers ultimate flexibility for both the programmer, the user, and system administrators by (dis)allowing access onto special hardware for more precision and numerical robustness or, in contrast, fast computation. For most of the systems, the programmer wants to use the DLS, where he picks the static linking variant if all implementations are already known at compile time, or the dynamic version, if new implementations become available at runtime, e.g. for long-running applications where new acceleration functions are constantly being developed. The system hence is also usable as runtime testing system for algorithms and their implementations.

5 Architecture

Our approach of combining exact arithmetics with a general runtime system for program adaptivity is based on the afore-mentioned HTX Testsystem (cf. Figure 1). A modified Linux kernel in version 2.6.20 runs both the runtime system and the programs, which may connect to the UoH HTX Board via the HyperTransport Interface of the AMD Opteron that runs at 2.0 GHz.

On the FPGA of the HTX Board, up to 16 EAUs are created and interconnected as illustrated in Fig. 2. The system clock is 100 MHz, cooperating with the 200 MHz HT clock (double-edge clocking) at a bandwidth of four 16 bit-wide transfers per system clock cycle.

The runtime system is responsible for mapping accumulate and multiply-accumulate operations onto the respective library functions, enabling emulated exact arithmetics, hardware support for exact arithmetics, or execution in regular single/double floating-point format on the Opteron's FPU.

The described 100 MHz system clock is due to hardware constraints. In order to foster maximum use of the available accelerator hardware, we employ the dynamic linking system to allow different threads to run with different implementations concurrently, allowing computation in parallel.

6 Results

In this section, we first present our hardware implementation results for the Virtex-4 FX100 1152-10 as available on the UoH HTX-Board [10], obtained with XST (ISE 9.2.04). We then give the results of some preliminary benchmarks results for different libraries both without and with use of our runtime system. We conclude by showing some potential benefit for numerical applications that suffer from convergence problems.

6.1 Implementation Results

Targeting ultimate flexibility, we implemented a basic EAU with accumulation support only and an enhanced unit with additional multiply-accumulate support so that based on available hardware resources, the operating system, dynamic runtime system or the user herself can decide how much and which coprocessor support to exploit in the current setup.

Hence, we present in Tables 1 and 2 the results of the synthesis runs for 1, 2, 4, 8, and 16 accumulator-only and MAC-extended EAUs, respectively. As we can see, the HT interface logic and the mapping interface only make up for a minor part of the system and with increasing number of EAUs, the routing costs rise enormously. Note that the number of needed resources depends strongly on several factors such as software, host system used for synthesis, synthesis settings, and the hardware description itself. Consequently, the results cannot be regarded as accurate, but only indicate the approximate amount of required resources.

The actually better results for the multiplication unit arise from the need to specify very strict requirements for synthesis and mapping.

Rather independent from the number of EAUs, the maximum theoretical design frequency is about 120 MHz for the accumulate-only design and about 100 MHz for the MAC design, independent of the number of EAUs. The critical path is determined by selecting from the large register; this can however be circumvented by splitting the large register into several smaller resources or mapping it directly onto the hardware BRAM resources. The HyperTransport core requires its client to run at 100 MHz or 200 MHz respectively, using a differential clocking with 200 MHz for the bus interface and merging the 16 bit connection into an internal 64 bit wide bus interface.

6.2 Runtime Results

For the results below, we indicate the mean value of ten runs, measured in clock ticks as reported by the CPU's timestamp counter.

First, we measured the time in clock ticks for both an IJK and IKJ matrix multiplication with arbitrarily chosen sizes of 30x20 * 20x16 and 100x100 * 100x16. The programs were compiled with `-O2`. Note that the enormous runtime for the 1-MAC-unit version is due to the necessity to read values in between of two same operations or to do any other operation freeing the virtual HyperTransport channel for the accumulator. Also, the software-emulated MAC support for high-precision arithmetics still sometimes produces erroneous results and cannot be compared therefore, but already gives a rough estimate. The results without any additional runtime support are given in Figure 5.

Table 1. Hardware implementation results for accumulation-only units

Resource	Used Resources					Available
	1 EAU	2 EAUs	4 EAUs	8 EAUs	16 EAUs	
Slice Flip Flops	4,835	5,350	6,409	8,536	12,783	84,352
Occupied Slices	7,795	9,501	13,701	19,254	31,923	42,176
4 input LUTs	10,324	13,274	19,312	31,477	55,788	84,352
Logic	10,066	12,888	18,654	30,291	53,562	
Route-thru	234	362	634	1,162	2,202	
Shift registers	24	21	24	24	21	
RAMB16s	26	27	29	33	41	
Equiv. gate count	1,811,662	1,902,807	2,085,366	2,451,628	3,184,303	

Table 2. Hardware implementation results for multiply-accumulation units

Resource	Used Resources					Available
	1 EAU	2 EAUs	4 EAUs	8 EAUs	16 EAUs	
Slice Flip Flops	5,085	5,890	7,505	10,728	17,167	84,352
Occupied Slices	7,955	10,718	14,846	24,008	39,141	42,176
4 input LUTs	11,355	15,535	23,868	40,570	73,868	84,352
Logic	11,059	15,077	23,084	39,134	71,129	
Route-thru	272	434	760	1,412	2,715	
Shift registers	21	21	21	21	24	
RAMB16s	26	27	29	33	41	
DSP48s	4	8	16	32	64	160
Equiv. gate count	1,821,103	1,923,091	2,126,585	2,533,808	3,346,401	

Execution times increase for the IKJ multiplication as both compiler optimization makes up for the non-optimally aligned memory accesses and few potential for overlapping computation in hardware is given due to the layout of the algorithm. The runtime of the software-emulated MAC unit is a rough estimate only as the implementation hampers from incorrect algorithmic implementation and may produce wrong results.

Furthermore, we evaluated three different routines for getting the sine value of the program's argument. The first implementation is the regular call of the sine function in the math library of the GNU C Library, the second uses a look-up table as has been done for example in Quake III Arena to offer sufficient speed while achieving good precision on those days' PCs, and finally, the Taylor series:

$$sin(x) = \sum_{i=1}^{n} (-1)^{i+1} \frac{x^{2i-1}}{(2i-1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \quad (5)$$

The Taylor series implementation is much slower than the others, but it allows external coprocessor support for accumulating each summand in hardware and also swapping the construction of the summands into the hardware accelerator. Figure 6 presents the runtime results for different implementations with and without support for exact arithmetics plus the number of iterations needed until the result was stable. For the three different routines, the runtime system has been used to allow measuring runtimes of implementation alternatives by simply switching the function pointers from inside the application at program runtime, with the static linking of the DLS being sufficient for this purpose. The result was stable after 9 iterations for double precision, 4 iterations for emulation without MAC support, and 5 iterations for the remaining runs.

6.3 Exact Arithmetics

Accuracy is achieved as shown in Table 3: with exact arithmetics, the result is more precise than with regular single-precision operation. The bad runtime for the MAC-enabled unit is again due to the additional operation needed for clearing the HT interface registers. Except for the double-precision runs, after 10 iterations the result is already stable. This is due to obtaining single-precision values only when reading the accumulator value: the convergent Euler series is also convergent for the iterations following iteration number 10, which hence do not influence the single-precision value to any extent. Reading double values from the EAU in subsequent implementations will produce different results. After 39 and 178 iterations respectively, the single and double precision windows are too small for storing the component values of the product to be accumulated.

A more detailed result[1] for the EAU can be obtained from the hardware accumulator when adding the subsequently read value to the intermediate coarse result value after subtracting it from the accumulator. The gain in precision with the MAC unit is ascribed to not losing the re-

[1] 2.71828183528879208097350783646106719970703125 (accumulator only) and 2.71828183518901056459071696735918521881103515625 (MAC-extended)

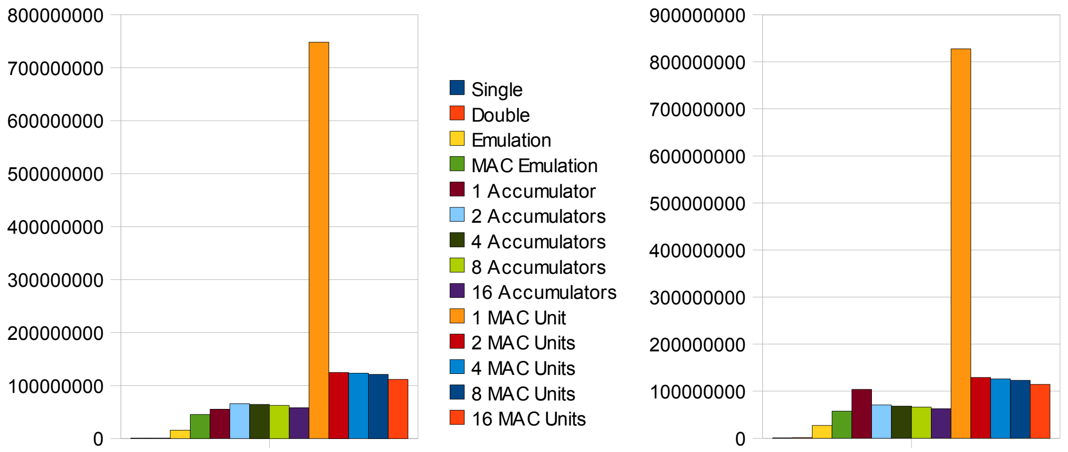
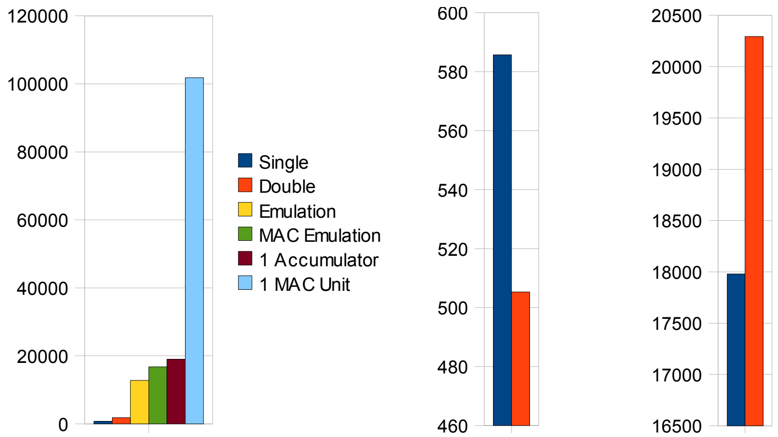


Figure 5. Comparison of runtimes for IJK and IKJ matrix multiplication with and without exact arithmetics



(a) Taylor series (b) Look-up table (c) GNU C Library

Figure 6. Runtimes and iterations for calculating the sine value of 1.0

Table 3. Runtimes and iterations for calculating the Euler number

Implementation	Result	Runtime	Iterations	
			Result fixed	Summand! = 0
Single Precision	2.7182819843292236328125	3,150	10	39
Double Precision	2.71828182845904553488480814849026501178741455078125	6,566	17	178
Emulation w/o MAC	2.71828174591064453125	19,935.9	10	39
Emulation w/ MAC	2.71828174591064453125	26,171.3	10	39
Accumulator	2.71828174591064453125	31,486.2	10	39
Multiply-Accumulator	2.71828174591064453125	188,945.7	10	39

mainders of the product. This clearly shows how accurate the obtainable results are in comparison to single-precision format. Similarly, when reading from the coprocessor register, the results of the Taylor sine routine are a little more precise than their native FPU counterpart.

With multiplication in hardware, we gain from not being limited to the format in use, i.e. single floating-point precision with a mantissa of 23 bits, but instead being able to accumulate the exact product onto the previous register value. With accumulation only, the multiplication has to be carried out on the host processor and only the rounded result, limited to the precision in use, can be accumulated, hence loosing valuable information.

For example, multiplying single-precision data converted to double-precision values would achieve a more precise representation of the product than with single-precision only; but as soon as the result is converted for adding regular single-precision values, the additional precision is lots. On the MAC hardware, though, the result of additional MAC and accumulation operations onto the previous multiplication result is still precise and the accumulated error due to rounding and format limits is much smaller. Of course, its additional precision will be lost as well when converting to single-precision format for further processing on the host processor.

7 Conclusions and Outlook

Developments in interconnection and FPGA technologies enable the use of reconfigurable logic as application-specific hardware accelerators.

In this paper, we showed an implementation of an exact arithmetics hardware unit using an FPGA-equipped HyperTransport device as a coprocessor. Through a lightweight runtime systems, dynamic per-function control of which implementation or software to use for calculations is made possible. The hardware is easy to use from a programmer's view and can be completely hidden from the user because the runtime systems offers the necessary abstraction and wrapping and because control is possible from within the application itself, if desired.

The arithmetics unit uses a wide fixed-point representation of the data, enabling accumulation of both very small and very large numbers altogether. The hardware unit is addressed via memory-mapping, which enables separate usage of up to 16 arithmetics units at once, thus speeding up parallel, separate computations by hiding latency. The HyperTransport bus is controlled by an AMD Opteron, the Virtex-4 is connected to the HT bus through an HTX board. For now, only single-precision is supported, but the increase in exactness due to not loosing the additional bits when multiplying and adding in our hardware in contrast to computing on regular floating-point units already proves the benefit of such architectures and proves both feasibility and reliability of such an arithmetic coprocessor for exact arithmetics.

The runtime system is a lightweight extension to the Linux kernel, altering the dynamic loading of libraries that can be controlled via the *procfs* interface. Alternate libraries may emulate hardware, access it directly, or offer fast and unreliable implementations – the user or a runtime system can choose, which one to use in his system based on availability and load.

We conducted several experiments regarding the basic operations required by the targeted numerical applications such as frequent multiply-accumulate as needed in solving linear equation systems. The results deliver further proof that supporting single-precision is far not enough when targeting real-world applications as the regularly obtainable data is not precise enough. We thus deduct that exact accumulation is only useful when appropriate means are given for retrieving those bits that cannot be represented in the regular floating-point format so that additional digits of a calculated high-precision value be obtainable. It also becomes more expedient when supporting double precision, offering more precision than regular double-precision without introducing the need for additional computation libraries such as QD [1].

Thus, we can conclude that 1) using coprocessor technologies for arithmetics is a valid and suitable approach, 2) HyperTransport fits well as interconnection technology, and 3) usage of exact arithmetics needs no longer be an issue for programmers due to the achievable enhancements of dynamic runtime linking.

For the hardware, our ongoing work includes extending the arithmetic unit by exact multiplication, decreasing overall latency and area, and increasing clock frequency of the implementation, thereby enabling usage for even small amounts of computations without a large amount of runtime overhead. Support for double-precision is also a nearby goal, both for reading double values, accumulating double values and accumulating double-precision products. Further plans for the runtime system include using it for a wider range of applications such as dynamic systems and debugging, but also incorporating reconfigurable hardware as memory-mapped devices into the operating system, which allows building and loading custom accelerators per application. Access onto the available resources will have to be managed then by offering only a few virtualized units to applications.

Acknowledgment

The authors would like to thank Reimar Döffinger for the initial work on the exact accumulation unit.

References

[1] D. H. Bailey, Y. Hida, K. Jeyabalan, X. S. Li, and B. Thompson. QD. Web site: http://crd.lbl.gov/~dhbailey/mpdist/.

[2] U. Brüning. The HTX board – a universal HTX test platform. Web site: http://www.hypertransport.org/members/u_of_man/htx_board_data_sheet_UoH.pdf.

[3] R. Buchty, D. Kramer, M. Kicherer, and W. Karl. A light-weight approach to dynamical runtime linking supporting heterogenous, parallel, and reconfigurable architectures. In *Architecture of Computing Systems – ARCS 2009, 22nd International Conference*, Lecture Notes in Computer Science (LNCS), Delft, Netherlands, March 2009. GI e.V. to appear.

[4] Celoxica. Handel-C Language Reference Manual, 2001.

[5] ClearSpeed Technology plc. ClearSpeed Advance X620 and e620 Accelerator Boards, 2006. Web site: http://www.clearspeed.com/products/cs_advance/.

[6] Cray Inc. Cray XD1 Supercomputer, 2004. Web site: http://www.cray.com/downloads/Cray_XD1_Datasheet.pdf.

[7] E. Ej-Araby, I. Gonzalez, and T. El-Ghazawi. Bringing High-Performance Reconfigurable Computing to Exact Computations. *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 79–85, Aug. 2007.

