IPSJ Transactions on System LSI Design Methodology Vol.6 112–121 (Aug. 2013)

[DOI: 10.2197/ipsjtsldm.6.112]

Regular Paper

Design and Implementation of IP-based iSCSI Offload Engine on an FPGA

Amila Akagic[1,a)] Hideharu Amano[1,b)]

Received: November 28, 2012, Revised: March 8, 2013,
Accepted: April 26, 2013, Released: August 5, 2013

Abstract: The IP-based storage systems often require bandwidth intensive access to storage devices, thus they exhibit high CPU utilization and low throughput when executed in a principally software implementation. This is especially evident for multi-Gbps networks where the impact of computational overhead is so pronounced that the current state of the art processors cannot take advantage of the capacity of the network. In this paper we propose new iSCSI Offload Engine architecture for high data rate storage networking. Based on our analysis of open source Open-iSCSI initiator, we offload the most computationally intensive and the most executed functions in a common case scenario, while other functions are implemented in a modified Open-iSCSI initiator on a general purpose processor. Our architecture overcomes the performance limitations imposed by a single processor which runs on 15x higher operating frequency than our accelerator. It exhibits very low CPU utilization of approximately 3% on the host CPU, which is 10–15x reduction compared with software implementation. The maximum transmission throughput is 7.81 Gbps, while reception throughput is 7.34 Gbps, which is 2 times speedup over software. The new architecture also shows comparable performance with Chelsio T110 ASIC-based HBA, and has more flexibility.

Keywords: Hardware Acceleration, Internet Small Computer System Interface (iSCSI), Offload Engine, FPGA, IP-based storage

1. Introduction

The IP-based storage systems provide a flexible and high-performance block data access for storage applications. Their unique contribution is the ability to integrate storage networking into mainstream data communications. The iSCSI protocol [1] defines one such approach for accessing and transporting data over commonly utilized TCP/IP infrastructure. The protocol ensures high data integrity through header and data digests in the specific iSCSI Protocol Data Units (PDUs). However, the processing of iSCSI digests is considered to be the most computationally intensive part of the iSCSI protocol processing [2]. This is especially evident for multi-Gbps networks where the impact of computational overhead is so pronounced that the current state of the art processors cannot take advantage of the capacity of the network. Thus, it is common practice to disable data digests [3]. In such cases, data integrity is ensured with only TCP and/or Ethernet error detection mechanisms, which have significantly lower detection capabilities than the more robust 32-bit Cyclic Redundancy Check (CRC) iSCSI digest. TCP checksums cannot detect errors which occur between upper layer protocol transitions.

Thus far, commercial hardware iSCSI solutions have been implemented by using TCP/IP Offload Engines (TOE) or iSCSI host bus adapters (HBA). These systems offload either TCP/IP protocol stack or both TCP/IP and iSCSI protocol onto a specialized hardware. Offloading TCP/IP protocol stack shows significant decrease of CPU utilization, but digest processing still requires significant processing time on a CPU. iSCSI HBAs guarantee performance for computationally intensive applications, but it is difficult to add new functions due to their inflexibility. There has been only one attempt to offload iSCSI protocol to an FPGA [4]. However, the maximum reported throughput is only 86 Mbps without processing digests, and 31.84 Mbps with digests. There are three primary reasons why we believe offloading the iSCSI protocol is challenging. First, the scope of iSCSI code is too large and requires a lot of programming effort and time. Second, some functions such as authentication, authorization and security are challenging to implement in hardware. Third, it is thought that operating frequency of FPGAs is not enough to accomplish required throughput for high-speed networks. The performance of software initiators is limited by the processing power of a general purpose processor, especially for the multi-Gbps networks [5]. The biggest concern is high level of CPU utilization that it causes. This has led to extensive research of offloading protocol processing to hardware.

In this paper we address the problem of efficiently implementing IP-based iSCSI Offload Engine which operates on the top of the TCP/IP protocol stack. The target applications are mission-critical applications which require high data integrity, such as those of financial and banking transactions where database integrity failures might lead to lost funds, inaccurate stock exchange or credit card transactions. In these systems it is required

[1] Department of Information and Computer Science, Keio University, Yokohama, Kanagawa 223–8522, Japan
[a)] amila@am.ics.keio.am.jp
[b)] hunga@am.ics.keio.am.jp

to enable header and data digests, which adversely affects overall performance.

Precisely, our contributions are:

(1) We analyze iSCSI traffic and identify the most commonly used functions. We measure and analyze CPU utilization and throughput of Open-iSCSI [6], which is an open source software based iSCSI initiator.

(2) Based on (1), we offload data transfer and related non-data functions to an FPGA based adapter. Data transfer functions are the most computationally intensive and the most executed functions in a common case scenario. Other functions which do not affect performance are implemented in software on a general purpose processor. The resulting architecture relieves the host CPU from computational burden imposed by the software implementation.

(3) It is proved that the new architecture can overcome the performance limitations imposed by a single processor which operates on 15 times higher frequency than our FPGA implementation. The iSCSI Offload Engine allows very low utilization on the host CPU of approximately 3%.

(4) Our architecture guarantees flexibility, since many functions are implemented on a general purpose processor. Any new feature, such as security functions, specification updates, CRC standards, etc., can be easily implemented.

The organization of this paper is as follows. In Section 2. we present an overview of iSCSI architecture, an analysis of software based Open-iSCSI initiator and related works. In Section 3. we describe new iSCSI Offload Engine architecture and modification of Open-iSCSI initiator. In Section 4. we present results and analysis of our iSCSI Offload Engine, and we compare our results with related works. We conclude the paper in Section 5.

2. Overview of iSCSI

2.1 The iSCSI Protocol

The iSCSI protocol is a transport for SCSI packets over TCP/IP infrastructure. The information exchange is based on a client/server model where the client is called *initiator*, and server *target*. The initiator and target divide their communications into messages, which are called Protocol Data Units (PDUs). Typically, an initiator issues commands to a SCSI target to request transfer of data to/from I/O devices. The group of TCP connections that link an initiator with a target form a session. A session has two phases: Login and Full Feature Phase. In the Login Phase, an initiator and a target negotiate protocol parameters, security parameters, and authenticate each other for the rest of the session. The session then transitions to the Full Feature Phase. In this phase, an initiator may send SCSI commands and data to various SCSI devices on the target. The majority of protocol processing load happens in the second phase.

2.2 Processing of iSCSI Read and Write Commands

The principal layers of the storage networking model based on iSCSI are shown in **Fig. 1**. The data segment encapsulates the SCSI command set for communication with SCSI devices. iSCSI layer is responsible for transmitting and receiving SCSI commands over TCP/IP infrastructure. The TCP layer is used as

Fig. 1 Layers of iSCSI packet. The formation of the packet begins with data segment, creation of header and data digests, and an appropriate iSCSI header. Then, the packet is build out through TCP, IP and Gigabit Ethernet layers (Optional fields are marked with *).