[8] C. F. Fang, T. Chen, and R. A. Rutenbar. Floating-point error analysis based on affine arithmetic. In *Proc. IEEE Int. Conf. on Acoust., Speech, and Signal Processing*, pages 561–564, 2003.

[9] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2):13, 2007.

[10] H. Fröning, M. Nüssle, D. Slogsnat, H. Litz, and U. Brüning. The HTX-Board: A Rapid Prototyping Station. *Proceedings of the 3rd Annual FPGA World Conference*, November 2006.

[11] G. Gerwig, H. Wetter, E. M. Schwarz, and J. Haess. High Performance Floating-Point Unit with 116 Bit Wide Divider. In *ARITH '03: Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH-16'03)*, page 87, Washington, DC, USA, 2003. IEEE Computer Society.

[12] T. Granlund. The GNU MP Bignum Library, 2008. Web site: http://gmplib.org.

[13] K. Group. Khronos OpenCL API Registry. December 2008. http://www.khronos.org/registry/cl/.

[14] B. Hendrickson and J. Berry. Graph Analysis with High-Performance Computing. *Computing in Science & Engineering*, 10(2):14–19, March-April 2008.

[15] S. S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah. Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary. In *ECOOP 2008 Object-Oriented Programming*, volume 5142/2008 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2008.

[16] HyperTransport Consortium. Low Latency Chip-to-Chip and beyond Interconnect, 2005. Web site: http://www.hypertransport.org/.

[17] N. Inc. High Performance FPGA Computing Solutions for Defense and HPC, 2005. Web site: http://www.nallatech.com/.

[18] J. Kernhof, C. Baumhof, B. Höfflinger, U. Kulisch, S. Kwee, P. Schramm, M. Selzer, and T. Teufel. A CMOS Floating-Point Processing Chip for Verified Exact Vector Arithmetic. *Proceedings ESSCIRC '94*, pages 196–199, September 1994.

[19] J. Koshy. libelf by Example. Web site: http://people.freebsd.org/~jkoshy/download/libelf/article.html.

[20] U. Kulisch. The XSC tools for extended scientific computing. In *Proceedings of the IFIP TC2/WG2.5 working conference on Quality of numerical software*, pages 280–284, London, UK, 1997. Chapman & Hall, Ltd.

[21] U. W. Kulisch. *Advanced arithmetic for the digital computer: design of arithmetic units*. Springer, 2002.

[22] M. Tim Jones. Access the Linux kernel using the /proc filesystem. In *IBM developerWorks*, 2006. Web site: http://www.ibm.com/developerworks/library/l-proc.html.

[23] R. E. Moore. *Interval arithmetic and automatic error analysis in digital computing*. PhD thesis, Stanford University, Stanford, CA, USA, 1963.

[24] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent RAM. *IEEE Micro*, 17(2):34–44, 1997.

[25] J. G. Rokne. Interval arithmetic and interval analysis: an introduction. *Granular computing: an emerging paradigm*, pages 1–22, 2001.

[26] P.-M. Seidel. High-radix implementation of IEEE floating-point addition. *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on*, pages 99–106, June 2005.

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

[28] T. Sterling, J. Brockman, and E. Upchurch. Analysis and Modeling of Advanced PIM Architecture Design Trade-offs. In *SC'2004 Conference CD*, Pittsburgh, PA, November 2004. IEEE/ACM SIGARCH.

[29] D. Tan, A. Danysh, and M. Liebelt. Multiple-precision fixed-point vector multiply-accumulator using shared segmentation. *Computer Arithmetic, 2003. Proceedings. 16th IEEE Symposium on*, pages 12–19, June 2003.

[30] S. D. Trong, M. Schmookler, E. Schwarz, and M. Kroener. P6 Binary Floating-Point Unit. *Computer Arithmetic, 2007. ARITH '07. 18th IEEE Symposium on*, pages 77–86, June 2007.

[31] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte. The Molen Polymorphic Processor. *IEEE Transactions on Computers*, pages 1363–1375, November 2004.

[32] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 156–166, New York, NY, USA, 2007. ACM.

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.

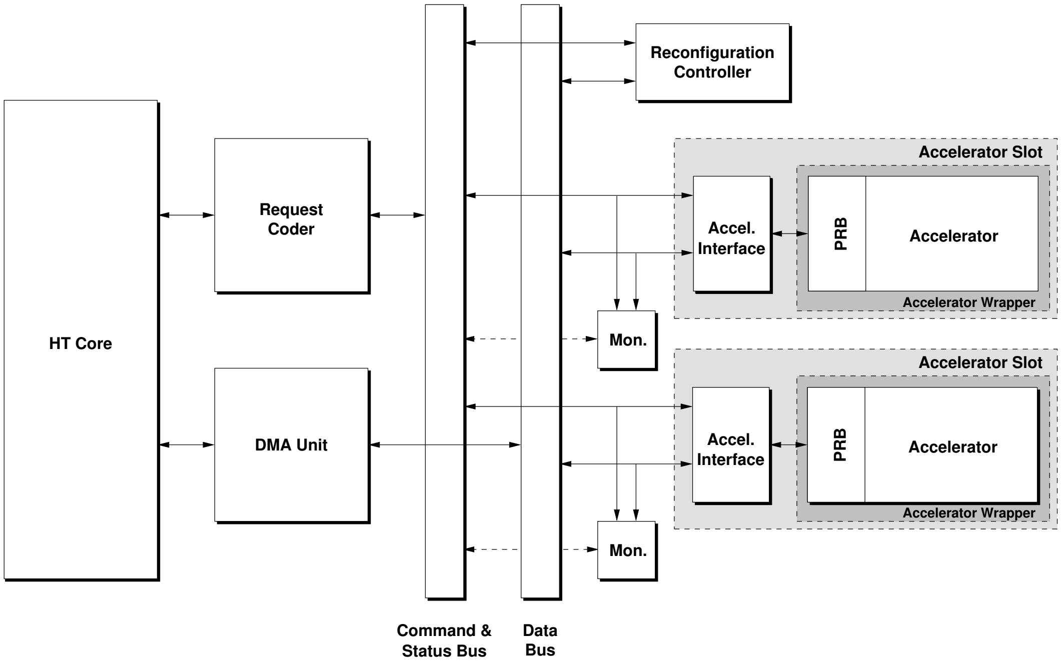


Figure 1. Hardware Composition

```
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,

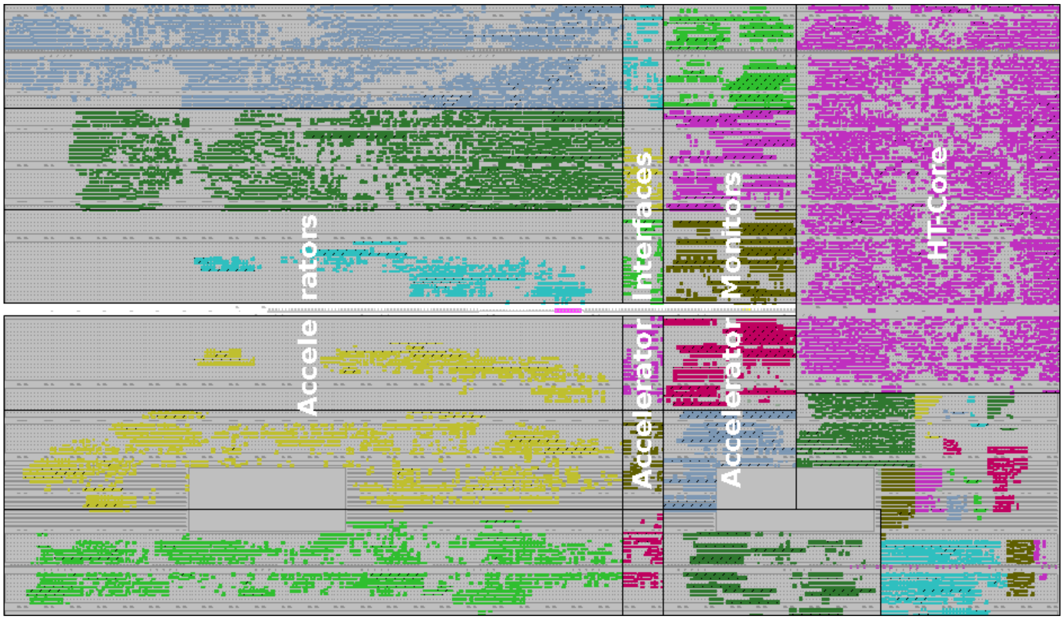


Figure 3. Floorplan

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-

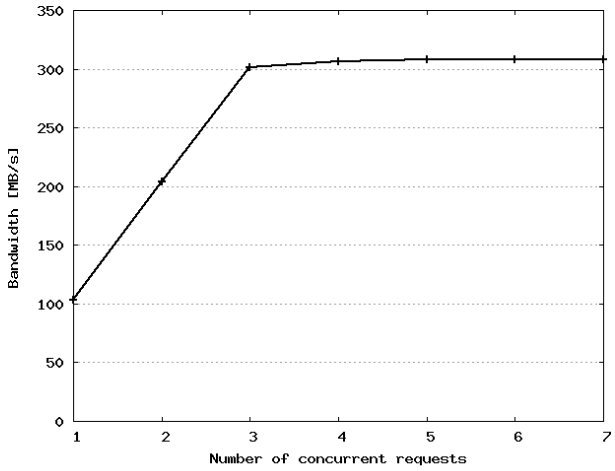


Figure 4. Achieved read bandwidth

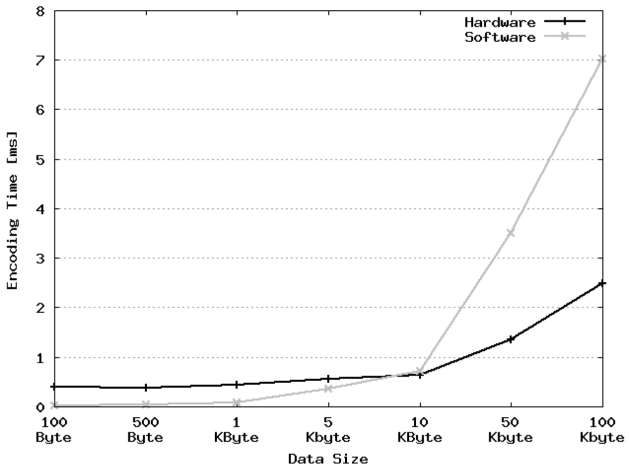


Figure 5. 3DES: Encoding time for software or hardware encryption

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.

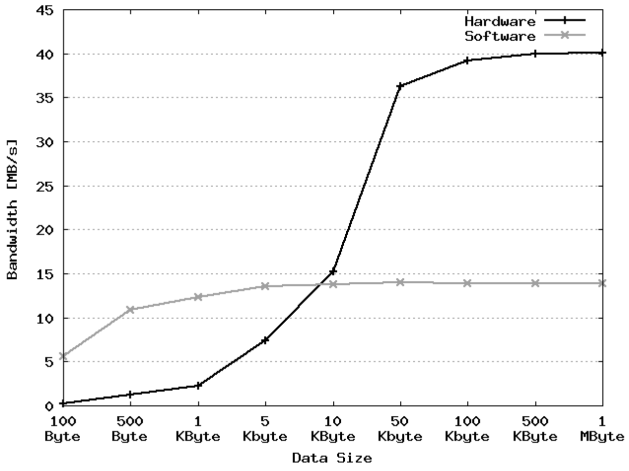


Figure 6. 3DES bandwidth

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 accelerators. 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.

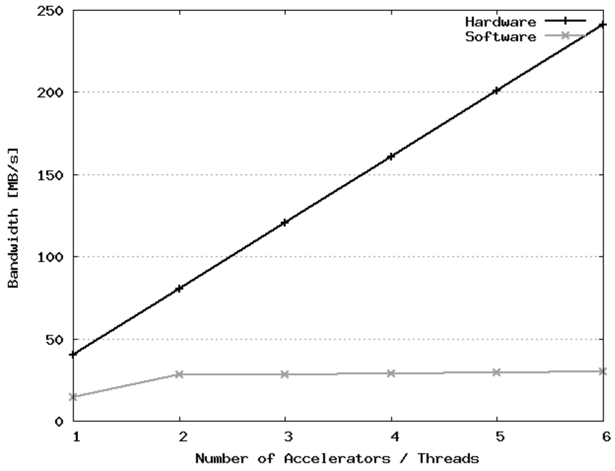


Figure 7. 3DES: Bandwidth – concurrent encryption

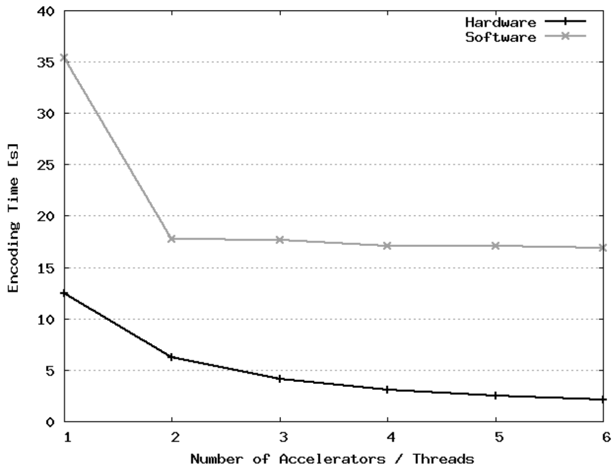


Figure 8. 3DES: Duration – concurrent encryption

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^{TM}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^{TM}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 opensource 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.

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

PGAS Model for the Implementation of Scalable Cluster Systems[*]

Juan A. Villar, Francisco Andújar
Francisco J. Alfaro, José L. Sánchez
DSI – Univ. of Castilla–La Mancha
02071 – Albacete, Spain
{juanan, fandujar, falfaro, jsanchez}@dsi.uclm.es

José Duato
DISCA – Tech. Univ. of Valencia
46022 – Valencia, Spain
jduato@disca.upv.es

Abstract

This paper introduces an extended version of the traditional Partitioned Global Address Space (PGAS) model, for the implementation of scalable cluster systems, that the HyperTransport Consortium Advanced Technology Group (ATG) is working on. Using the Simics and GEMS simulators, we developed a software module that approximates the behavior of a PGAS cluster. This approach mainly provides the simplest mechanism to evaluate how much the PGAS infrastructure will affect overall the application performance. The aim of this work is to study the feasibility of the ATG's PGAS model for running applications with high memory requirements. Such a model, will let manufacturers build clusters that enable the execution of these applications, in such a way that it will be impossible to run them in a single processor, or in a multi–processor.

1. Introduction

Traditionally, shared memory systems have been used to run applications requiring a high memory space. However, such as systems do not scale more than tens of processors. As memory requirements of applications and the number of applications that run concurrently on computers have been increasing, designers have made proposals to partially solve the lack of memory on computer systems. In this way, modern operating systems provide advanced virtual memory managers that solve the lack of memory. These managers utilize secondary devices for freeing contents of memory whenever it is necessary. This approach provides a simple way of running applications that have a running image size bigger than available physical memory size. When an application is running and the system is low of memory, the virtual memory manager can evict it to a special device called a "swap device", or swap, to free memory. This technique is called swapping [17], and there are several approaches in the literature about it. Just to mention a few, Unix–based systems use a separate swap partition type that is hosted in the user file system. In contrast, Windows uses a user–space file that is hosted inside the file system, while the MacOS X operating system can use partitions and files.

Nevertheless, swapping has several drawbacks. The first one comes from the access time of the swapping device, which is usually a hard disk, therefore, one or more orders of magnitude higher than the memory access time. The second drawback is thrashing, which occurs when the memory manager evicts parts of the running image of a process and, after a time, it reallocates those parts in memory again. Thus, solving the lack of memory always involves a high run time.

The AMD's Opteron processor can be used as a commodity to build clusters. This processor includes the memory controller on–die, in such a way that all memory is accessible from one memory controller. The Opteron processors use the AMD's HyperTransport protocol [8] for communicating with each other. Moreover, the HyperTransport protocol enables CPUs to directly connect the Opteron HyperTransport link to add–in card subsystems via the HTX connector [5], which is placed in the motherboard.

The HyperTransport Consortium Advanced Technology Group (ATG) is working on an extended version of the traditional Partitioned Global Address Space (PGAS) programming model [3]. Such a model will let manufacturers build clusters with PGAS native support. In this paper we carry out an assessment of "rough" PGAS model through Simics [10] and GEMS [11] simulators. However, this work has not attempted to conduct a study using a hardware implementation, because the ATG has not completed the specification of its PGAS model yet, and therefore it is impossible to achieve that kind of evaluation. The behavior of the final system will be similar (bridging the gap) to the behavior in a cluster with PGAS native support.

[*] This work has been jointly supported by the Spanish MEC and European Commission FEDER funds under grants Consolider "Ingenio–2010 CSD2006–00046" and "TIN2006–15516–C04–02"; and by Junta de Comunidades de Castilla–La Mancha under grant PCC08–0078–9856 and Beca Predoctoral de Investigación 07/096.

In addition, the increasing use of interconnection networks to intercommunicate the processor and memory, such as AMD HyperTransport, might allow the construction of scalable systems, from hundreds to thousands of nodes composed of multicore processors. Therefore, the Opteron processors will provide the basis to build the clusters with PGAS native support.

The remainder of this paper is structured as follows: In the next section we present a summary of the related work. The details of the proposed model are explained in Section 3, while Section 4 details the simulation scenarios and the results obtained. In the last Section, we conclude and provide a brief overview of the future work.

2. Related Work

HyperTransport is an interconnection technology which enables connecting the processors among each other and with the I/O devices. It provides an extremely low latency, high bandwidth and excellent scalability. Moreover, the definition of the HTX connector allows co–processing and acceleration based on ASIC or FPGA technologies. In particular, it is receiving a highlighted interest of the community because it makes easy to reduce execution time by the use of accelerators.

Partitioned Global Address Space languages combine a Single Program Multiple Data (SPMD) programming model with a global address space, which is logically partitioned to give each thread a portion of shared memory to which it has affinity [19]. In the SPMD model, a fixed number of threads are created at program startup, and every thread runs the same program. Each thread has both a space for private local memory and some partition of the shared space to which it has affinity. A private object may only be accessed by its corresponding thread, whereas all threads can read or write any object in the shared address space. The partitioning of the shared space into regions with logical affinity to threads allows programmers to explicitly control data layout, which is then used by the runtime system to map threads and their associated data to processors: on a distributed memory machine, the local memory of a processor holds both the thread's private data and the shared data with affinity to that thread.

The HTC Advanced Technology Group (ATG) [1] is working on developing proposals for HyperTransport to define an address space globally and dynamic partitioning (PGAS) for using in scalable clusters. The idea is not new, but it comes from the existing PGAS models [3]. In the bibliography several PGAS applications can be found, for example, Unified Parallel C [4] to define models of programming languages. In addition, developers of these languages have tools like GASnet [2] which is a communication interface for programming languages such as Unified Parallel C. GASnet is a language independent of the network that allows the definition of libraries providing global addressing. GASnet is inspiring the work of HTC Advanced Technology Group for the implementation of PGAS [18] in a native way.

The ATG is proposing the mechanisms and abstractions that will allow the construction of clusters using Opteron processors. The motherboard containing Opteron processors will support the HTX connector. By plugging extension cards on the HTX connector will allow the formation of a cluster of motherboards. The memory controller of each Opteron will divide the whole range of physical addresses in regions and distribute them among the memory of other Opterons. Subsequently, the memory controllers will be able to access remote regions of memory transparently. Following this approach, the Opteron processors can avoid the use of a device for the lack of memory, since access to remote memory controller will be lower than access to a local secondary storage device, and hence system performance will be enhanced.

In such systems, the physically addressable memory in all nodes is part of a global address space with non–uniform access time from any specific node. From the perspective of a node, the global address space is composed of local partitions and remote partitions where the former can be accessed with the lowest latency and the latter can be accessed with larger and possibly non–uniform latencies (across distinct partitions). In this model, a local partition refers to DRAM accessed through a tightly integrated memory controller. The latency of remote memory accesses will be non–uniform because it will depend on interconnection performance and the load and contention of the network.

Specific details of the implementation of the PGAS model cannot be provided in this paper because they are still confidential, and some aspects of it are still under deliberation of the ATG group.

The Computer Architecture Group at the University of Heidelberg in Germany has the expertise to design complex hardware/software systems. The HTX–Board [7], a contribution of this group, provides a convenient and efficient way to evaluate user specific devices connected to the Hypertransport connector standardised under the name of HTX–Connector. In [16] they published the architecture and mechanisms of the HTX–board. Subsequently, in [9] they have introduced a novel communication engine in combination with the HyperTransport interface. They provide an excellent prototype to get real–world measurements connecting two Opteron through two HTX–boards. Moreover, they also show the initial latencies of sending a HyperTransport packet on a HyperTransport link and propose some optimisations that can minimise that latency (e.g. doubling the HyperTransport clock frequency or migrating FPGA to ASIC technology).

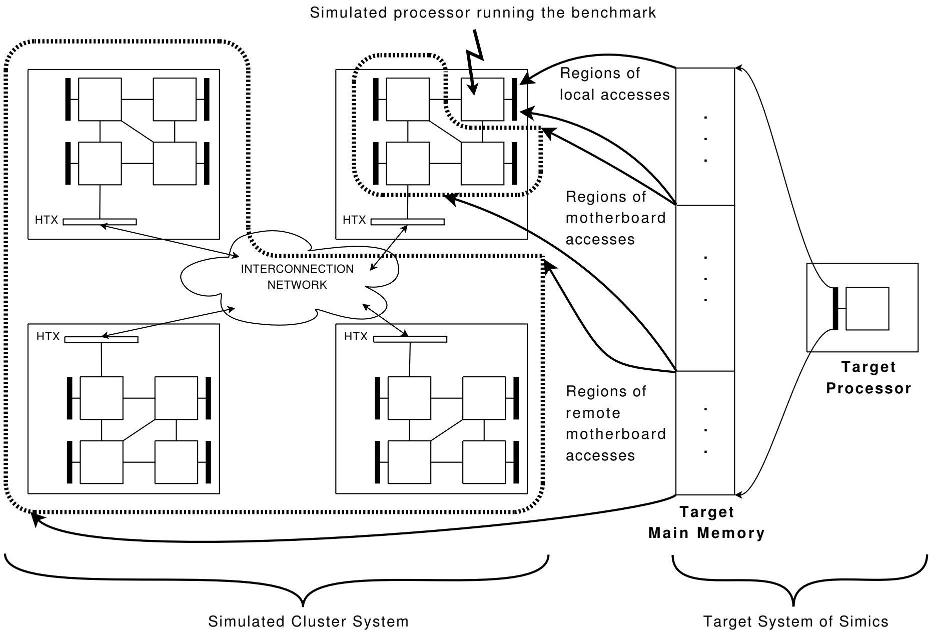

Simulated processor running the benchmark
Regions of local accesses
Regions of motherboard accesses
Regions of remote motherboard accesses
HTX
HTX
HTX
HTX
INTERCONNECTION NETWORK
Target Processor
Target Main Memory
Simulated Cluster System
Target System of Simics

Figure 1. Diagram of the simulated cluster.

3. Model

Because of the fact that the ATG has not completed the specification of its model PGAS yet, it is impossible to carry out an evaluation using a hardware system. However, it is feasible to carry out an approximate evaluation of the PGAS model by simulation.

In order to simulate the cluster, Simics 2.1 and GEMS 2.2.19 simulators were used. In the Simics context, two fundamentals terms are always used:

- The computer on which we are running Simics is referred to “host system”.
- The computer simulated by Simics is referred to “target system”. Specifically, it simulates the cluster with PGAS support.

Our approach consists in simulating the execution of one sequential application as if it would be running on a cluster with PGAS support just using the execution–driven Simics simulator. In that hypothetical cluster, any processor might issue requests of the global address space. The global address space will consist of the joint of every memory in the cluster. From the processors point of view, most of the physical memory in the cluster can be accessed, except some private areas that will not be allowed to access. The Figure 1 explains how a cluster can be simulated starting from a simulated processor in Simics.

Without lossing generality, we run one sequential application in the target system and we process every message going inside the system. The messages are deliverated by the simulator as they are in a usual execution, but the delay of every message is customised depending on the type, source and destination of the message.

In the AMD’s whitepaper [14] a suite of benchmarks is examined to illustrate their performance and scalability in single, multi–processor, and cluster configurations. Its results clearly show the exceptionally responsiveness of an Opteron–based NUMA support system. Specifically, it shows the cost for one processor for accessing to the shared memory in a four AMD Opteron motherboard. The latencies are given depending on the distance between the transmitter and receiver processors.

Regarding the application, it must be a benchmark that makes an intense use of memory. Also, a sequential application is preferred in order to avoid any dependency produced by a parallel execution. Stream [12] is a well–known benchmark that measures bandwidth sustainable by ordinary user programs, and not the theoretical peak bandwidth that vendors advertise [13]. Moreover, it is used by AMD to measure the performance of the memory of their processors [15].

The benchmark performs functions with matrices that are stored in memory. The functions are executed several times. When the benchmark concludes, it returns the rate of traffic data of memory in MB/second and execution time (average, minimum and maximum) in seconds, for each function. Both performance indexes are the most relevant in this kind of benchmarking. The execution time gives the global performance measure and the traffic rate offers the real load of the memory system.

4. Evaluation

In this section, we start describing the simulation model we have used to carry out our experiment. Then, we present the results we have obtained and some comments about them.

4.1. Simulation Model

In all the simulations, our customized GEMS module is loaded. It is responsible for managing all the messages sent by the memory system. We assume that the target processors utilised in this work are used to build systems with a coherent shared memory, similar to the Opteron processors. Therefore, the GEMS module has to manage the messages caused by the memory coherence protocols. The study of memory coherence protocols is out of the scope of this paper. Thus the simulations have been carried out with a single processor in order to reduce the influence of these protocols.

The host system is a SUN W2100Z workstation that has two Opteron processors at 2.4 GHz and a DDR–400 memory of 4 GB. SuSe 10.2 is used as operating system. The target system is the *sarek* preinstalled system of Simics which is a UltraSPARC processor at 75 MHz. Solaris is used as operating system and an amount of 512 MB of memory is configured.

The assumption that the whole memory of the cluster is 512 MB seems initially nonsense. However, the aim of this work has never been to propose a detailed PGAS simulation model because the ATG has not finished its PGAS model yet. Considering an increase of the target memory size, that is the memory size of the cluster, requires to increase the Stream benchmark size and then it causes an exponential simulation time growth that would be unaffordable. Even though the results remain representative because the benchmark spreads the accesses out the memory address space.

Additionally, the parameters of the Stream benchmark have been set to achieve the following behavior:

1. The target host is running a unique process of Stream benchmark in absence of processes that interfere with Stream. Meanwhile, the memory accesses are controlled by the customized GEMS module.
2. Stream runs two series of functions (copy, scale, add and triad). Previous tests had proved that increasing the number of series does not alter the final outcome in the absence of processes that interfere with Stream.
3. The size of Stream is 460 Mbytes. This size was chosen because it represents 90% of the available simulated memory (512 MB).
4. The GEMS simulator requires to set the latencies in the target processor cycles, so a 2.4 GHz processor was considered for the translation from real nanoseconds to simulated cycles.

It must be noticed that the latencies in our simulations are considered as an approximation. However, we have selected values that reflect real systems:

- When an access is destined for memory allocated in the same motherboard, a latency of 115 ns for each access has been considered [14]. In that case, this value is the mean time for both read and write accesses.
- When an access is destined for memory allocated in a remote motherboard, the latency has been deduced from the proposed delays in [9, 16]. We assume all the improvements suggested by [9, 16] like ASIC technology instead of using FPGA technology, doubling the HyperTransport link frequency, and a 16–bit HyperTransport link width. In this case, the calculation of the latencies is:
    - Due to all the technology improvements, [9] claims a total latency of 130 ns for the transmission and it expects a fixed latency (for local CPU and remote memory controller) of about 300 ns. The functionality of [9] is exactly the behavior of a remote write operation (transmision of data from source node to destination node). Hence we assume a latency of 430 ns for a write operation of 64 bytes payload.
    - The read is much more costly. The Opteron used in [16] only issues 32 bit read operations. The read operation consists of transmiting the read request to the destination node, and then transmitting the result back to the source node. Because of the access granularity of 8 bytes (it is a limitation of the Opteron K10 architecture) a simple 8 bytes read operation costs 610 ns. Therefore, a series of 8 consecutive operations have to be issued to retrieve the total amount of 64 bytes. This is the reason for assuming the read operation takes 4880 ns.
- When an access is destined for memory mapped in a swap device, we assume a latency of 45,600 ns as

it is suggested in [6] for enterprise class harddisks. In that case, this value is the mean seek time for a variable 512 bytes sector size. We assume this time as a representative, so we do not consider neither the sector size effect nor cache implications.

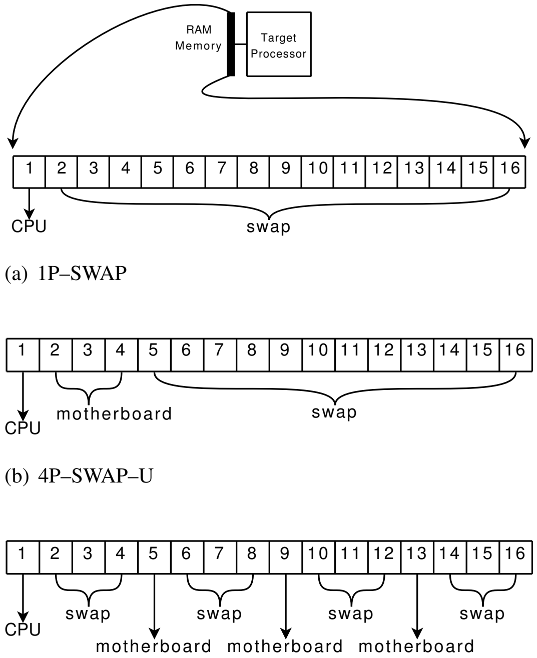
Regarding to simulation scenarios, we have considered the following scenarios:

- Local scenario (1P): It represents a desktop system with just one processor and one motherboard.
- Shared scenario (4P): It represents a server system, commonly known as a shared memory multiprocessor. There are four processors assembled in one motherboard.