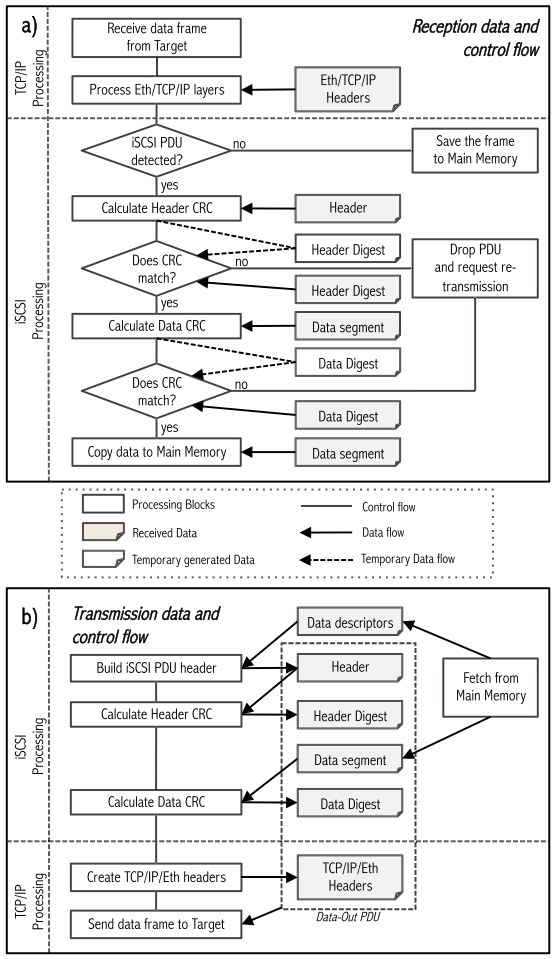

Fig. 2 Flow diagram for processing of a) Data-In PDU on the initiator, which performs SCSI read on the target; b) Data-Out PDU on the initiator, which performs SCSI write on the target.

end-to-end protocol to establish a reliable session, and for delivering in-order TCP segments to the iSCSI layer. The IP layer is used to route the data between network devices, and the Ethernet layer is used as MAC protocol handler to transfer Ethernet frames across the physical link.

Figure 2 illustrates an exemplary flow diagram for processing a) incoming Data-In PDU (part of *READ* operation) and b) outgoing Data-Out PDU (part of *WRITE* operation). These units are two main vehicles by which SCSI data payload is transmitted between an initiator and a target.

In the case of Fig. 2 (a), the initiator first sends the request for reading data in the form of SCSI READ commands to a target. When a target sends data, the incoming data first goes through TCP/IP Offload Engine (TOE) to process Ethernet, TCP and IP layers. Then, if the frame does not correspond to an iSCSI PDU, it is forwarded either to a different network processor or to the main memory. Else, the header is validated by calculating its digest in the second phase. If the newly calculated digest is not the

same as the received one, the PDU is dropped and re-transmission request is sent. In the third phase, the information in the iSCSI frame is identified and corresponding operations are performed. In the final phase, the digest of data segment is calculated and compared with the received data digest. If two digest values are equal, the data segment is copied to the main memory. If not, the frame is dropped and re-transmission is requested.

In the case of Fig. 2 (b), the initiator first sends the request for writing data in the form of SCSI WRITE commands to a target. Then, the target sends *R2T PDU* informing the initiator that it is ready to transmit. When the initiator receives R2T, transmission of data-out PDU may begin. The formation of Data-Out PDU begins with construction of its header. Then, the header and data segment digest are calculated, respectively. The header, header digest, data segment and data digest are then encapsulated with TCP, IP and Ethernet layers to form a Data-Out PDU and sent to a target.

2.3 Implementation Approaches

In recent studies we found three major implementation choices for iSCSI (**Fig. 3**):

Type 1: *iSCSI Driver with NIC*: coupled with a generic Ethernet NIC with software implementation of iSCSI initiator.

Type 2: *iSCSI Driver with TCP offload engine*: entire TCP/IP stack is offloaded onto a special purpose hardware accelerators coupled with operating system based iSCSI initiator.

Type 3: *iSCSI Host Bus Adapter*: TCP/IP and iSCSI initiator functions are offloaded to a special purpose hardware.

For some applications, software initiators (Type 1) will suffice, but more-demanding applications require offloading of iSCSI processing to hardware initiators. The main advantage of software based iSCSI initiators is their ability to easily adapt to modifications in the protocol. There are two types of iSCSI hardware initiators. Type 2 only offloads TCP/IP processing from the system's CPU to a specialized Ethernet card which is called TCP Offload Engine. Type 3 offloads both TCP/IP and iSCSI processing from the system CPU to a specialized adapters known as iSCSI Host Bus Adapters. They are usually implemented as ASIC solutions with superior performance when compared to performance of Type 1, but they lack flexibility. Examples of existing implementations are reviewed in Related Work.

2.4 Performance Analysis of Open-iSCSI

We analyzed iSCSI traffic with Wireshark [7], the open source network packet analyzer. We measured traffic between a software initiator and a target by using a set of microbenchmarks. The microbenchmarks transmitted arbitrary number of data in both directions. The iSCSI commands are issued to read/write from/to the same disk block address multiple times in order to minimize the number of cache misses. We setup a software initiator with Open-iSCSI [6] on Intel Core2 CPU 2.40 GHz with 8 GB of RAM, and a target by using Linux SCSI target framework [8] on the Intel Core2 Quad CPU 2.83 GHz with 8 GB of RAM. The operating system on both CPUs was based on Linux kernel 2.6.34.

In the most common case transmission, 60–70% of instructions

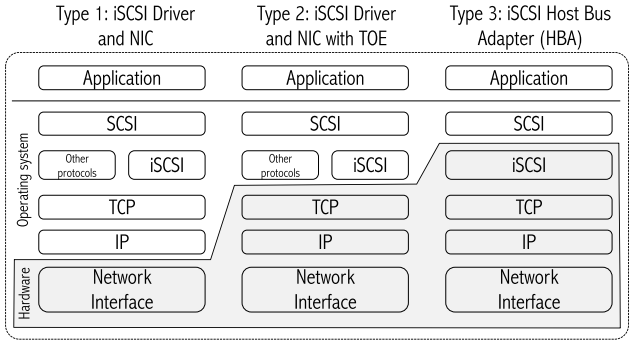

Fig. 3 Three major implementation choices for iSCSI.

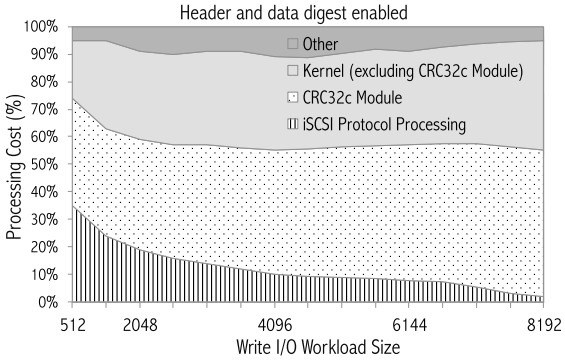

Fig. 4 The performance profile of processing Data-In PDUs on Intel Core2 CPU 2.40 GHz, when header and data digests are enabled.

were Data-In and/or Data-Out PDUs, following by R2T, SCSI Commands and Responses with 10–20%. The remaining instructions were related to mostly Login Phase, connection cleanup and connection termination. Then, we analyzed number of instructions and CPU utilization with Oprofile [9], which is a system-wide profiler for Linux kernel. **Figure 4** shows the performance profile of processing Data-In PDUs when header and data digests are enabled. The cost of data digest processing (with kernel's CRC32c module) represents about 50% of the total number of instructions for 8 KB workload size, while the iSCSI protocol processing is only 4%. As expected, the data digest processing increases linearly with I/O workload size, while processing cost is decreasing. During our experiments, CPU utilization was the highest when data digests were enabled, varying from 33% to 70% of processor's resources. Thus, little or no processing resources are left for other applications. When data digest is disabled, the cost of header digest processing is indistinguishable with 1% of the total number of instructions.