- Remote scenario (16P): It represents a enterprise system or a cluster. A total of sixteen processors are distributed between four motherboards of four processors per motherboard that are interconnected using HTX connectors.

A group of extra three scenarios have been selected to evaluate the loss of performance due to the utilisation of swapping. Figure 2 shows the distribution of the target memory into regions and the access type that is associated with each region. Each region corresponds to a contiguous physical memory partition controlled by a single node. In our test, we assumed all regions are of the same size. These extra scenarios are based on the previous scenarios and they consist in distributing the regions of memory between swapping devices, as if they were accesses to secondary devices and therefore such accesses suffer an extra delay. All the extra scenarios assume a partitioning of the memory in 16 regions. Specifically, these three extra scenarios are:

- 1P–SW: It is basically the 1P scenario, but the accesses from the second to the last regions are accesses to a swap device (see Figure 2(a)).
- 4P–SW–U: It is similar to the 4P scenario. The swap regions are assigned uniformly (see Figure 2(b)).
- 4P–SW–D: Similar to the 4P–SW–U scenario, but the swap regions are interleaved on the memory (see Figure 2(c)).

The previous partitioning of the memory is quite extreme because it assigns up to 90% of the target memory to swap. However, that partitioning represents a suitable configuration for checking the influence of the target operating system and how its target virtual memory manager allocates the target memory (e.g. how the target memory is allocated to the applications).



(a) 1P–SWAP

(b) 4P–SWAP–U

(c) 4P–SWAP–D

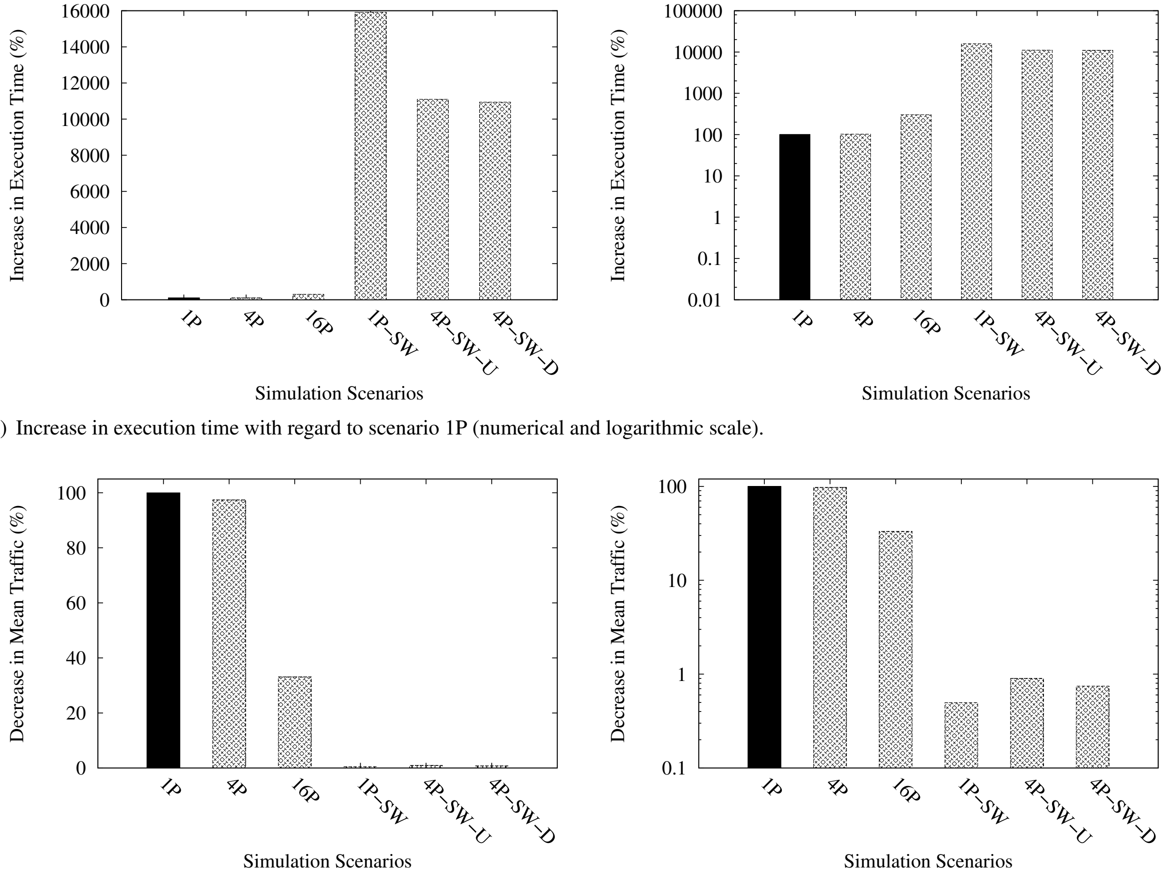
Figure 2. Distribution of target memory into regions and their access type.

4.2. Simulation Results

The aim of this work has never been focused on the performance of the application, but the behavior of a cluster with native PGAS support (scenario 16P) to run applications. Note that it will be impossible to run these applications in a single processor (scenario 1P) or multi–processor (scenario 4P) due to the memory requirements of these applications. Note also that the only alternative in these cases is the use of swapping devices, which is considered in the 1P–SW and 4P–SW scenarios.

Mainly, it is interesting to know how the execution time evolves. Figure 3 depicts that the execution time increases for 4P and 16P on average 0.54 seconds (2.68%) and 40.76 seconds (202.68%) with regard to the 1P scenario. Because of it is a memory benchmark, it is also interesting to know how the traffic memory evolves, so Figure 3 depicts that the performance of the memory for 4P and 16P decreases 0.53 MB/s (2.63%) and 13.46 MB/s (66.93%), regarding the performance achieved by scenario 1P. When extra scenarios are studied, the results are even worse, as it is expected. Both, the execution time and the memory traffic, fall dramatically for all scenarios.

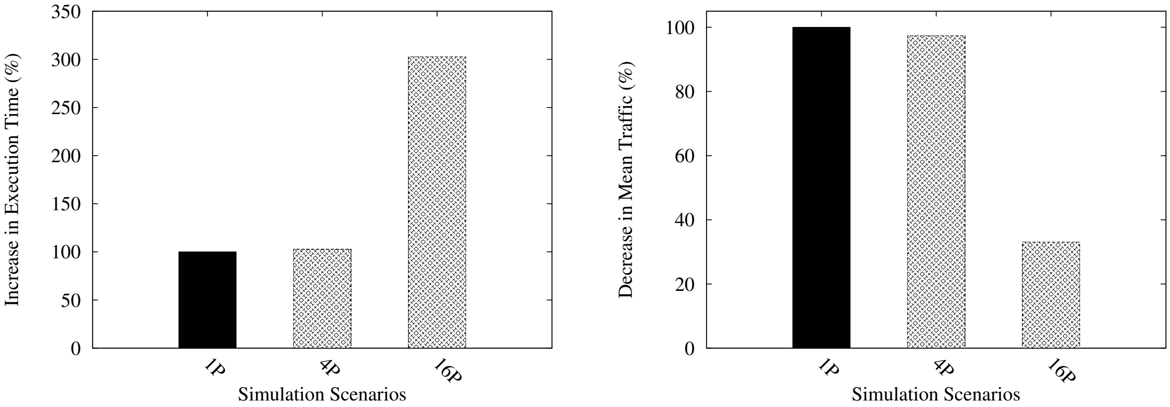
In Figure 4 we show the same results, but only for the 1P, 4P and 16P scenarios in order to improve the readability of the Figure 3.



(a) Increase in execution time with regard to scenario 1P (numerical and logarithmic scale).

(b) Decrease in mean traffic with regard to scenario 1P (numerical and logarithmic scale).

Figure 3. Performance results of the Stream application in each scenario.



(a) Increase in execution time with regard to scenario 1P.

(b) Decrease in mean traffic with regard to scenario 1P.

Figure 4. Detailed performance results of the Stream application in 1P, 4P and 16P scenarios.

The results have shown that scenario 16P using the PGAS model is always a better choice than extra scenarios that implement swapping. Of course, the performance of the unfeasible scenarios 1P and 4P is much better than the performance of the 16P scenario, but this one is the best option for those applications with high memory requirements.

5. Conclusions and Future Work

This paper presents the results of the preliminary assessment of the work in progress that is made by the ATG. Because the ATG has not completed the specification of its PGAS model yet, it is not possible to carry out a hardware evaluation. However, we have performance a simulation–driven study.

Firstly, we have developed a module of the GEMS simulator for tracking the memory requests and customizing their latency. By this module, we could simulate approximately the behavior of an application running in a cluster with PGAS support. This cluster would run any application with high memory requirements if they do not exceed the whole physical memory of the cluster, because the application will be spread out into the DRAM memory of the processors in the cluster.

As it was explained, swapping can solve the lack of memory, but the application performace falls dramatically as the lack of memory increases. This paper has introduced the PGAS model as one alternative to swapping. Results have showed that the PGAS model would never be a better option than having enough memory in the processor that runs the applications, because a lot of time would be spent in accesses to remote memories. However, it will be always better than using swapping, because the latencies of the inter–memory communications will be lower than accesses to swapping.

As future work it is interesting to keep updated of all the work that is done by the ATG, for example, the real implementation of the PGAS support, the future improvements of the HTX–board, and the specification of the HTX connector.

Acknowledgements

The authors want to give special thanks to Dr. Holger Fröning of the University of Heidelberg for his helpful comments and assistance for the development of this work.

References

[1] HyperTransport Consortium Advanced Technology Group. `http://www.hypertransport.org`.

[2] D. Bonachea. Gasnet Specification, version 1.1, Report No. UCB/CSD–02–1207, October 2002.

[3] P. Charles, C. Grothoff, V. Saraswat, and et. al. X10: An object–oriented approach to nonuniform cluster computing (OOPSLA'05). In *Proceedings of the 20th annual ACM SIGPLAN Conference on object oriented programming, systems, languages, and applications*, 2005.

[4] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley and Sons–May, 2005.

[5] D. Emberson and D. O'Flaherty. HTX Specification for HyperTransport 3.0 Daughtercards and ATX/EATX Motherboards. Technical report, HyperTransport Consortium, June 2008.

[6] Enterprise–class versus Desktop class Hard Drives, Revision 1.0, April 2008.

[7] H. Fröning, M. Nüssle, D. Slogsnat, H. Litz, and U. Brüning. The HTX–Board: A Rapid Prototyping Station. In *Proceeding of 3rd Annual FPGAWorld Conference, Stockholm, Sweden*, November 2006.

[8] HyperTransport Consortium. Hypertransport Specification, Revision 3.0, April 2007.

[9] H. Litz, H. Froening, M. Nuessle, and U. Bruening. VELO: A Novel Communication Engine for Ultra–Low Latency Message Transfers. In *Proceedings of 37th International Conference on Parallel Processing (ICPP–08), Portland, Oregon, USA*, September 2008.

[10] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform Computer. *Computer*, 35(2), February 2002.

[11] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution–driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News (CAN)*, September 2005.

[12] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. Technical report, University of Virginia (USA), 2007. `http://www.cs.virginia.edu/stream`.

[13] S. A. McKee. Reflections on the Memory Wall. In *Conference Computing Frontiers*, 2004.

[14] D. O'flaherty and M. Goddard. AMD Opteron Processor Benchmarking for Clustered Systems. Technical report, Advanced Micro Devices, July 2003.

[15] Second–Generation AMD Opteron Processor Industry Standard Server Benchmarks. `http://www.amd.com`.

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

[17] W. Stallings. *Operating Systems: Internals and Design Principles (6th Edition)*. Prentice Hall, April 2008.

[18] S. Yalamanchili, J. Young, J. Duato, and F. Silla. A Dynamic, Partitioned Global Address Space Model for High Performance Clusters. Document GIT–CERCS–08–S01, School of Electrical and Computer Engineering (Georgia Institute of Technology (USA); Universidad Politecnica de Valencia (Spain), 2008.

[19] K. Yelick and et. al. Productivity and Performance Using Partitioned Global Address Space Languages. In *Proceeding of International Workshop on Parallel Symbolic Computation (PASCO'07)*, July 2007.

Extending HyperTransport Protocol for Improved Scalability*

J. Duato*, F. Silla
Technical University of Valencia, *Simula Labs

S. Yalamanchili
Georgia Institute of Technology

B. Holden, P. Miranda, J. Underhill, M. Cavalli
HyperTransport Consortium

U. Brüning
University of Heidelberg

Abstract

HyperTransport 3.10 is the best open standard communication technology for chip-to-chip interconnects. However, its extraordinary features are of little help when building mid- and large-scale systems because it is unable to natively scale beyond 8 computing nodes. Therefore, it must be complemented by other interconnect technologies.

The HyperTransport Consortium has intensively stimulated discussions among its high-level members in order to overcome those shortcomings. The result is the High Node Count HyperTransport Specification, which defines protocol extensions to the HyperTransport I/O Link Specification Rev. 3.10 that enable HyperTransport to natively support high numbers of computing nodes, typical of large scale server clustering and High Performance Computing (HPC) applications. This extension has been carefully designed in such a way that it extends the maximum number of connected devices to a number large enough to support current and future scalability requirements, while preserving the excellent features that made HyperTransport successful and keeping full backward compatibility with it.

1. Introduction

HyperTransport 3.10 [5] (hereafter referred to as HT3.10 or simply as HT) is currently the lowest latency, highest bandwidth openly licensed standard communication technology for chip-to-chip and board-to-board interconnects. This performance leadership was achieved by:

- Minimizing packet protocol overhead
- Adopting a clock-forwarding scheme that eliminates clock recovery overhead
- Eliminating control and command signals required by other communication standards
- Reducing crosstalk and electromagnetic interference

HyperTransport was devised as an efficient replacement of the traditional system bus and it has become the interconnect of choice for on-board communications. Furthermore, HT Rev. 3.x specifications (HT3) significantly extended the scope of HT Rev. 2.0 by providing support for chassis-to-chassis – i.e. short-haul system-to-system interconnects for rack-mounted server clusters – and backplane implementations. This is achieved through the AC mode and the link-splitting features. The former supporting links up to 1m (3 feet) in length at full speed and the latter by increasing the number of HyperTransport links in and out of a device without having to increase the number of pins. Hot-plugging – also introduced with HT3 – helps to enhance HyperTransport's dynamic expansion capabilities and to improve system availability in HT-based servers and storage systems.

At link level, HT uses a lean packet protocol that carries significantly less overhead than other interconnect technologies. PCI Express, for instance, requires 2 bytes for framing and 20 percent 8b10b encoding overhead for the physical layer, 8 additional bytes for the data link layer and 12 or 16 extra bytes for the transaction layer. By contrast, HT requires no overhead for the physical layer and only 8 to 16 header bytes for the transaction/data link layer.

The characteristic that most uniquely distinguishes HyperTransport from other interconnection technologies in the market, however, is its being processor-native – i.e. integrated in the processor chip, as in the case of various AMD CPUs, as well as a number of specialty processors and SoCs from Bay Microsystems, Broadcom, NetLogic Microsystems, PMC-Sierra, Raza Microelectronics, and Tarari.

Traditionally, CPU-to-peripheral communication has been accomplished by going through a north bridge controller (competing with main memory accesses) and reaching the destination peripheral device via one of various communication standards, like PCI or PCI Express. The peripheral device would then communicate with the CPU by following the same path in reverse. By virtue of its processor-

*This work was supported by the Spanish program CONSOLIDER-INGENIO 2010 under Grant CSD2006-00046, by the Spanish CICYT under Grant TIN2006-15516-C04-01, and by program PROMETEO under Grant PROMETEO/2008/060 from Generalitat Valenciana.

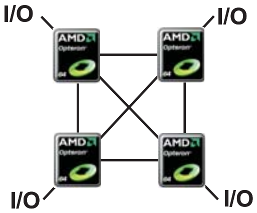


Figure 1. 4-way Opteron system. Processors are natively interconnected by HT

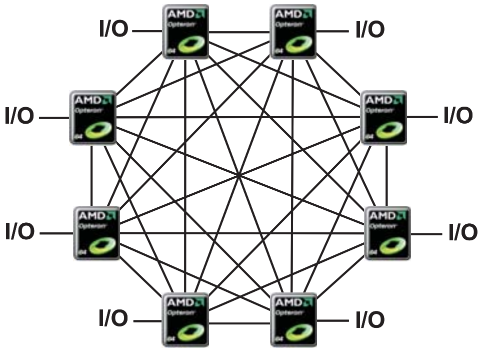


Figure 2. 8-way Opteron system. All-to-all connections are allowed by HT's link-splitting

native support, HT processor-to-peripheral communication is speeded up because it takes place on a daisy-chained, direct point-to-point link in which intermediate control functions such as north bridge controllers – with their intrinsic overhead penalties – are eliminated. Additionally, no protocol translation is required. The result of such architectural innovation is that HyperTransport collectively combines interconnect integration with high bandwidth, low latency, and low implementation cost.

HyperTransport's processor-native feature has been demonstrated to be so successful in reducing communication latency that other microprocessor manufacturers, like Intel Corporation, have modified the way their processors communicate by including this feature in their own processors. To do so, Intel developed a new proprietary interconnect technology, called Quick Path Interconnect (QPI). The first Intel processors featuring this new interconnect technology were recently introduced to market.

Thanks to its powerful features, HyperTransport has the potential to weave off-the-shelf CPU subsystems and servers into highly scalable system fabrics and clusters. Examples are the 4-way and 8-way CPU architectures proposed by AMD and shown in Figures 1 and 2. Opteron chips in Figure 1 are natively interconnected via 3 coherent HT links per processor (chips may use an additional HT link for connecting to I/O, if required). By considering that more sockets could be directly connected via HT3's link-splitting feature, and that they could be populated with 8-core Opteron processors per AMD's product roadmap, these systems may straightforwardly become 64-way systems with today's HT specifications (Figure 2).

Looking into the future, the necessity for higher performance and higher scalability solutions should make us – high-tech purveyors – alert that future High Performance Computing (HPC) platforms will have to support significantly greater scalability. This is to say that, if systems with tens of thousand of processing nodes continue to represent the elite play – i.e. limited volume opportunity with not necessarily limited profits – mid-scale systems with several hundreds of processors should progressively become common place. A growing market sector that HyperTransport Rev. 3.10 – by itself and with its present capabilities – will be increasingly unable to compete for and capitalize on.

The HyperTransport Consortium, aware of these limitations, has stimulated an extension to HT3.10 in order to overcome those shortcomings. As a result, the High Node Count HyperTransport Specification – born from the contribution of HyperTransport Consortium's high-level commercial and academic members – was recently released by the Consortium as an extension to the HyperTransport 3.10 Link specification and providing the means for HyperTransport to support large systems. This paper presents such an extension to HT. To do so, Section 2 presents the market trends that motivated such an extension. Section 3 introduces the context for the extension by defining the system model to use. In Section 4 the need for a HT protocol extension is discussed. Next, Section 5 presents some considerations taken into account inside the HyperTransport Consortium during the development of the new specification. Section 6 briefly presents the new High Node Count HyperTransport Specification at the same time that it explains why some of their features have been devised that way. Next, Section 7 provides some insights on how to implement these extensions in the current technology arena. Finally, Section 8 draws some conclusions.

2. HyperTransport limitations

As described above, HyperTransport offers some degree of scalability latitude by virtue of its link-splitting, AC-mode, and hot-plugging capability, which could enable the implementation of efficient network topologies like 3D meshes or tori. However, such network topologies, when scaling to large sizes, require routing capabilities beyond current HyperTransport ones. Specifically, HT lacks support for the following:

- Global device addressability beyond 32 HT devices, required for medium and large size clusters of processing nodes

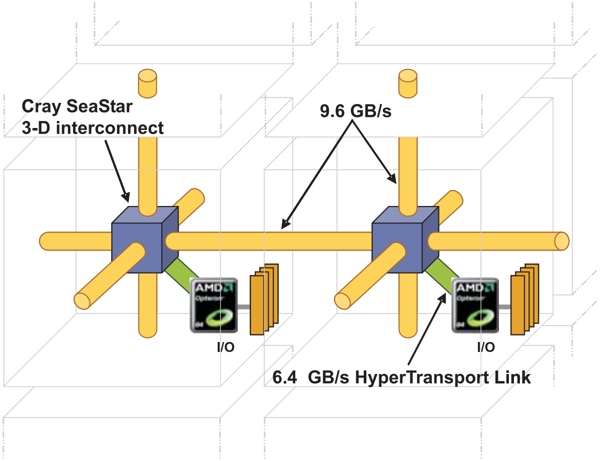


Figure 3. Cray XT4 scalable architecture. HyperTransport is used to connect Opteron chips to the proprietary interconnects

- Efficient routing in scalable network topologies
- Scalable congestion management mechanisms
- Dynamic reconfiguration of routing information after hot plug/swap/removal of components – i.e. no automatic finding of better routing paths after changing system topology

As a direct result of HT's scalability shortcomings, HPC vendors have no choice but to complement HyperTransport with other interconnect technologies. Examples are Sun Microsystems, the extinct Newisys, and Cray.

In the first case, Opteron boxes are interconnected via Gigabit Ethernet or InfiniBand, providing a non-shared memory system composed of several independent computers that communicate via some kind of message-passing protocols. Thus, the system cannot be viewed as a single large-scale system, but as an aggregation of small systems, between which communication takes place explicitly. This configuration is similar to the one traditionally used in clusters and PC farms based on Ethernet intra-system interconnect backbones, which can be further performance-accelerated by means of HT-enabled network interface Cards (NICs). However, such kind of communication model is burdened by the latency penalty introduced by the process of creating by software the messages that enable inter-processor communications. This latency penalty usually includes one or more system calls at the source and destination ends of the communication links, noticeably lowering performance. Additionally, peripheral devices cannot be easily shared among processors.