2.5 Related Work

The most common Type 1 implementations in the research community are open source Open-iSCSI [6] and UNH-iSCSI projects [10]. Examples of Type 2 are ASIC-based *10 GbE* TOEs: Chelsio's Terminator 3 chip [11] and NetEffect's NE010 adapter [12]. Both adapters show low CPU utilization and near 10 Gbps performance, especially for larger data sizes. However, very little information is available concerning their architectures. There is some research about TOEs on FPGAs. In Ref. [13], Wu et al. introduced a hybrid TOE which processes IP, ARP, and ICMP protocols on an FPGA, and TCP on an em-

bedded processor by using software. In Ref. [14], Jang et al. presented the design and implementation of a TOE by means of hardware/software co-processing. Both implementations focus on decreasing CPU utilization by offloading TCP/IP processing to an FPGA. The maximum reported throughput is bellow 1 Gbps: Wu et al. reported 300 Mbps [13], Jang et al. reported 673 and 551 Mbps [14]. However, two companies recently announced FPGA based TOEs which operate at full 10 Gbps [15], [16] line rate. As we argued in the previous section, digest processing occupies significant amount of CPU utilization. Thus, it is not enough to offload TCP/IP processing to a special purpose hardware. This is especially evident for multi-Gbps networks where the impact of computational overhead is so pronounced that the current state of the art processors cannot take advantage of the capacity of the network.

References [4], [17], [18] are examples of Type 3. In Ref. [4], Han-Chiang Chen et al. proposed offloading of TCP/IP and iSCSI to an embedded OS on a PowerPC 405 CPU, which is part of Xilinx FPGA embedded platform. This is the only attempt to offload iSCSI to an FPGA. The implementation does not have any hardware accelerated modules, and it only consists of running unmodified software initiator on the PowerPC 405 CPU. The maximum reported throughput is 86 Mbps without digests, and 31.84 Mbps with digests. The low throughput is attributed to the low frequency of PowerPC 405 CPU (300 MHz). The CPU utilization was 1.5%. In Ref. [17], Chung-Ho Chen et al. proposed a hardware accelerator for data transfer iSCSI functions. The accelerator is designed with direct C-to-HDL translation of specific submodules of UNH-iSCSI software. The design is evaluated with UMC 0.18 μ technology with 100 MHz system clock. The accelerator is able to meet the requirements of 1 Gbps network when the average PDU size is greater than 125 bytes. In Ref. [18], the peak throughput performance of Chelsio T110 (10 Gbps iSCSI ASIC-based HBA) is 6.69 Gbps without digests, and 5.9 Gbps with digests. The average CPU utilization is 30% without digests, and 37% with digests.

3. Design and Implementation of iSCSI Offload Engine

3.1 Overview of iSCSI Offload Engine Architecture

Figure 5 illustrates the structure of iSCSI adapter based on the architecture of the iSCSI Offload Engine proposed in this paper. The design overview is based on Xilinx ML605 Evaluation Board. The iSCSI adapter consists of an iSCSI Offload Engine, a TCP/IP Offload Engine, an iSCSI Offload Engine Interface, a memory controller, a 10-Gigabit Ethernet Media Access Controller (MAC) and an eXtended Attachment Unit Interface (XAUI) Core. The 10-Gigabit Ethernet MAC is used to interface to Physical Layer devices in a 10-Gigabit Ethernet (10 GE) system. The XAUI Core allows physical separation between the data link layer and physical layer devices in a 10 GE system. More implementation details are provided in Section 4.1. Our architecture is relying on the existing TCP/IP Offload Engine, which is well researched subject [13], [14], [15], [16].

In a typical iSCSI session, an initiator initiates series of read and/or write SCSI commands, after which appropriate responses

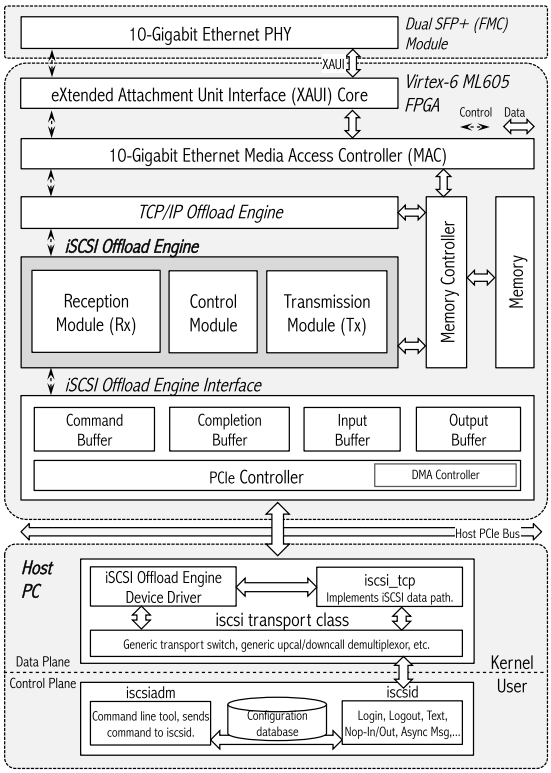

Fig. 5 Design overview of proposed iSCSI Offload Engine and modified Open-iSCSI implementation. The design is based on Xilinx ML605 Evaluation Board, which has only a 1000-BASE Ethernet interface. Hence, additional Dual SFP+ FMC [24], [25] and 10 GbE SFP+ transceiver are required to achieve throughput of over 1 Gbps. More implementation details are provided in Section 4.1.

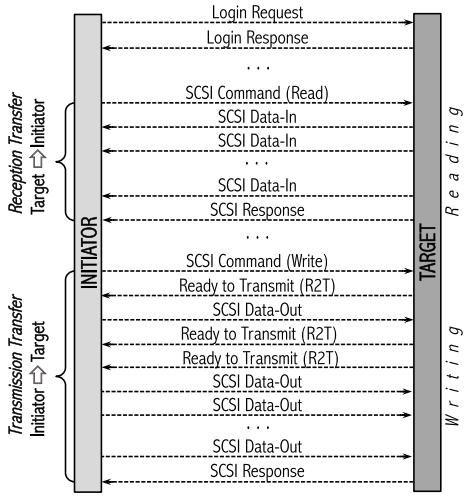

Fig. 6 An overview of two transfer directions and two common sets of operations executed during reading and writing processes.

follow, as illustrated in **Fig. 6**. Several read and/or write commands, as well as their data and responses usually intertwine, depending on the readiness to transmit data on initiator and target side. Data transmitted from a target to an initiator is regarded as *reading part* of the session (reception). Similarly, the transfer from an initiator to a target is regarded as *writing part* of the session (transmission). Thus, *the iSCSI Offload Engine* consists of two modules which divide processing work into reception work - *the Reception Module* (Rx) and transmission work - *the Transmission Module* (Tx). The architecture of two modules is discussed

Table 1 A minimum set of opcodes defined on an initiator and a target. The iSCSI Offload Engine processes the most computationally intensive data-transfer and related non-transfer operations in both directions, marked in bold.

Initiator to Target (Tx)			Target to Initiator (Rx)		
No.	Opcode	Name	No.	Opcode	Name
T1	0x00	NOP-Out (H&D)	R1	0x20	NOP-In (H&D)
T2	0x01	**SCSI Command (H&D)**	R2	0x21	**SCSI Response (H&D)**
T3	0x02	SCSI Task Management function request (H)	R3	0x22	SCSI Task Management function response (H)
T4	0x03	Login Request	R4	0x23	Login Response
T5	0x04	Text Request (H&D)	R5	0x24	Text Response (H&D)
T6	0x05	**SCSI Data-Out (H&D)**	R6	0x25	**SCSI Data-In (H&D)**
T7	0x06	Logout Request (H)	R7	0x26	Logout Response (H)
T8	0x10	**SNACK Request (H)**	R8	0x31	**Ready To Transfer (R2T) (H)**
T9	0x1c-1e	Vendor specific codes	R9	0x32	Asynchronous Message (H&D)
			R10	0x3c-0x3e	Vendor specific codes
			R11	0x3f	Reject (H&D)

(H&D) : Header and data digest (H) : Header digest

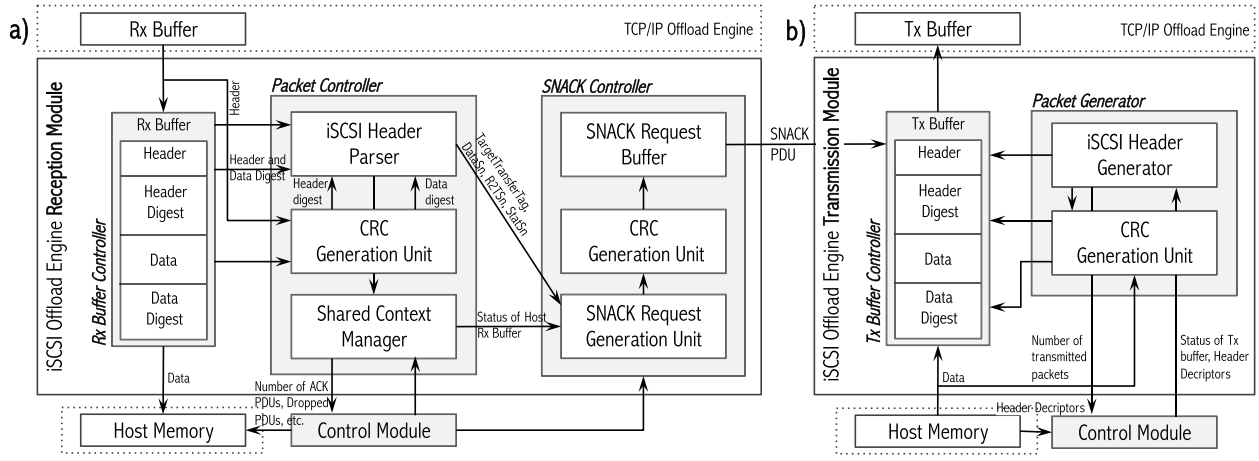

Fig. 7 The structure of Reception and Transmission Modules in the iSCSI Offload Engine.

in Sections 3.2 and 3.3, respectively.

The Control Module enables sharing of data between Reception and Transmission Modules, TCP/IP Offload Engine, and modified Open-iSCSI initiator. The memory controller handles buffer memory which holds the packet buffers. The packet buffers consist of a header, data, header digest and data digest areas.

Table 1 displays the minimum set of opcodes defined on an initiator and a target. Based on our analysis of Open-iSCSI (Section 2.4), we offload processing of PDUs marked in bold to an FPGA. These PDUs are the most computationally intensive and the most frequently executed. Except R2T and SNACK Request, they all require data digests. The operations such as Asynchronous Message, Text Request and Text Response, Nop-In and Nop-out, and Reject also require data digest, but they are executed far less frequently. Thus, these functions are implemented on a general purpose processor.

On the host CPU, we modified Open-iSCSI and Linux kernel to bypass certain iSCSI functions and TCP/IP layers. The Open-iSCSI is partitioned into kernel and user parts (Fig. 5), which implement iSCSI data plane and the control plane, respectively. The interface between these two parts is implemented using Netlink sockets. The socket library functions are handled in a single system call (sys_socketcall). Depending on the type of a function, the sys_socketcall calls either the iSCSI Offload Engine Device Driver or the TOE Device Driver. The iSCSI Offload Engine Device Driver consists of a set of routines which control the iSCSI Offload Engine. The modifications are discussed in details in Section 3.6.

When a user requests iSCSI Offload Engine service, this request is first delivered to the iSCSI Offload Engine Interface. The modules then read the request from the Command Buffer and perform required operations. The data is copied to/from main memory of the host CPU into Input/Output buffers by using DMA. The host CPU then fetches results from the Completion and Output Buffers and delivers them to the user program.

3.2 The Reception Module (Rx)

Figure 7 (a) illustrates the structure of Reception Module (Rx). It consists of the Packet Controller, the SNACK Controller and the Rx Buffer Controller. After the incoming packet is processed by the TCP/IP Offload Engine, the TCP payload is transferred to the Rx Buffer Controller. The Packet Controller parses the header, identifies a PDU and validates header and data digest. If a PDU represents a SCSI Data-In, a SCSI Response or a R2T PDU, it is processed by the Packet Controller, else it is forwarded to the main memory to be processed by the software initiator or to a different processing engine.

Some operations are executed in parallel in order to improve the performance. Parsing of a header and calculation of header's digest are executed in parallel, as well as calculation of data digest and validation of the header digest. When data digest is validated, the data is copied from Rx Buffer directly to the host memory via

DMA without a copy (direct data placement). The header and data digests are calculated with CRC Generation Unit, and validated by the Parser. The architecture of CRC Generation Unit is detailed in Section 3.4.

Even though *SNACK Request* is originally in the transmission data-path, we implement the SNACK controller in the Reception Module in order to shorten the time required to generate a request for re-transmission or acknowledgment of data. However, a SNACK PDU is sent through the Tx Buffer. The re-transmission request (SNACK) is generated when the header or data digests are not validated (the packet in the Rx Buffer is dropped in this case). In order to reduce the overhead of acknowledging each incoming packet, we design the SNACK Controller to generate a single delayed SNACK request for a group of missed status, data, or R2T PDUs within a task. This decreases the number of interrupts and improves the performance, since there are fewer number of requests.