In the case of Cray's XT4 and XT5 supercomputers [4], HyperTransport provides a 6.4 GB/s direct connection between the Opteron processors and Cray's SeaStar interconnect backbone. The SeaStar interconnect is based on Cray's SeaStar2 chips and implements a proprietary protocol to directly connect up to 30,000 processing nodes in a 3D torus topology. With this approach, the cost and complexity of external switches is entirely removed and systems can be easily scaled in field. Figure 3 shows the profile of Cray's interconnect, with the AMD Opteron processors connected to the SeaStar chips via HT links (green pipes) and the SeaStar chip linking each processing node to all others via proprietary links and protocol (orange pipes). It is important to note that in addition to being able to differentiate from competitors, the main reason that compels companies like Cray to use proprietary interconnects is not necessarily to attain higher bandwidth – i.e. HT3.10's 25.6 GB/s (16-bit) bandwidth is much greater than the 9.6 GB/s required by Cray – but highly likely to compensate for HyperTransport's inability to scale up to such large system requirements.

With its proprietary Horus chip [7], Newisys proposed a different approach, which extends HT's basic functionality and enables Symmetric Multi-Processing topologies of up to 32 AMD Opteron chips (32-way) with full cache coherence support. Horus chips appeared to AMD CPUs as CPUs themselves and provided special routing for data and command packets, as well as local cache and hidden directory scheme to significantly enhance the performance of cache coherence protocols. The resulting system was a single, coherent shared-memory machine that supported implicit communication without the use of system calls, thereby significantly accelerating communication among processors. Unfortunately, scalability of Horus-based systems was limited to only 32 host nodes likely due to the unsolved inability of cache coherent node clusters to scale efficiently and, in addition, to the intrinsic inability of effective scaling of AMD's proprietary cache coherence protocol.

In summary, HyperTransport is an excellent interconnection technology that provides the highest bandwidth and lowest latency. However, HyperTransport's benefits are primarily confined to host-to-host and host-to-I/O subsystems within the realm of a single motherboard. Even with the introduction of the AC mode in HT3, the extraordinary features of HyperTransport are of little help when building large systems because HT is unable to natively scale as required by mid- and large-scale HPC applications and, therefore, must be complemented by other interconnect technologies. Note that this inability is not due to insufficient bandwidth or connectivity. In fact, as Figure 2 shows, Opteron processors may have up to 8 HyperTransport links (by using the link splitting feature), and therefore, efficient network topologies, like 3-D meshes or tori can be implemented. However, these topologies would require extending current HyperTransport scalability characteristics. This was even stated by AMD several years ago [1].

3. System model and definitions

Enhancing HT to natively support a large number of processors brings the opportunity to define a new system architecture that is scalable, flexible, and simplifies application development, all of this with minimal additional cost. As HyperTransport is a shared-memory oriented protocol, its enhancement would naturally provide a shared-memory system. However, it is well known that large-scale cache-coherent shared-memory systems have never been feasible. Large systems are message-passing flavored, instead.

Devising a message-passing HT for high node count systems would deliver little improvement over current large installations, which are already message-passing oriented. Moreover, message-passing application programmers interfaces (APIs), like MPI, may not be enough in the multicore era [3]. Additionally, other programming models, like PGAS [8] are starting to play an important role. On one hand, their performance can equal that of MPI codes and, for most humans, they are much easier to learn [2]. Also, PGAS is not less scalable than MPI and permits sharing, whereas MPI rules it out [9]. On the other hand, PGAS implements a one-sided communication model (faster than two-sided), where caching is not required and the programmer makes local copies and manages their consistency. Because of this, no cache coherence protocol is needed, except between the network interface and the processes in a node. Additionally, a one-sided put/get message can be handled directly by a network interface with RDMA support, avoiding interrupting the CPU or storing data from it.

Because of all the benefits that a global address space delivers, the Consortium decided to allow the enhanced HyperTransport to naturally provide what it provided before: a shared-memory system. Note that now this system model may be efficiently supported because cache coherence is not maintained by hardware, and is not enforced for every memory access. And it is more efficient than other PGAS implementations because HT interfaces are directly attached to the processors.

Thus, the system model in mind was based on a large number of HyperTransport devices that use the HT protocol to perform memory transactions. This will be referred to as *High Node Count Network*. We may think of this network as a network of processors, each of them with its local memory and I/O, as depicted in Figure 4. The way such processors are physically interconnected is not relevant at this point of the discusion.

Moreover, in order to devise a general and architecturally independent system model, we should define the concept of *Nest*. A nest is defined as each of the components of the High Node Count Network. A nest may be something as simple as a single CPU or something much more complex, as a motherboard containing four CPU devices, each of them containing four processor cores. Basically, the term *Nest* refers to a network-addressable entity. In Figure 4, each of the processors depicted would be a nest. Hereafter, we will use the term *Nest* instead of CPU, processor, or motherboard.

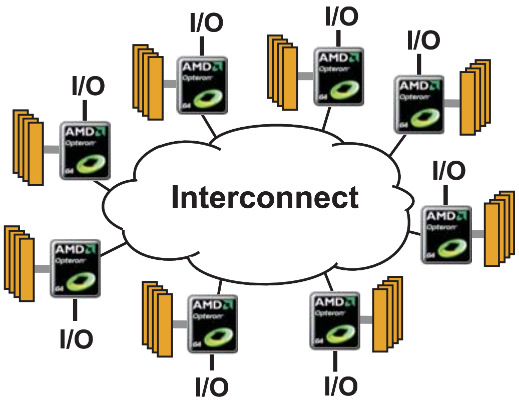


Figure 4. System model assumed in the High Node Count HyperTransport Specification

As one of the goals is providing HyperTransport with the capability of addressing a large number of devices, in the system model described above each nest will be assigned a *NestID* that unambiguously identifies it in the High Node Count Network. Note that a NestID is a network-style address and not a memory-style address. Moreover, a protocol that sets up the identifier of each nest during system initialization is required. Such a protocol, or a variant of it, may also be required after hot-plugging of components. The definition of these protocols is outside the scope of this discusion as far as a unique system-wide NestID is provided for each nest in the system.

In the system model proposed in Figure 4 – a physically distributed logically shared-memory system – each nest has access to its local memory via conventional memory buses and has access to memory belonging to other nests via HT packet exchanges. In these exchanges, the nest that sources the request must include its NestID in the packet as well as the NestID of the destination of the request, i.e., requests will carry a *SrcNest* and a *DestNest*. Once the request reaches its destination, that nest will use the SrcNest as the destination identifier for the response packet.

4. The need for a protocol extension

At first glance, it may seem that the considered distributed shared-memory model is completely compatible with current HT3 specification, as current HT requests that specify an address use 40-bit addresses with an optional 24-bit address extension and therefore, the upper part of the 64-bit global memory addresses could be used as the destination identifier. However, despite providing support for 64-bit addresses, the HT3 protocol does not provide the functionality required for a number of reasons:

1. Requests targeted to another nest may use the current address extension to identify the destination nest. However, the address extension can only be used for commands that include an address. Therefore, responses back to the source nest – which would also require a destination identifier to be appropriately routed inside the interconnect – cannot use an address extension as currently defined because current packet responses do not include an address field. Thus, responses could not be returned to the source nest. Similarly, some HT commands, like Flush, may be directed to remote nests. These commands do not include a 40-bit address in their packet and, therefore, they are not eligible to be extended by an address extension according to current HT specifications. However, when targeting these commands to a remote nest, a DestNest identifier is required in order to forward the packet to the right destination. Therefore, the address extension, as defined in the HT specification, does not provide the required support.
2. Once the target nest has accessed its local memory as the result of a remote request, it needs to know where to return the corresponding response. To accomplish this, it is necessary to include a SrcNest identifier in the requests, so that target nests use it as the destination identifier in the response. However, the source identifier extension as currently defined in the HT specifications does not allow this feature because it only includes a 16-bit address in bus-device-function format.

For the reasons mentioned above, current address and source identifier extensions, as defined in HT3, do not support the proposed system model.

On the other hand, HyperTransport's scalability limitations may not only be analyzed from the NestID point of view, but also from the interconnect topology perspective. As described in Section 4.1 of the HT3 specification, and using HT3 terminology, HT I/O fabrics are implemented as one or more daisy chains of HT devices, with a bridge to the host system at one end. Multiple daisy chains can be interconnected using bridge devices, forming a tree topology. Additionally, the host can contain multiple bridges, each supporting either a single HT I/O chain or a tree of HT I/O chains.

These topologies were conceived to attach a set of peripheral devices or controllers to a single host. The only exception are double-hosted chains, but even in this case, either one host acts as a slave and routes all of its transactions through the master host, or the chain appears logically as two distinct daisy chains – each attached to only one host bridge. Nevertheless, none of these topologies fit the requirements of large scale systems.

As can be seen, the HT3 specification does not provide suitable support for interconnecting a large number of hosts and for routing messages between them. Also, it does not provide support for identifying nests in the system, as discussed above. Consequently, either the entire specs should be modified to provide the required support, or a clever way to extend the specs while maintaining backward compatibility should be found. Modifying the specs should be ruled out as it would compromise the extensive investments already made by Consortium members in current and previous generations of HT technology. Thus, devising backward compatible HT extensions should be the recommended path to follow. After almost two years of discussions and deliberations among high-level members of the Consortium, the HT extensions have been formalized and released in the form of the High Node Count HyperTransport Specification.

5. What to include in the extension

If HT was to natively support a large number of hosts (or nests), some extensions to the protocol were required. These extensions had to be implemented in such a way that the resulting protocol was as efficient and fast as the current HT3 version. Additionally, the following goals have been considered when extending HT:

1. Analyze the ideal characteristics of extended HT while monitoring HT backward compatibility, so that existing designs could be reused in extended devices as much as possible.
2. Minimize the extensions' overhead – i.e. use of extra bandwidth, additional latency and cost. This should be done without crippling the design.
3. Optimize the extensions by taking into account that the majority of the system platforms will be rather small-scale.
4. Allow for easy addition of new features that could be deemed necessary in the future.

Moreover, the extensions of the HyperTransport protocol had to be done in such a way that support for future features commonly found in large scale systems was straightforward.

As discussed above, interconnecting a large number of hosts requires an interconnect that supports topologies other than current chains and trees, which may not be efficient for interconnecting many hosts. It also requires a global enumeration scheme across multiple hosts, so that each host knows the unique identifier of every other host in the system. Moreover, some efficient routing strategies are needed, especially in those cases where multiple physical paths exist among pairs of hosts. Therefore, in order to build a system with a large number of hosts, the following areas were identified by the Consortium as the minimum extension set to be considered:

- Addressing scheme: ability to address a much larger number of devices. This mainly affects packet formats.
- Network topology: support for topologies with much higher connectivity that will enable many more concurrent transmissions in large platforms, provide shorter paths, and provide alternative paths in case of failure.
- Routing mechanisms: a routing algorithm that supports routing messages in the above topologies. The implementation of the routing logic should be efficient both for small and large systems.

The three issues above are closely related to each other, and design decisions for one of them may significantly impact the others. Additionally, in order to develop an efficient extension of the protocol, the packet format used in the extended HT should be optimized by taking into account – at least – the addressing scheme.

At this point, the HyperTransport Consortium had to make a decision about which features to include in the future extension and how those features would look like. Note that not all such characteristics required to be phased into the new HT specification, as some of them may be implementation-dependent. For example, conveniently defining the packet format (which would necessarily imply defining the addressing scheme) may allow leaving both the network topology and the routing mechanism undefined. These two topics would be addressed/defined by manufacturers when designing their products, or could be considered at a later time for inclusion in subsequent HT specifications. Additionally, this would open up market opportunities for the companies member of the HyperTransport Consortium at the same time that allow these characteristics to mature before developing more advanced specifications. For these reasons, the High Node Count HyperTransport Specification recently released focused on defining the minimum set of extensions to the HT protocol that allow HyperTransport to efficiently support large system sizes. Network topologies and routing mechanisms were classified as implementation-dependent. However, the use of distributed routing is assumed in the interconnect. In this way, packets exchanged between nests will be kept small. Additionally, the amount of routing information contained in them is constant and therefore independent of the path between the source and the destination nests. This simplifies decoding the packets. Moreover, adaptive routing may be used in the future. Finally, other issues, like the required protocols that set up the identifier of each nest during system initialization were also defined as implementation-dependent.

6. The high node count specification

This section describes and explains the extensions to HT3 intended to allow HyperTransport to natively provide support for high node count environments [1].

One of the topics addressed by the HyperTransport Consortium while defining these extensions was the maximum number of nests to be supported. The addressing scheme and the new packet format depended on this. On one hand, the maximum allowed number of nests should be large enough to support current and future HPC needs. Actually, it should be large enough so that the new specification under development would not require to be modified in the near term. This system size is probably larger than the market may require (at least for quite a while), thus smaller systems would be paying a performance penalty for HT being able to scale up to those large sizes. On the other hand, defining relatively small NestIDs, optimized for smaller, more popular systems, would likely impose the need for further specification extensions in just a few years to satisfy market-driven scalability requirements.

Ultimately, the decision made by the Consortium was to support multiple system sizes. This would keep complexity low for small system configurations while enabling HT to scale up as needed. Or course, it was taken into account the space availability in the format of the extended packets as well as cost and performance constraints.

With the NestID length defined, the format of the new specification extensions was also defined. These extensions are based on a new control doubleword: the NestID extension, which allows nests in the system to identify themselves and also to univocally address other nests, and therefore, it is one of the key elements of the specification extensions.

The format of the new control doubleword allows an easy decoding of the extensions, aligning the new specification with the second goal in Section 5 that established that extensions should keep overhead as low as possible. Additionally, the presence of NestID extensions in a request packet is fully compatible with other extensions such as source identifier or address extensions – remember that the first goal mentioned in Section 5 established the need for full backwards compatibility.

Response packets may also be extended by NestID extensions. In fact, when a request reaches a nest, it will probably have to generate a response packet intended for the nest that initially created the request. Therefore, in order to properly forward the response to the right source nest, a NestID extension must be used. Obviously, the DestNest

[1] It should be noticed that the HT Consortium has decided not to make the new extensions available to the general public, so that they will only be usable by HT Consortium Promoter and Contributor members. Therefore, because of confidentiality constraints, the description in this section will not provide the complete details of the new extensions.

identifier used in a response packet is copied from the SrcNest identifier of the corresponding request packet.

The exact location of the NestID extension in a request packet is not accidental, but designed so to improve system performance because when a nest sends a request to another nest, the DestNest identifier needs to be decoded in order to forward that packet to the target nest. Therefore, properly locating the required information in the request packet reduces routing time. Additionally, the location of the DestNest identifier in response packets has been carefully designed so that not only routing time is minimized but also routing of both request and response packets in the fabric is kept efficient.

Moreover, packet overhead is minimized for small systems. The NestID extension has been meticulously designed for small-scale systems, where the new format is heavily optimized. Actually, this was the third goal to keep in mind mentioned in Section 5. In small systems, packet length optimization translates in request packets being 20% shorter than non-optimized packets. Additionally, they are 42% shorter if compared with packets for large systems.

Note that some extra configuration information is needed in a nest so that it properly interprets the new extensions. The required configuration information is located in the High Node Count Capability Block, the other master piece of the specification. Unfortunately, because of confidentiality constraints, the format and usage of this capability block cannot be disclosed.

7. Protocol extension implementation

The High Node Count HyperTransport Specification can be implemented in two different ways, each of them having a different impact on cost, benefits, and market opportunities. The following discussion is independent of the maximum global memory size and thus it is equally valid for any of the choices mentioned above.

7.1. Native implementation

The implementation option yielding the best system performance would be modifying the HT logic within the Opteron processors, in order to enable them to intercommunicate directly using the proposed extension and to avoid the use of any external logic, thereby minimizing latency and maximizing bandwidth. Additionally, the embedding of such HT extensions into Opteron processors interconnected by large inter-processor networks would require embedding in the Opteron architecture some kind of routing logic also, so that HT packets are forwarded from the given source to the proper destination. This approach would be similar to the Alpha 21364 processor [6], which integrated a router function and allowed processors to be interconnected by a 2D torus with a maximum of 128 processors. Figure 5 shows a similar scenario of a 2D mesh with 16 processors. In this case, processors would require only four HT links for complete interconnection. If more links are available – e.g. by means of HT3 link-splitting capability – more efficient topologies like 3D meshes or tori could be deployed. These topologies require 6 links per processor. This implementation option is certainly the most effective long-term for best performance and lowest implementation cost. However, it is not the most ideal time-to-market wise and cost wise, as modifying the processor's logic would be quite time and resource laden.

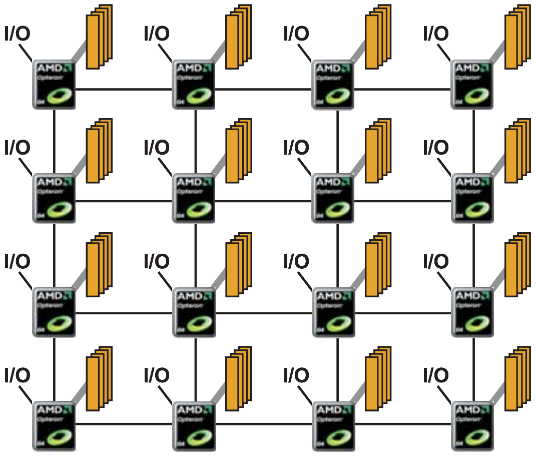


Figure 5. 16-way 2D-mesh system. Processors have embedded protocol extension and routing logic

7.2. Bridged implementation

Actually, embedding these new HT extensions does not necessarily impose changes to the processor design. In fact, an alternative, more flexible and time-to-market friendly choice would be the implementation of the extended HT functionality in external logic – i.e. in a bridge chip – whose main purpose would be translating requests and responses from one version of the protocol to the other – i.e. a chip that implements standard HT3 on one side and extended HT on the other. The logic inside this chip could be designed in such a way that it detects a current HT chain on one side and a large HT network on the other. The chip should also incorporate some routing capabilities in order to forward messages to their intended destination. Figure 6 shows such a system, composed of 16 processors that communicate with each other through such bridge chips.

The technical and commercial advantages of such external logic solution greatly outweigh its disadvantages compared to the first option. The bridge implementation would allow HT extensions' functionality to mature before full

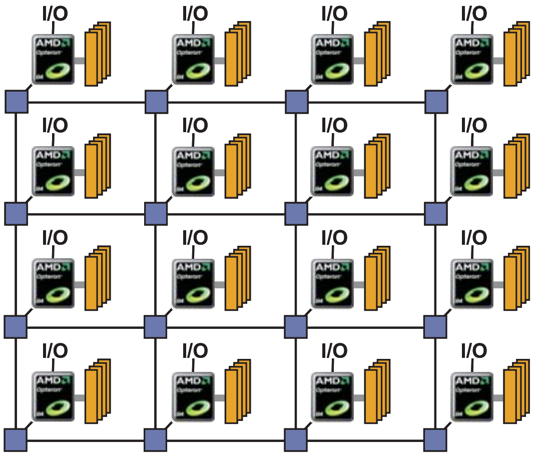


Figure 6. 16-way 2D-mesh system. Protocol extension and routing logic integrated in the bridge chip

integration into the CPU. It would also open up interesting market opportunities for HyperTransport Consortium member companies and the HT technology ecosystem. The trade-off will be a slightly higher latency and higher system cost (1 bridge chip per nest).

The implementation of the bridge chip requires including a matching unit that associates returning responses to previously sent requests. In the case for the bridge chip at the source end, it must first extend the original request by using the NestID extensions. Once the extended packet arrives at the destination bridge chip, it must aggregate requests from several sources by translating them into local requests which are uniquely defined by the combination of UnitID and SrcTag. Thus, the destination bridge must store the value of the UnitID, SrcTag, and SrcNest fields in the incoming packet along with the new local values for UnitID and SrcTag, so that when the response comes from the local chain, the bridge can associate it to the initially received request and translate back those fields to the initial ones before sending back the response to the source end. Once the response is received at the source end, the source bridge will remove the NestID extensions to deliver a final response to the initial source of the request.

8. Conclusions

Large computing systems are interesting because of their aggregate computing power and overall memory capacity. These large systems require high bandwidth, low latency interconnect technologies for inter-processor communication. HT3 has the capability and latitude required by these systems, except for its inability to scale appropriately. Fortunately, it is feasible and cost-effective to infuse such needed scalability into HyperTransport, as described by the High Node Count HyperTransport Specification.

The bridged implementation of such specification may be quick-to-market, not requiring changes to current processor architectures, but will not provide the best latency performance. On the contrary, the integration of the new HT extensions in the processor architecture would allow future Opteron chips to be powerful building blocks for systems of any viable size and scale.

The proposed protocol extensions are fully compatible with the HT3 specification. Packet ordering is preserved by processing packets in compliance with current ordering rules. On the other hand, flow control is also complied with because extended packets will use current buffers and therefore no change is required in the NOP flow control packets. Nevertheless, buffer size should be enlarged in order to store the extended packets.

Regarding protocol overhead, the proposed extensions add a few bytes to current HT packets, but only in the case of packets targeted to remote processors. Instead, packets traveling through the local HT chain and intended for local I/O do not require to be extended and, therefore, no overhead is added. Moreover, for those cases where overhead is a primary concern, the new protocol extension has been optimized for smaller system sizes.

References

[1] A. Ahmed, P. Conway, B. Hughes, and F. Weber. Hammer Shared Memory Multi Processor Systems. *HotCHips 14*, August 2002.

[2] W. Camp. Computer architecture: Opportunities and challenges for scalable applications. *Sandia CSRI Workshop on Next-generation scalable applications: When MPI-only is not enough*, June 2008.

[3] E. Chow. Non-MPI Apps: Why we don't use MPI-only. *Sandia CSRI Workshop on Next-generation scalable applications: When MPI-only is not enough*, June 2008.

[4] Cray Inc. Cray XT5 Specifications. *http://www.cray.com/Products/XT/Product/Specifications.aspx*, 2008.

[5] HyperTransport Technology Consortium. HyperTransport I/O Link Specification Revision 3.10. *available at http://www.hypertransport.org*, 2008.

[6] S. S. Mukherjee, P. Bannon, S. Lang, A. Spink, and D. Webb. The alpha 21364 network architecture. *IEEE Micro*, 22:26–35, January 2002.

[7] R. Oehler and R. Kota. HORUS - Enabling large scale, 32-way Opteron Enterprise Servers. *HotCHips 16*, August 2004.

[8] S. Yalamanchili, J. Young, J. Duato, and F. Silla. A dynamic, partitioned global address space model for high performance clusters. *Available at http://www.cercs.gatech.edu/tech-reports/tr2008/git-cercs-08-01.pdf*, 2008.

[9] K. Yelick. Programming models: Opportunities and challenges for scalable applications. *Sandia CSRI Workshop on Next-generation scalable applications: When MPI-only is not enough*, June 2008.

Run-Time Reconfiguration for HyperTransport coupled FPGAs using ACCFS

Jochen Strunk*, Andreas Heinig*, Toni Volkmer*, Wolfgang Rehm* and Heiko Schick†
*Chemnitz University of Technology
Computer Architecture Group
Email: {sjoc,heandr,tovo,rehm}@cs.tu-chemnitz.de
†IBM Deutschland Research & Development GmbH
Email: schickhj@de.ibm.com

Abstract

In this paper we present a solution where only one FPGA is needed in a host coupled system, in which the FPGA can be reconfigured by a user application during run-time without loosing the host link connection. A hardware infrastructure on the FPGA and the software framework ACCFS (ACCelerator File System) on the host system is provided to the user which allow easy handling of reconfiguration and communication between the host and the FPGA. Such a system can be used for offloading compute kernels on the FPGA in high performance computing or exchanging functionality in highly available systems during run-time without loosing the host link during reconfiguration.

The implementation was done for a HyperTransport coupled FPGA. The design of a HyperTransport cave was extended in such a way that it provides an infrastructure for run-time reconfigurable (RTR) modules.

1. Introduction

With the emergence of dynamically and partially reconfigurable (DPR) FPGAs, the possibility to reconfigure partially reconfigurable regions (PRR) with run-time reconfigurable modules has appeared. This feature enables FPGA customers to change the design of a certain region of the FPGA during run-time while maintaining the full functionality of the remaining part. This new degree of freedom also facilitates system designers to develop single FPGA chip solutions where additionally required hardware, e.g. a peripheral interconnect, is also located inside the FPGA.

For host coupled FPGA systems, solutions are conceivable where a static part of the FPGA covers the host interface core and the remainder of the device can be reconfigured during run-time with one or more user specific application modules.

Such a FPGA system would offer continuous host link connectivity during the time of partial reconfiguration and would not depend on exclusive hot plug solutions, where the board, the BIOS and the operating system must support hot plug functionality, which is currently not the case for standard motherboards with operating systems like Linux and Windows.

Two distinct options for connecting FPGA accelerators to a host system do exist, either via a peripheral bus (e.g. PCI Express) or processor bus. Well suited for direct processor bus coupled FPGA systems are the AMD CPUs because of the open standard and low latency HyperTransport (HT) protocol.

Sharing the resources of a single FPGA between users is also imaginable. In a multi user or multi process environment several modules could be run simultaneously on the same FPGA if resources are sufficient.

Partial reconfiguration offers the chance of reducing implementation time of FPGA designs (rapid prototyping) if supported by the FPGA synthesis tools. Already functional parts could be left on the FPGA and only functionality under test is exchanged. It should be noted that this requires a strict modular overall design.

For highly available and real-time processing systems with host connection, run-time reconfiguration enables to exchange or to add functionality during system operation.

In the field of high performance computing nodes with FPGAs used as accelerators, run-time reconfiguration can be utilized to change offload compute kernels and to share FPGA device capacity.

Using FPGAs for acceleration, due to the creation of specialized processing engines utilizing the highly parallel nature of FPGAs, can lead to a significant reduction of compute time. A speedup of more than 50 compared to a CPU was achieved by Woods et al. [1] accelerating a Quasi-Monte Carlo financial simulation.

Zhang et al. [2] gained a speedup of 25 for another Monte-Carlo simulation.

To run such compute kernels on a single chip FPGA solution making use of the run-time reconfigurability, three main components have to be provided to the user. The first one is the operational infrastructure for running run-time reconfigurable modules (RTRM) on a host interface on the same FPGA. The second part consists of a framework which allows the user to build its own RTRMs. Last but not least, an generic interface must be provided to a user which offers functions for reconfiguration and communication between the host and the RTRM located inside the FPGA.
The rest of the paper is organized as follows:

Section 2 is devoted to related work. In section 3 capabilities of run-time reconfigurable FPGAs and the principles of creating partial configuration bit stream files are shown.

Section 4 describes the run-time reconfiguration support for a FPGA directly connected to AMD's processor bus. The enhancement for a HyperTransport cave implemented as host interconnect is shown. The infrastructure needed on the FPGA for the support of run-time reconfigurable modules and their creation is presented.

The software framework provided to the user is based on ACCFS (Accelerator File System) which is explained in section 5.

As proof of concept we have implemented two distinct compute kernel offload functions as run-time reconfigurable modules in section 6. The first RTR module acts as an offload function which finds patterns in a bit stream (pattern matcher) and the second module a Mersenne Twister generates pseudo random numbers at high output frequency.

Section 7 concludes the results of this paper.

2. Related Work

Utilizing RTR capabilities of FPGAs and building CPU coupled systems have been proposed under various aspects. Some are dealing with internal communication structures while others concentrate more on system integration.

A tool-flow for homogeneous communication infrastructure for RTR capable FPGAs was presented by Hagemeyer et al. [3] built upon the Xilinx design flow. In contrast Koch et al. [4] designed a framework named ReCoBus-builder without applying Xilinx's partial reconfiguration flow. Only Virtex-II and Spartan-3 FPGA are supported by the builder so far. Switch architectures with routers between RTR modules have been examined also in [5] [6].

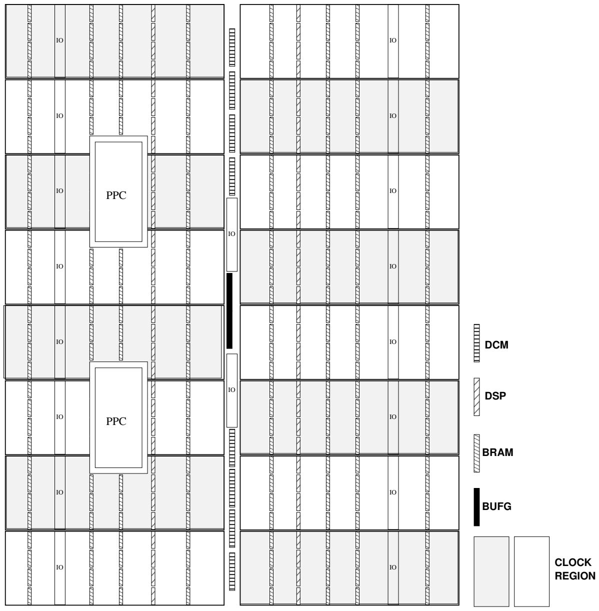


Figure 1. Schematic view of Xilinx XC4VFX60 FPGA

On the matter of integration of FPGA modules or threads for embedded systems different models have been proposed. ReconOS [7], a real time operating system implemented with static FPGA threads, is based on memory mapping and is used in embedded systems. Another model, BORPH [8], is based on the UNIX IPC mechanism and utilizes the integrated PowerPC as host.

For the integration of host coupled accelerators we proposed and implemented the Accelerator File System (ACCFS) [9]. This framework is based on the concept of a virtual file system. We have already shown the integration of the Cell/B.E. processor. In this paper we will show that ACCFS is best suited for the integration of FPGAs, even RTR capable FPGAs, into a host system.

3. Run-Time Reconfiguration on FPGAs

This section addresses the conditions which must be fulfilled, when using the feature of run-time reconfiguration on Xilinx FPGAs. These are important for the implementation of a HT cave which supports run-time reconfigurable modules.

3.1. Dynamic Partial Reconfiguration for FPGAs

This subsection is devoted to the dynamic partial reconfiguration (DPR) of Virtex-4 and Virtex-5

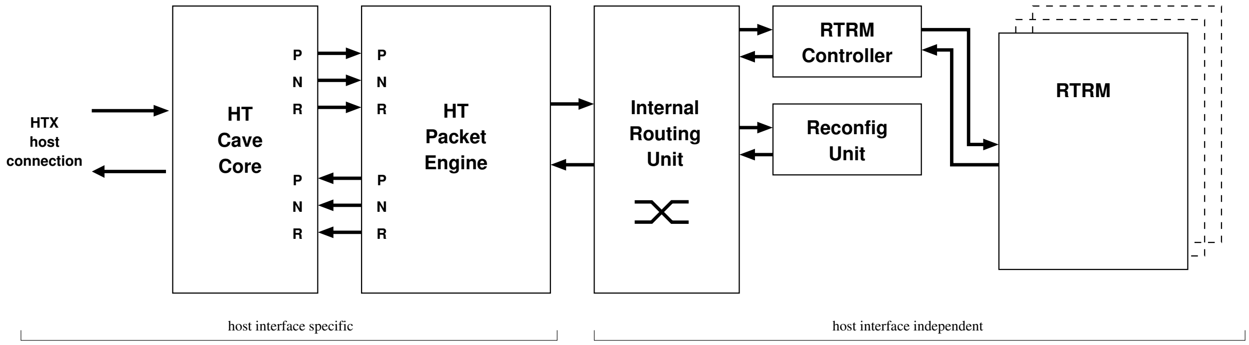


Figure 2. Infrastructure of HT cave with RTR support

FPGAs [10] from Xilinx which is one of the few manufacturers which offer DPR. The granularity of a partially reconfigurable region (PRR) is directly related to the configuration frames [11], which describe the function or contents of the slice containing LUTs or block RAM for example. The granularity in the height of a PRR matches the height of a clock region for Virtex-4 (16 CLBs) and Virtex-5 (20 CLBs). In the horizontal direction a PRR must begin with an even and end with a odd slice number. Figure 1 is a schematic view of the Virtex-4 XC4VFX60 FPGA used for the implementation of a HyperTransport cave supporting RTRMs described in the next sections. Note that we have a total of 16 clock regions available. For run-time reconfiguration three different interfaces are available, which are able to read the configuration bit stream of a RTRM. One of these is the JTAG port, which is a bidirectional serial host-clocked link. It is generally used for prototyping and debugging, working up to the speed of 24 MHz with available JTAG programmers. Another mode is SelectMAP, which works on a parallel interface connected to the physical IO pins of the FPGA achieving high throughputs. The third variant is the internal configuration access port (ICAP). It is an internal version of the external SelectMap working at a clock speed of up to 100 MHz at 32 bits width. For host coupled systems it is best suited, because it does not depend on external IOs and allows the shortest reconfiguration time.

3.2. RTR Modules and Design Flow

In this subsection the design flow is introduced for the creation of run-time reconfigurable modules. It also covers challenges and limitations of dealing with RTRMs. The design flow is based on ”Module based Partial Reconfiguration” [12] for Xilinx FPGAs. As a first step the HDL sources must be assigned either to the static part, which is constantly available during run-time, or the dynamical part. All communication between the two distinct parts has to go through hard macros, also known as bus macros. Clock resources and the hard macros must be instantiated in the HDL source and need to be assigned to a fixed location inside the FPGA. The run-time reconfigurable module itself is only instantiated as a black box, whose interface (entity) can not be changed during run-time. This means that a common interface must be created if other modules should be loaded in the partially reconfigurable region (PRR). The location and size of a PRR must be specified for the place and route process using the ”AREA_GROUP” constraint. It should be noted that for standard static design, neither PRR nor bus macros need to be specified. The same applies for the definition of the location of clock resources. To conduct the partial reconfiguration flow a patch is provided by Xilinx which must be applied to the standard synthesis tools.

4. Run-Time Reconfiguration Support for a HyperTransport Cave

For a single FPGA chip solution connected to a host utilizing HyperTransport as interconnect, it is essential not to loose the link during the time of the reconfiguration of a RTRM. This implies that the HyperTransport IP-core implementing a HT cave must be kept inside the FPGA as static part. Hot plugging is not supported so far by off the shelf systems. Even if the hardware is capable of handling such requests, most operating systems do not support this. Other RTRMs inside the FPGA would suffer also from the link loss. For that reason the HyperTransport cave is kept in the static region. In this section the enhancement of a HT cave is shown which provides an infrastructure for dealing with RTRMs.