The other example when the SNACK Controller is used is when a session supports error recovery. In this case, the target requests a positive acknowledgment in the form of SNACK DataACK PDU. This operation begins in parallel with the operation to store the validated packet from the Rx Buffer to the host memory. By implementing this operation in the hardware, the resources at the target are released faster, thus enabling more resources for other transactions. The SNACK Controller has independent CRC Generation Unit, thus it can generate header digest in parallel with the Packet Controller.

3.3 The Transmission Module (Tx)

Figure 7 (b) illustrates the structure of Transmission Module (Tx). It consists of the Packet Generator and the Tx Buffer Controller. When the iSCSI Offload Engine device driver requests creation of an iSCSI PDU, the header descriptors are fetched from the main memory via DMA and forwarded to the Packet Generator. The iSCSI Header Generator creates a new header and forwards it to CRC Generation Unit to create a header digest.

The following two sets of operations are executed in parallel. First, the generation of header digest is executed in parallel with transfer of a header from the iSCSI Header Generator to the Tx Buffer. Second, the generation of data digest is executed in parallel with transfer of data from the main memory to the Tx Buffer. The Tx Buffer Controller forwards a PDU to the TCP/IP offload engine for creation of TCP/IP/Eth header information. The TCP/IP Offload Engine then sends a request to the Gigabit Ethernet controller to transmit the packet.

3.4 The CRC Generation Unit

Figure 8 illustrates the architecture of CRC Generator Unit based on our previous research with high-speed CRC accelerators [19], [20], [21]. The CRC algorithm deploys eight tables ($T8, ..., T2, T1$) with pre-computed remainders. The architecture is pipelined in three stages, and the throughput is 64 bit/cycle. The digest (CRC) of input data is formed with the following steps. In the first iteration, the *Intermediate Address* is formed by XORing input data with initial value (*Init*). In the every other iteration, the *Intermediate CRC* is used instead of *Init*. The *Intermediate Address* is then sliced into eight 8-bit slices, which are used as addresses to access eight tables in parallel. Eight remainders are XORed to form the *Intermediate CRC*. The *Intermediate CRC* is XORed with the final value (*XorOut*) when the controller indicates the end of data. The digest is stored in the digest area of a buffer.

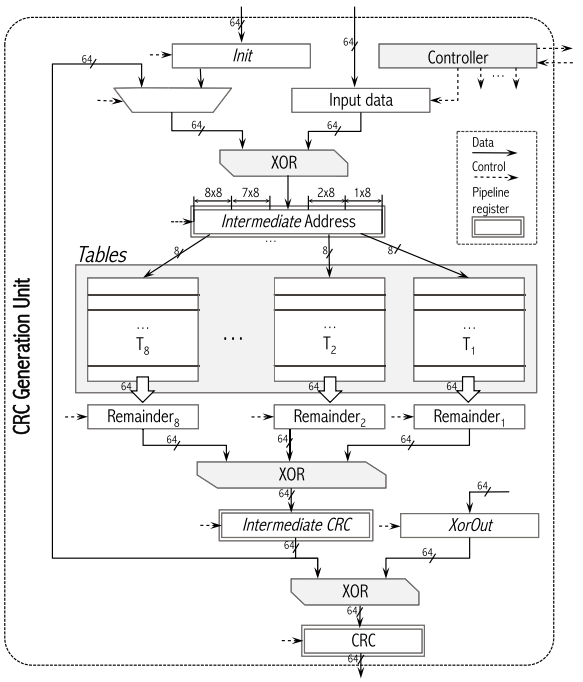

Fig. 8 The architecture of CRC Generation Unit.

In order to achieve high degree of flexibility, we made some modifications to the original design. We enabled support for any given 32-bit CRC Standard defined in the iSCSI Specification [1]. The values of parameters *Init* and *XorOut*, and the contents of tables depend on a CRC standard. The values of parameters are changed on the request of the iSCSI Offload Engine device driver. However, the contents of tables is challenging to be replaced in the similar manner. Thus, we use difference-based partial reconfiguration. We minimize the dynamic part of the circuit to only tables, which allows us to generate a small bitstream containing only differences between two versions of the design. Then, we wrote a set of scripts to automatically assign new values corresponding to a CRC Standard, and stored them into the tables (BRAM components). The new values are stored into tables with the Xilinx FPGA Editor. We automatize the process of generating new bitstream which allows complete flexibility. The idea is to provide a number of pre-generated bitstreams (for a set of CRC standards) with the unit.

3.5 The Control Module

The Control Module shares information among four components: Reception and Transmission Modules, TCP/IP Offload Engine, and iSCSI software initiator. It supports fast and efficient data sharing by using quad-port memory [22]. In **Fig. 9** we illustrate an exemplary exchange of information between an initiator and a target, where italic font displays direction of the new information coming from an initiator to a target, and inversely. The iSCSI initiator must verify consistency of the values used in all

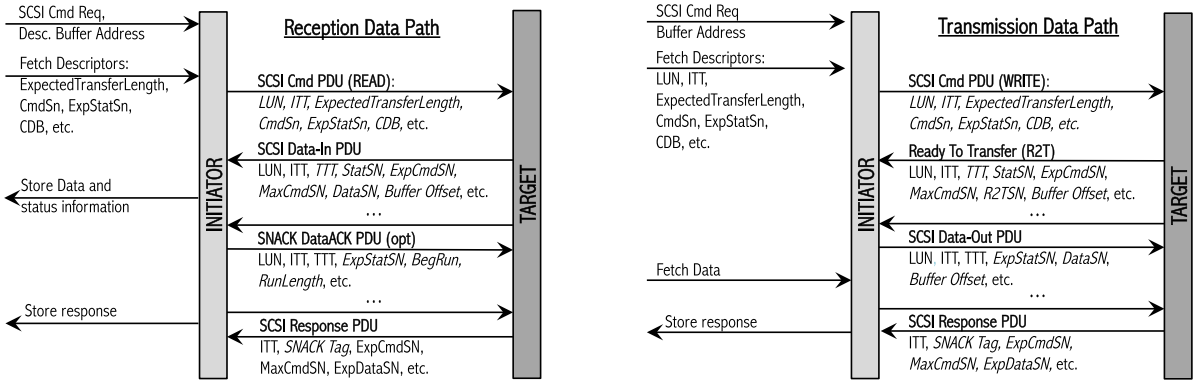

Fig. 9 An exemplary exchange of information between the initiator and target. Italic font displays direction of the new information coming from an initiator to a target and inversely. The information is used for validation of a PDU.

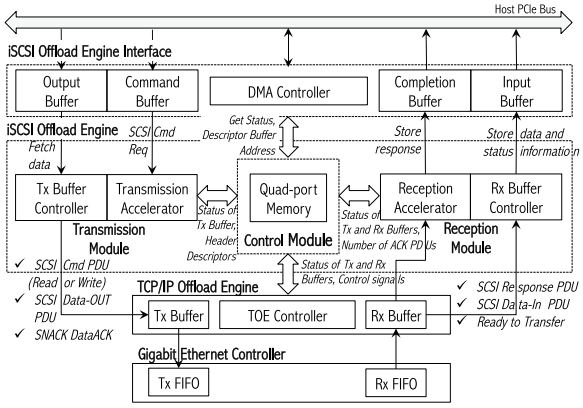

Fig. 10 The flow of information between modules in the iSCSI Offload Engine.

task-related PDUs. Thus, it stores important information in *five* look-up tables in a memory of a Control Module and forwards them to an appropriate unit for verification.

Figure 10 illustrates the flow of information between modules in the iSCSI Offload Engine. Before sending a request to create a command PDU, the device driver first checks the status of iSCSI Offload Engine via the *session* table in the Control Module. Along with the request, it sends an address of the buffer with a set of information required to form a command PDU. This set is defined by the RFC 3720 [1]. In the case of "*SCSI Cmd PDU*" (Fig. 9), following kinds of information are being exchanged: Logical Unit Number (LUN), Initiator Task Tag (ITT), expected transfer length, command sequence (CmdSn), expected status number (ExpStatSn), etc. The LUN is used to identify a Logical Unit within a target, and ITT to identify a new task in the initiator. The *command* table is used to store these information. In the response to a command, the target sends a set of information such as a Target Transfer Tag (TTT), expected command sequence number, sequence number of data PDU, etc. The Control Module also holds information regarding the status of transmit and receive buffers in the host memory, as well as in the TCP/IP Offload Engine. The *R2T* table holds information received from a target through R2T PDU, which are later used by the Transmission Module to create a SCSI Data-Out PDU. The *SNACK* table holds information required to generate SNACK requests, which can be requested from iSCSI Offload Engine device driver or generated directly by the Reception Module. When a PDU is acknowledged, it sends necessary information to modified Open-iSCSI. Lastly, the *data_address* tables holds the address where the data is directly copied from Rx Buffer to the host memory via DMA.

3.6 Modification of the Open-iSCSI Initiator

We modified Open-iSCSI's data-path to bypass some of the SCSI functions and TCP/IP layers in the Linux kernel. The Open-iSCSI spawns two threads for every connection in a session: a transmit thread (tx_thread) and a receive thread (rx_thread). **Figure 11** shows an exemplary unmodified and modified data-paths for creating a SCSI Command by the tx_thread. First, the SCSI Mid-Level passes commands to the low level drivers through queuecommand() call. Then, the initiator generates unique Initiator Task Tag (ITT) and allocates memory for a new command initialized with it (a). The PDU fields are then prepared and stored in the memory (b). A new command is added to the linked list of all pending commands (c), and tx_thread is woken up to send the PDU to a target (d). Then, a TCP routine is called to send a SCSI Command PDU to a target (e).

The tx_thread and rx_thread data-paths are modified to bypass the processing of T2, T6 and T8 PDUs, and R2, R6 and R8 PDUs, respectively. In the transmission path, a command is first identified by its opcode and forwarded either to unmodified data-path (tx_thread) or to the new iSCSI agent - offload_engine_agent (f), which is responsible for performing communication with iSCSI Offload Engine. Then, a new request is forwarded to the sys_socketcall to be transmitted to a target through either (g) iSCSI_OE_socketcall (T2, T6, T8) or (h) TOE_socketcall (T1, 3-5, 7). In both cases, the Linux TCP/IP stack in the sys_socketcall is bypassed, by which we eliminated copying of user data to the socket buffer (a kernel copy). Instead, we translate the virtual address of the user's data into a physical address by using get_user_pages and kmap functions. The address is sent to iSCSI Offload Engine, which is followed by the DMA request. The requests are created and pushed into the Command Buffer.

4. Implementation Results and Analysis

4.1 iSCSI Offload Engine Board

Design of our iSCSI Offload Engine is best suited for new Xil-

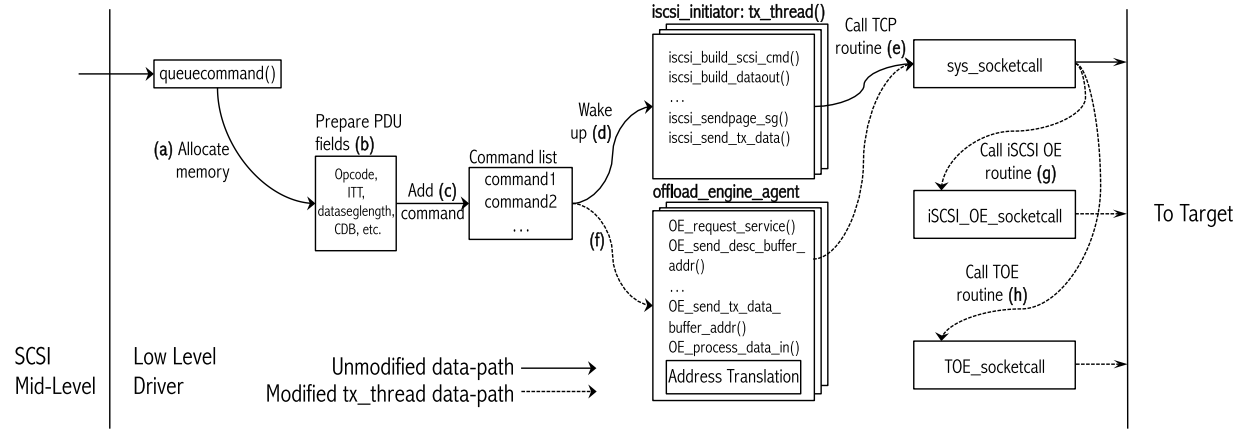

Fig. 11 Unmodified and modified data-paths for creating a SCSI Command by tx_thread.

inx platforms, such as Virtex-6 HXT or Virtex-7 FPGAs, which contain all the necessary hardware for high-bandwidth and high-performance applications. However, the functionality of the proposed iSCSI Offload Engine is verified on the ML605 board, which is equipped with the Virtex-6 XC6VLX240T-1FFG1156 FPGA [23]. Our test platform includes the Multi-port memory controller for accessing the external DDR3 memory. It is connected with host PC with 64-bit PCI Express x4, with transfer rate of 16 Gbps in a single direction. The synchronization between the CPU and the FPGA is performed by the PCIe Message Signaled Interrupts (MSI). This allows an FPGA task to wait for a data being produced by a software task and inversely.

The board has only a 1000-BASE Ethernet interface, hence additional Dual SFP+ FMC [24], [25] and 10 GbE SFP+ transceiver are required to achieve throughput of over 1 Gbps. The Dual SFP+ FMC is an FPGA Mezzanine Connector [26] daughter card with two SFP+ connectors, two 10 Gbps physical layer transceivers which provide full PCS, PMA, and XGXS sub-layer functionality. We utilize only one transceiver. The daughter card is connected to the High Pin Count (HPC) J64 connector of the ML605 board.

As illustrated in Fig. 5, the iSCSI PDUs are formed and encapsulated by iSCSI and TCP/IP Offload Engines and sent out through the LogiCORE IP 10-Gigabit Ethernet MAC [27]. The 10-Gigabit link is supported by the LogiCORE IP XAUI core [28] using a SFP+ cable. The iSCSI Offload Engine is clocked at the standard Ethernet interface frequency of 156.25 MHz, which allows fully synchronous and lowest latency data exchange with DINIGroup TCP/IP Offload Engine [15] and the MAC.