```
entity rtrm is
port (
crq_c2m_addr      : in  STD_LOGIC_VECTOR(31 downto 0);
crq_c2m_data      : in  STD_LOGIC_VECTOR(31 downto 0);
crq_c2m_rq_valid  : in  STD_LOGIC;
crq_c2m_stop      : in  STD_LOGIC;
crq_m2c_data      : out STD_LOGIC_VECTOR(31 downto 0);
crq_m2c_rp_valid  : out STD_LOGIC;
crq_m2c_stop      : out STD_LOGIC;
crq_c2m_wr_rd     : in  STD_LOGIC;

mrq_m2c_addr      : out STD_LOGIC_VECTOR(31 downto 0);
mrq_c2m_data      : in  STD_LOGIC_VECTOR(31 downto 0);
mrq_c2m_rp_valid  : in  STD_LOGIC;
mrq_c2m_stop      : in  STD_LOGIC;
mrq_m2c_data      : out STD_LOGIC_VECTOR(31 downto 0);
mrq_m2c_rq_valid  : out STD_LOGIC;
mrq_m2c_stop      : out STD_LOGIC;
mrq_m2c_wr_rd     : in  STD_LOGIC;

c2m_clk           : in  STD_LOGIC;
c2m_res_n         : in  STD_LOGIC;
m2c_intr          : out STD_LOGIC
)
end rtrm;
```

Figure 3. *Entity of RTRM*

4.1. RTR Infrastructure

A run-time reconfigurable infrastructure for a HyperTransport cave has to provide a communication mechanism between the host and the RTRM and perhaps between RTRMs themselves. It also has to comply to the rules of partial reconfiguration and the partial design flow. To ease porting the infrastructure to other interconnects, e.g. PCI Express, the functionality which must be implemented for a RTR infrastructure should be divided into two parts. One covers the host interconnect specific functions and the other the host interconnect independent portions.

The infrastructure designed for a HyperTransport cave supporting RTRMs consists of two host interface specific, i.e. HT Cave Core and HT Packet Engine, and four host independent parts, an Internal Routing Unit, a RTRM Controller, a Reconfig Unit and one or more RTRMs. The design of this infrastructure of a HT cave with RTR support is depicted in Figure 2. The HT cave design for the HyperTransport interconnect originates from [13]. The task of the HT Package Engine is to decode the HT packets coming from the host and to convert these into appropriate actions targeting the units inside the FPGA. This includes the creation of responses to requests from the host by injecting valid packets to the HT Cave Core. The Internal Routing Unit routes requests to and from internal units, e.g. RTRM Controller and Reconfig Unit. For fast run-time reconfiguration of RTRMs it is recommended to make use of an internal reconfiguration port. This is done by the Reconfig Unit which controls the internal configuration access port (ICAP) for Xilinx FPGAs. The Reconfig Unit itself is controlled by the vendor specific driver on the host, which validates if requests concerning the creation of new RTRMs can be served. The allocation of RTRMs to available RTR regions is also decided by the host system.

4.2. RTRM

Each RTRM has its virtual address space which is implemented 32 bits wide. This means that a global address space is not divided between the RTRMs using fixed addresses. It would be very difficult to resolve a request when two RTRMs demand the same fixed physical address for their memory regions which are exported to the user application using an entry in the virtual file system implemented on the host system.

The interface (entity) of a RTRM serves as an interconnect to the RTRM controller. Communication in both directions, i.e. controller requests (crq) and module requests (mrq), are possible using a stop and valid protocol. The entity of a RTRM in VHDL is shown in Figure 3.

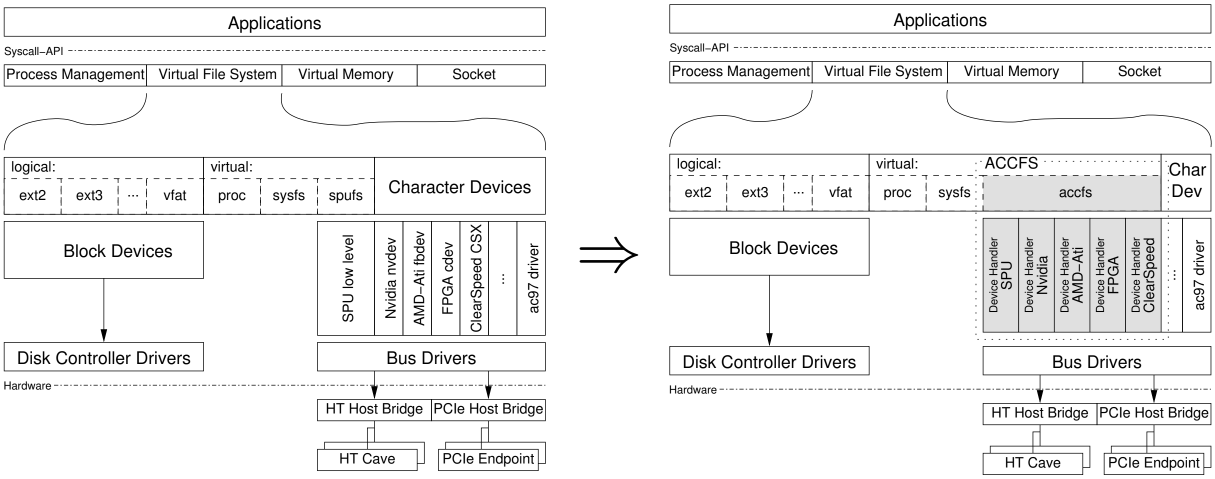
4.3. RTRM-Controller

The RTRM controller handles requests coming from the HT Core originated by the user application or from the RTRM itself. It converts physical addresses for directly accessing the RTRM, e.g. through direct load and store operations from the host to virtual RTRM addresses. The controller can also be used for RTRM to RTRM communication if desired.

4.4. Framework for a HT Cave supporting RTR

For generating the static part, i.e. the HT cave with RTR support, and the dynamical RTR modules, scripts are provided. The intention is to ease the creation of RTRMs for an application developer who is not so familiar with FPGA IP-core designs and run-time reconfiguration.

The top VHDL module is synthesized with the instantiated HT core, the HT packet engine, the internal routing unit, the RTRM controller and the Reconfig Unit by the `build_static` script. The RTRM module is only instantiated as a black box module. Then the static part is implemented with the partial flow option. While the user constraints file (ucf) normally contains location (LOC) constraints for external IO pins, this file must also contain additional LOC constraints for the PR flow covering all clock resources, in particular



(a) **Without ACCFS**

(b) **ACCFS: Common Generalized Interface**

Figure 4. ACCFS - Layered Structure

clock buffers and digital clock managers (DCMs). The resulting placed and routed design represents the basis for creating the dynamic configuration bit stream.

For the dynamic part, the user must supply an interface-compliant RTR module with the top entity name ”rtrm” and a description of the file entries which should be exported by ACCFS. This description consists of the type, the size and the virtual address which are essential to export the functionality to the user application. This additional information is added later to the final ACCFS configuration bit stream as a part of the header.

Using the `build_dynamic` script the user-supplied RTRM module is implemented with the partial flow option. Next, the Xilinx tools `PR_verify` and `PR_assemble` are used to build the partial bit stream file. Then the ACCFS RTRM bit stream file is created by adding header information containing the HT cave version, the FPGA board version and the user-supplied module description. Due to this header information, it is possible to transfer ACCFS RTRM bit stream files to other hosts which contain the same FPGA accelerator board and use the identical HT cave version.

5. ACCFS for Host System Integration

Different solutions exist for operating system integration of a FPGA. For example, BORPH [8] or ReconOS [7] provide a hardware process/thread abstraction which coexist beside ”normal” software processes. However, deep modifications of the Linux kernel are necessary to implement them. Furthermore, it is required to run Linux on the processing unit of the FPGA.

Due to the mentioned disadvantages we proposed and implemented the Accelerator File System (ACCFS) [9]. In this section we describe the major aspects of ACCFS for the integration of FPGAs into a host system. We start with a brief overview in subsection 5.1. Subsection 5.2 depicts the concepts of ACCFS. Thereafter, we present the integration steps for the HT-coupled Virtex-4 card in subsection 5.3.

5.1. Overview

ACCFS is an open generic system interface for the integration of different accelerator types into the Linux operating system. It is based on SPUFS (Synergistic Processing Unit File System) [14] which is used to access the Synergistic Processing Units of the Cell/B.E. processor. The goal of ACCFS is to replace the different character device based interfaces (cf. Figure 4a) with a generic file system based interface (cf. Figure 4b).

In the case of character devices the hardware functionalities are usually exported through the `ioctl` system call. However, this system call has the disadvantage of a non-standardized interface. Hence, the usage differs from one vendor to another.

In contrast, ACCFS defines a well structured `ioctl`-free interface based on a Virtual File System (VFS) approach. In Figure 4b the parts of ACCFS are shown as gray boxes. To be customizable when integrating new hardware ACCFS was split into two parts. Part one (”accfs”), provides the user interface, and the other parts (”device handlers”) integrate the hardware.

Device vendors as well as library programmers benefit from ACCFS. Only the lowest abstraction levels have to be implemented inside the device handlers. The whole user interface is already provided by accfs. Thus integrating a new accelerator requires less device driver programming costs. The library programmer benefits from basic design concepts introduced in the next subsection.

5.2. Basic Concepts

In the previous subsection we already described the concept of **functionality separation** which eases the integration of new hardware. Another concept was the usage of a **VFS** which maps the accelerator to normal files. This enables us to implement a `ioctl` free and hence a nearly standard conform approach. All supported file I/O operations are POSIX conform with some exceptions. For example, it is not possible to write beyond the end of a file or to change the position of the current file pointer on some files.

ACCFS is designed to support the **virtualization** of the accelerators. We abstract the *physical accelerator* with an *accelerator context*. The context is the operational data set of the accelerator. It includes all information which are necessary to describe the current hardware state in such a way that the operation can be interrupted and resumed later without data loss. During the interruption another context is able to utilize the physical hardware. Virtualization optimizes the resource usage of the accelerators. Contexts which do not make use of the hardware at a given time are not scheduled on the physical accelerator.

Each context is bounded on a directory inside the VFS under the ACCFS mount point. The files inside this directory represent the functionalities of the accelerator. To support reconfigurable hardware the file set is **dynamically exported** and can change during runtime. For example, an additional memory can be exported due to reconfiguration of the FPGA with a new RTR module.

To interact with the accelerator several methods are feasible. One is the simple memory mapped IO with standard load/store machine instructions. In this direct memory access (DMA) method the host is the active part who issues a read/write for every memory access. Another method is DMA-bulk transfer. Here the accelerator needs a DMA unit capable of moving the data asynchronously to the host processor execution. In cases where the accelerator is able to initiate these transfers by itself, the DMA unit has to handle virtual memory managing issues, too. However, not every accelerator supports virtual memory. For this reason we restrict our solution to **host initiated DMA**, where the host setups the memory management unit and initializes the data transfer. The actual data movement is done asynchronously by the accelerator.