4.2 Elapsed Time of Main Operations

In order for a network adapter to achieve the throughput of approximately 10 Gbps, it has to be able to process a 1,500-byte packet in 1.2 μs. **Table 2** shows elapsed time of main operations processed in iSCSI Offload Engine for a 1,500-byte packet with data digests enabled. We design our iSCSI Offload Engine to interface DINIGroup's TCP/IP Offload Engine [15], which works at the full 10 GbE line rate. Input to output packet latency of the TOE is less than 1 μs, however elapsed time of some operations are already included in elapsed time of iSCSI Offload Engine, such as fetching and storing data from/to host memory and DMA Initialization. These operations require 58% of total time for transmission, and 53% for reception processing. Thus, elapsed time for TCP/IP processing is only 0.21 μs for transmission, and 0.23 μs for reception. The total elapsed time for transmitting a 1,500-byte packet is 1.449 μs, and receiving 1.538 μs.

Table 2 Elapsed time of main operations processed in the iSCSI Offload Engine for a 1,500-byte data packet.

	Hardware Operation	Elapsed time (μs)
Transmission Module	(1) DMA Initialization	.045
	(2) Fetching descriptors from host memory and Header Generation	.038
	(3) Header Digest Generation	.038
	(4) Fetching data from host memory and Data Digest Generation	1.162
DINIGrp TOE [15]	(5) TCP/IP Processing and storing a packet into network interface	.21
	Total:	**1.449**
DINIGrp TOE [15]	(1) Fetching a packet from network interface and TCP/IP Processing	.23
Reception Module	(2) Parsing header and Header Digest Generation and validation	.099
	(3) DMA Initialization	.045
	(4) Data Digest Generation and validation, storing data into host memory	1.162
	Total:	**1.538**

4.3 CPU Utilization and Throughput

Figure 12 shows CPU utilization and throughput of write micro-benchmarks of three implementations: Open-iSCSI running on Intel Core2 CPU 2.40 GHz with 8 GB of RAM, Chelsio T110 iSCSI ASIC-based HBA with CRC enabled (the results are published in Ref. [18]), and our iSCSI Offload Engine. We ran the same set of micro-benchmarks (as discussed in Section 2.4) for several thousand times with I/O sizes ranging from 128 bytes to 128 KB. We used Ethernet standard Maximum Transmission Unit (MTU) of 1,500 bytes. The processing cost of read micro-benchmarks is very similar to write micro-benchmarks, with slightly lower throughput for reading process.

The average CPU utilization of software-based Open-iSCSI varied from 33% to 55% according to a write size. The iSCSI

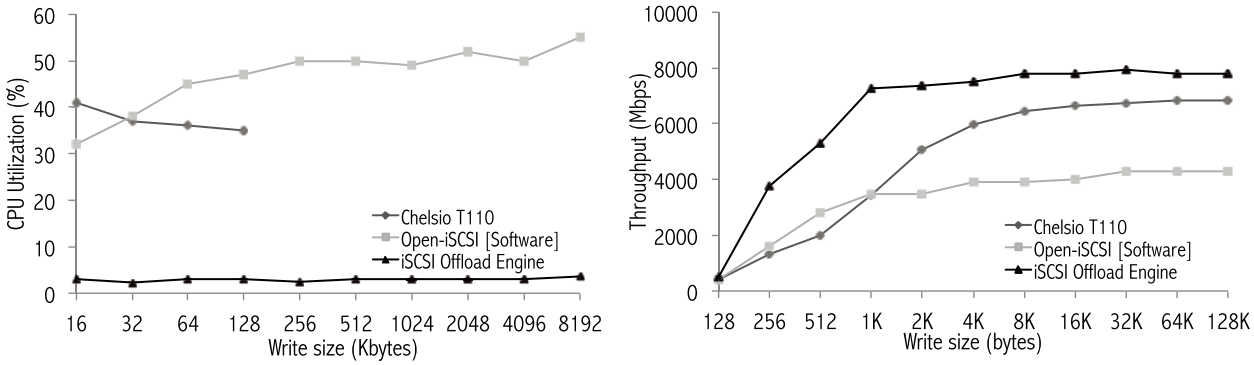

Fig. 12 Comparison of throughput and CPU utilization of write micro-benchmarks for 1,500 bytes MTU.

Offload Engine exhibits very low utilization of approximately 3% on the host CPU, which is 10–15 times reduction compared with Open-iSCSI implementation on Intel Core2 CPU 2.40 GHz, and 10 times reduction compared with Chelsio T110. Unfortunately, we were unable to acquire the host CPU Utilization for the Chelsio T110 for I/O workload sizes higher than 128 KB. Also, from Ref. [18] it is not clear why Chelsio T110 exhibits such high CPU Utilization on the host. However, we think it is because Chelsio T110 only offloads expensive byte touching operations, such as header and data digests generation/checking, while all other related tasks are performed on the host CPU. Our iSCSI Offload Engine additionally processes related non-data transfer functions on an FPGA, which results in decreased number of instructions and interrupts on the host CPU. Additionally, delayed SNACKs and direct data placement in the host memory also decreased the number of interrupts.

There has been significant increase in throughput when iSCSI is offloaded to hardware. Figure 12 shows the overall throughput for different values of write I/O workload size. The software-based Open-iSCSI is executed on 15 times higher clock frequency (2.40 GHz) than iSCSI Offload Engine. However, the maximum transmission throughput of iSCSI Offload Engine is 7.81 Gbps, while the reception throughput is 7.34 Gbps. The results show 2 times speedup over software-based Open-iSCSI. One of the principal reasons why iSCSI Offload Engine achieves higher throughput is because large number of operations are executed in parallel. Specifically, the Open-iSCSI requires 2.15 cycles *per byte* to generate a CRC value for 8 KB of data, while our CRC Generation Unit requires 1 cycle for *eight bytes*. Thus, our CRC Generation Unit requires 17 times less cycles to generate a CRC value than Open-iSCSI.

4.4 Reconfiguration Time

We measured the time required to upload full bitstream and bitstreams of Partial Reconfiguration (PR) modules for the CRC Generation Unit. We used JTAG to upload both bitstreams on the specified FPGA board. The configuration time depends on the size of a bitstream and Test Clock frequency (TCK) for boundary-scan operations. Fixed number of clock cycles required for pre- and post-processing while programming an FPGA is also included in configuration time as specified in Ref. [29], while minimum TCK frequency was 15 MHz. The size of the full configuration bitstream is 9 MB, and the size of PR bitstream is 43 KB. The time to upload these two bistreams is 6.15 s and 0.03 s, respectively. Thus, uploading the PR bitstream is 205 times faster than the time required to upload full bitstream. This feature ensures fast adaptability to new CRC Standards in the future of iSCSI protocol.

Table 3 Resource Utilization of two modules on Virtex-6 XC6VLX240T FPGA. The iSCSI Engine does not contain resource utilization of DINIGrp TOE.