```
struct accfs_vendor
{
  int vendor_id;
  int (*create)(...);
  int (*destroy)(...);
  int (*run)(...);

          ...

  ssize_t (*memory_sdma)(...);
  ssize_t (*config_read)(...);
  ssize_t (*config_write)(...);
};
```

Figure 5. *struct accfs_vendor*

Finally, ACCFS supports **asynchronous context execution** based on an explicit synchronization primitive. This concept eases the software development because multi-threading is not required when using multiple accelerator units. Every context runs asynchronously to the host system. The finish status can be read through a ”status” file.

5.3. FPGA Support

To support HyperTransport coupled FPGA boards within ACCFS a new device handler has to be written. This device handler has to provide the structure *accfs_vendor* (cf. Figure 5). The first four entries has to be set and the others are optional. For example, if the callback function for the DMA-bulk transfer is not set (`memory_sdma`), accfs will use the internal routines to copy the data from/to the FPGA.

Further details of the device handler implementation are described in the reset of this subsection with the help of the typical FPGA usage model shown in Figure 6. An example code fragment using this model is shown in Figure 7 of section 6, where the case study is conducted.

5.3.1. Create Context. ACCFS enforces an accelerator based programming model. The main program is running on the host system and executes the compute kernel on the accelerator. To outsource such a kernel the application has to create a context by invoking the `acc_create` system call.

Currently our device handler does not support virtualization hence we can only exclusively provide the FPGA to one application.

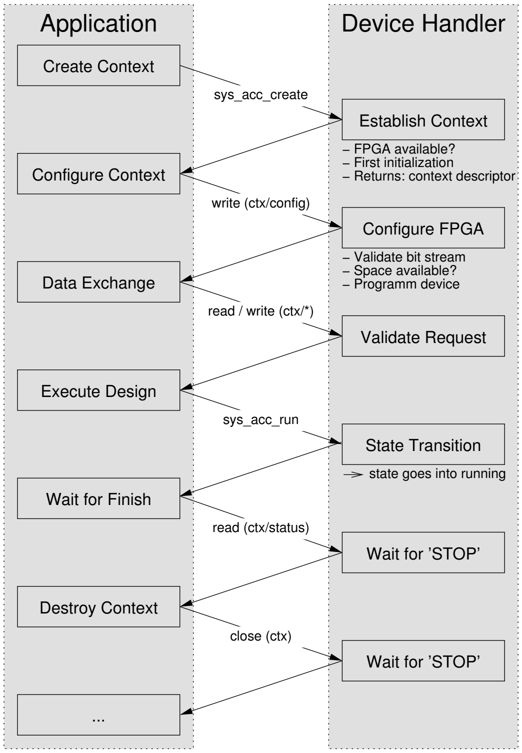


Figure 6. FPGA usage

5.3.2. Configuration. Loading the design is triggered by a `write` system call on the "config" file. The data has to be a valid ACCFS bit stream. To ensure that the RTRM matches the RTR infrastructure we provide a tool chain which generates such a bit stream file by writing a special header before the bit stream data. The header contains all necessary information describing the bit stream such as the RTR capable core and FPGA board version. If the validation is successful, the FPGA is programmed with the configuration bit stream file using the internal reconfiguration port ICAP for Xilinx FPGAs or through an external JTAG programming device, e.g. Xilinx USB platform cable. After a successfully configuration the exported memories of the FPGA design are visible in the context directory.

5.3.3. Data Exchange. The access of FPGA memory is possible with the `read` and `write` system calls. In a later development stage these calls start a host initiated DMA-bulk transfer. If the memory is exported as memory mapped IO, the `mmap` system call will map the memory into the address space of the application.

The "data exchange" operation is always possible after the configuration no matter whether the context is in execution or not.

5.3.4. Execute Design. To start the RTR module the application has to invoke the `acc_run` system call. The execution happens asynchronously, meaning that `acc_run` returns immediately. This enables the application to execute more than one context in parallel without using threads.

When the application needs to check the execution status, e.g. if the FPGA has finished its work, the "status" file can be read. Unless this file was opened with *O_NONBLOCK* the `read` system call will block until the RTRM inside the FPGA has finished its task.

5.3.5. Destroy Context. When the application closes the file handle returned by `acc_create` the context gets destroyed.

6. Case Study of RTRMs for a HT Cave supporting RTR

6.1. Overview

As proof of concept we designed two different compute kernels as RTRMs for a HyperTransport coupled Xilinx Virtex-4 FPGA plug-in card [15]. The user program using the virtual file system ACCFS is able to configure and access the two RTRMs consecutively during the run-time of the user program at the time when they are needed. The first RTRM acts as an offload function which finds patterns in a byte stream (pattern matcher) and the second module, a Mersenne Twister, generates pseudo random numbers at high output frequency. For generating the appropriate partial bit stream files of the RTRMs the framework presented in subsection 4.4 is applied.

As hardware for the host system an Iwill DK8-HTX motherboard with two Opteron processors is utilized. The pre-installed BIOS is replaced by a customized LinuxBios version to get the HTX-card enumerated by the host system. The FPGA on the HTX card is a Xilinx Virtex-4 XC4VFX60.

6.2. RTRMs - Pattern Matcher and Mersenne Twister

Two RTRMs have been implemented, which are described in this subsection, a pattern matcher and a Mersenne twister based on the MT19937 algorithm [16].

The latter uses the MT32 [17] implementation, which is able to provide a new 32 bits pseudo random number each clock cycle. When the host performs a

```
int matcher_run (void * search_db_in, int db_size
    void * patterns_in, int pattern_count,
    void * results_out, int results_size) {
  int ret;
  char bufstatus[12];
  // create context of our static FPGA design
  int fd_ctx = (int)acc_create(”example”, V_ID,
      D_ID, 0750, NULL);

  // configure the design
  int fd_cfg = openat(fd_ctx, ”config”, O_WRONLY);
  configure_fpga(fd_cfg, MATCHER_RTRM_BITSTREAM);

  // open memory and status
  int fd_mem = openat(fd_ctx, ”memory/FPGA MEM1”,
      O_RDWR);
  int fd_status = openat(fd_ctx, ”status”,
      O_RDONLY);

  // fill memory with data (DMA bulk transfer)
  pwrite(fd_mem, search_db_in, db_size, DB_OFFSET);
  pwrite(fd_mem, patterns_in, 4 * pattern_count,
      PATTERN_OFFSET);

  // start the matcher
  acc_run(fd_ctx, 0);

  // check status
  // (wait until context execution finished)
  read(fd_status, bufstatus, 12);

  // read results of operation (DMA bulk transfer)
  ret = pread(fd_mem, results_out,
      results_size, RESULTS_OFFSET);

  // close files
  close(fd_mem); close(fd_status); close(fd_cfg);

  return ret;
}
```

Figure 7. Pattern matcher user program

```
int run_compute_kernel (double * results_out,
    int results_count) {
  // create context of our FPGA design
  int fd_ctx = (int)acc_create(”example”, V_ID,
      D_ID, 0750, NULL);

  // configure the design
  int fd_cfg = openat(fd_ctx, ”config”, O_WRONLY);
  configure_fpga(fd_cfg, MERSENNE_RTRM_BITSTREAM);

  // open memory
  int fd_mem = openat(fd_ctx, ”memory/FPGA MEM1”,
      O_RDWR);

  // allocating buffer
  int32_t * buffer = (int32_t *) mmap(NULL,
      MEM_SIZE, PROT_READ | PROT_WRITE,
      MAP_SHARED, fd_mem, 0);

  int32_t * mt32_numbers = buffer + NUMBERS_OFFSET;

  // start the Mersenne twister MT32
  acc_run(fd_ctx, 0);

  // Example C function that uses random numbers
  c_kernel_function(results_out, results_count,
      mt32_numbers);

  // unmap buffer
  munmap((void *) buffer, MEM_SIZE);
  // close files
  close(fd_mem); close(fd_cfg);
  return 0;
}
```

Figure 8. Example that uses MT32 pseudo random numbers

read request on an arbitrary RTRM address, a new 32 bits number is provided.

The RTRM pattern matcher simultaneously compares several 32 bits patterns against a search database. The module consists of a finite state machine (FSM), four 32 bits comparators for each pattern, one control register, one status register as well as dual-port block RAMs for the search database, the search patterns and the results. Additionally, a 56 bits window is superimposed over the search database.

The registers and memories are mapped into the lower 27 bits addresses of the RTRM's address space and can be accessed by the host.

After the host has set the start bit in the control register, the FSM reads the search patterns from the pattern memory, the window is set to the beginning of the search database and the comparators are enabled.

Then, the first comparator of each search pattern tests the first 32 bits of the window, the second one 32 bits shifted by one byte, the third one 32 bits shifted by two bytes and the fourth the last 32 bits of the window against the search pattern. Hereby, the window can be shifted by 32 bits each clock cycle.

When the end of the search database has been reached, the results are written to the results memory. Afterwards, the 'finished' bit is set in the status register. Next, the host can read the matcher results from the results memory.

6.3. User Application accessing RTRMs

The user function `matcher_run` (cf. Figure 7) demonstrates the usage of the RTRM pattern matcher. First, this function creates a new context and partially reconfigures the FPGA by the function `configure_fpga`. Then, the search database and search patterns are written to the RTRM's database and patterns memory using the `pwrite` system call. Next, the matcher is started using `acc_run` and the user function waits until the execution has finished. After that, the results are read from the FPGA into the buffer `results_out` by the `pread` system call.

The user function `run_compute_kernel` (cf. Figure 8) uses the pseudo random numbers generated by the RTRM Mersenne twister for the computation kernel `c_kernel_function`. This RTRM is initialized using the same functions like in the previous

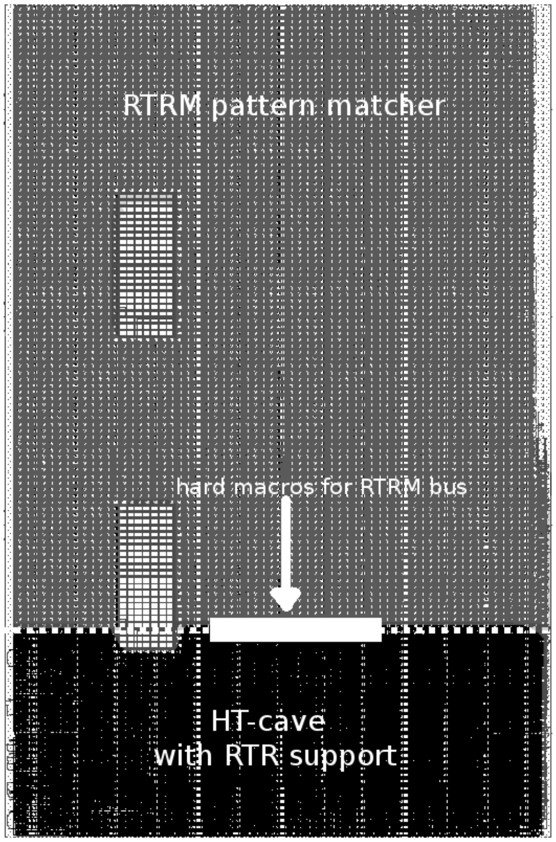


Figure 9. Placed and routed design of the HT cave with RTR support and the pattern matcher RTRM

example. In contrast to the previous one, the random numbers are not read using file handles, but can be accessed by the computation kernel via the memory-mapped buffer `mt32_numbers`.

6.4. Results of Case Study

The infrastructure for RTR modules based on the HT cave with RTR support was successfully implemented and verified. Furthermore, the virtual file system ACCFS was utilized for the integration and management of RTR modules on a HyperTransport plug-in card with a Xilinx Virtex-4 FPGA by using two example RTR modules which can be loaded onto the FPGA during run-time. For the implementation of the HT cave with RTR support at least 4 clock regions have to be reserved as static part.

The first RTR module acting as a offload function which finds patterns in a byte stream (pattern matcher) consists of 290 pattern matcher units resulting in a total of up to 116 billion 32 bits comparisons per second. This module nearly occupies all slices available within the clock regions designated for the RTRM. The placement is shown in Figure 9.

The second module implemented is a Mersenne Twister which generates pseudo random numbers at high output frequency.

For generating the partial bit stream file the framework presented in subsection 4.4 was applied.

7. Conclusion

By using the ability of run-time reconfiguration of FPGAs it is possible to build a single FPGA chip solution as a host coupled accelerator without loosing the host link connection during the reconfiguration of RTR modules. The design of a RTR infrastructure inside the FPGA was shown which allows to manage RTR modules during run-time. The implementation was done for FPGAs coupled directly to the HyperTransport processor bus of the host system. The concepts provided are applicable to other processor and peripheral bus coupled FPGAs. The software framework ACCFS, based on a virtual file system, provides a generic interface to user applications which is able to satisfy the demands of run-time reconfigurable computing.

8. Future Work

To speed up communication with high throughput between the host and a RTRM a memory transfer controller supporting bulk transfer between the different address spaces of the host and the RTRM should be implemented.

9. Acknowledgment

The project is performed in collaboration with the Center of Advanced Study Böblingen, IBM Research & Development GmbH, Germany.

References

[1] N. A. Woods and T. VanCourt, “FPGA Acceleration of Quasi-Monte Carlo in Finance,” in *Proceedings of the 2008 IEEE International Conference onField-Programmable Logic, FPL 2008, 8-10 September, Heidelberg*. IEEE, 2008, pp. 335–340.

[2] G. L. Zhang, P. H. W. Leong, C. H. Ho, K. H. Tsoi, C. C. C. Cheung, D.-U. Lee, R. C. C. Cheung, and W. Luk, “Reconfigurable Acceleration for Monte Carlo Based Financial Simulation,” in *FPT*, G. J. Brebner, S. Chakraborty, and W.-F. Wong, Eds. IEEE, 2005, pp. 215–222.

[3] J. Hagemeyer, B. Kettelhoit, M. Koester, and M. Pormann, in *Design of Homogeneous Communication Infrastructures for Partially Reconfigurable FPGAs (ERSA)*. CSREA Press, 2007.

[4] D. Koch, C. Beckhoff, and J. Teich, "ReCoBus-Builder a Novel Tool and Technique to Build Statically and Dynamically Reconfigurable Systems for FPGAs," in *Proceedings of the 2008 IEEE International Conference onField-Programmable Logic, FPL 2008, 8-10 September, Heidelberg*. IEEE, 2008.

[5] J. Surisi, C. Patterson, and P. Athanas, "An efficient run-time router for connecting modules in FPGAs," in *Proceedings of the 2008 IEEE International Conference onField-Programmable Logic, FPL 2008, 8-10 September, Heidelberg*. IEEE, 2008.

[6] T. Pionteck, C. Albrecht, K. Maehle, E., Hübner, M., and Becker, J., "Commuication Architectures for Dynamically Reconfigurable FPGA Designs," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, IPDPS USA, 2007.

[7] E. Lübbers and M. Planner, "ReconOS: An RTOS Supporting Hard-and Software Threads," in *Proceedings of the 2007 IEEE International Conference on Field-Programmable Logic and Applications*. Amsterdam: IEEE, 27-29 August 2007, pp. 441–446.

[8] H. K.-H. So and R. Bordersen, "File System Access From Reconfigurable FPGA Hardware Processes In BORPH," in *Proceedings of the 2008 IEEE International Conference onField-Programmable Logic, FPL 2008, 8-10 September, Heidelberg*. IEEE, 2008.

[9] A. Heinig, R. Oertel, J. Strunk, W. Rehm, and H. Schick, "Generalizing the SPUFS concept - a case study towards a common accelerator interface," in *Proceedings of the Many-core and Reconfigurable Supercomputing Conference*, Belfast, 1-3 April 2008.

[10] "Xilinx Virtex family," Website, 2008. [Online]. Available: http://www.xilinx.com/products/

[11] Xilinx, "Configuration Memory Frames," in *Virtex-4 FPGA Configuration User Guide (UG071)*, 2008.

[12] Xilinx, "Two Flows for Partial Reconfiguration: Module Based or Difference Based," in *Application Note: Virtex, Virtex-E, Virtex-II, Virtex-II Pro Families (XAPP290)*, 2004.

[13] D. Slogsnat, A. Giese, and U. Bruening, "A versatile, low latency HyperTransport core," in *Fifteenth ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2007.

[14] A. Bergmann, "The Cell Processor Programming Model," IBM Corporation, Tech. Rep., June 2005.

[15] M. Nuessle, H. Fröning, A. Giese, H. Litz, D. Slogsnat, and U. Brning, "A Hypertransport based low-latency reconfigurable testbed for message-passing developments," in *KiCC'07*, 2007.

[16] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, 1998.

[17] "Mersenne Twister, MT32. Pseudo Random Number Generator for Xilinx FPGA," Website, 2007. [Online]. Available: http://www.ht-lab.com/freecores/mt32/mersenne.html