Module	Buffer (KB)	Slices (%)	LUTs (%)	FFs (%)	BRAMs (%)
iSCSI Engine	4	5,983 (16)	15.7 k (10)	8.9 k (3)	24 (3)
	64	5,983	15.7 k	8.9 k	52(6)
DINIGrp TOE [15]	4	2,742 (3)	7 k (4.6)	4 k (1.3)	6 (.7)
	64	2,742	7 k	4 k	34 (4)

4.5 Resource Utilization

Table 3 shows resource utilization on Virtex-6 XC6VLX240T FPGA. The receive and transmit buffers are configurable 4 KB and 64 KB buffers mapped onto dedicated on-chip Block RAMs. The CRC Generation Unit requires only 540 LUTs, and 4 dual-port BRAM for holding contents of its pre-computed remainders. The small resource footprint indicates that multiple instances can be used, thus increasing performance of the system.

4.6 Comparison to Related Work

We compare our work with two other attempts to offload iSCSI to hardware [4], [17], which are discussed in Related Work. Han-Chiang Chen et al. [4] has significantly lower throughput than our Offload Engine. Their method is to execute iSCSI on a Xilinx FPGA embedded platform, without any hardware accelerated parts. This method has very low CPU utilization, but throughput is limited by the low frequency of PowerPC 405 processor and does not pose technological challenge. The method of Chung-Ho Chen et al. [17] consists of offloading data-transfer iSCSI functions by using C-to-HDL translation process. The difference in design is that our iSCSI Offload Engine offloads not only data transfer functions, but also related non-data functions such as SCSI Command, SCSI Response, R2T and SNACK request. This has contributed to faster processing of requests and release of resources. Even though our iSCSI Offload Engine uses higher capacity host bus and higher clock frequency, it has higher level of parallelism since we used more CRC Generation Units. The ar-

chitecture of CRC circuit in Chung-Ho Chen et al. [17] uses simple linear feedback shift register, which is less efficient than the architecture of our CRC circuit [19]. Thus, our initiator is able to achieve 7 times higher throughput. The CPU utilization is similar as in Han-Chiang Chen et al. [4] and Chung-Ho Chen et al. [17], since both offload iSCSI to hardware in some terms.

5. Conclusion

In this paper we propose new iSCSI Offload Engine architecture for processing iSCSI data transfer and related non-data functions on an FPGA based adapter. The functions which do not affect performance are implemented with modified Open-iSCSI initiator on the host CPU. Our architecture overcomes the performance limitations imposed by a single processor which runs on 15 times higher operating frequency than our accelerator. It exhibits very low CPU utilization of approximately 3% on the host CPU, which is 10–15 times reduction compared with Open-iSCSI implementation. The maximum transmission throughput is 7.81 Gbps, while reception throughput is 7.34 Gbps, which is 2 times speedup over Open-iSCSI. It outperformed Chelsio's T110 ASIC-based HBA, with slightly higher throughput and significant decrease of 10 times of CPU utilization. Since some functions are implemented in modified Open-iSCSI, the architecture provides more system flexibility than a dedicated custom designed interface. Any new features, such as security functions, specification updates, CRC standards, etc., can be easily implemented.

References

[1] Satran, J., Meth, K., Sapuntzakis, C., Chadalapaka, M. and Zeidner, E.: Internet Small Computer Systems Interface (iSCSI), RFC 3720 (April 2004).

[2] Khosravi, H.M., Joglekar, A. and Iyer, R.: Performance characterization of iSCSI processing in a server platform, *24th IEEE International Performance Computing and Communications Conference, IPCCC 2005*, pp.99–107 (April 2005).

[3] Daugherty, J.: *Understanding iSCSI Digests: Accurately Evaluating the Cost and Risk of Disabling Digests*, JDSU Medusa Labs (2009).

[4] Chen, H.-C., Wu, Z.-J. and Wu, Z.-Z.: Implementation of Offloading the iSCSI and TCP/IP Protocol onto Host Bus Adapter, *Conference on Mass Storage Systems and Technologies* (2006).

[5] Sarkar, P., Uttamchandani, S. and Voruganti, K.: Storage Over IP: When Does Hardware Support Help?, *FAST '03 Proc. 2nd USENIX Conference on File and Storage Technologies*, pp.231–244 (2003).

[6] Open iSCSI, Open source iSCSI Initiator implementation, available from ⟨http://www.open-iscsi.org/⟩.

[7] Wireshark network packet analyzer, available from ⟨http://www.wireshark.org/⟩.

[8] Linux SCSI target framework, available from ⟨http://stgt.sourceforge.net/⟩.

[9] OProfile: system-wide profiler for Linux systems, available from ⟨http://oprofile.sourceforge.net/⟩.

[10] The UNH-iSCSI project, available from ⟨http://unh-iscsi.sourceforge.net/⟩.

[11] The Unified Wire Engine Introducing Terminator 3, A Chelsio Communications White Paper, available from ⟨http://www.chelsio.com/assetlibrary/products/T3_Unified_Wire_Eng_WP.pdf⟩.

[12] Dalessandro, D., Wyckoff, P. and Montry, G.: Initial Performance Evaluation of the NetEffect 10 Gigabit iWARP Adapter, *IEEE International Conference on Cluster Computing* (2006).

[13] Wu, Z.-Z. and Chen, H.-C.: Design and Implementation of TCP/IP Offload Engine System over Gigabit Ethernet, *Proc. 15th International Conference on Computer Communications and Networks, ICCCN 2006*, pp.245–250 (Oct. 2006).

[14] Jang, H., Chung, S.-H. and Yoo, D.-H.: Design and implementation of a protocol offload engine for TCP/IP and remote direct memory access based on hardware/software coprocessing, *Microprocessors and Microsystems Journal: Embedded Hardware Design* (*MICPRO*), Vol.33, No.5-6, pp.333–342, Elsevier Science Publishers (Aug. 2009).

[15] DINIGroup TCP Offload Engine IP: For Latency Critical, FPGA-based Embedded Networking Applications, available from ⟨http://www.applistar.com/wp-content/uploads/2012/06/TOE_Brief_v092.pdf⟩ (accessed April 2012).

[16] Intilop's 76-nanosecond TOE Based System Etablishes a Record 93% TCP/IP Bandwidth at a Major Customer's 10 G Network Deployment, available from ⟨http://www.sbwire.com/press-releases/10g-toe/tcp-performance/sbwire-156548.htm⟩ (accessed Aug. 2012).

[17] Chen, C.-H., Chung, Y.-C., Wang, C.-H. and Chen, H.-C.: Design of a Giga-bit Hardware Accelerator for the iSCSI Initiator, *Proc. 31st IEEE Conference on Local Computer Networks* (2006).

[18] Chelsio Communications T110 10-gigabit HBA: iSCSI HBA Performance Testing by VeriTest (June 2004).

[19] Akagic, A. and Amano, H.: Performance Analysis of Fully-Adaptable CRC Accelerators on an FPGA, *22nd International Conference on Field Programmable Logic and Applications — FPL 2012*, Oslo (2012).

[20] Akagic, A. and Amano, H.: A Study of Adaptable Co-processors for Cyclic Redundancy Check on an FPGA, *International Conference on Field-Programmable Technology — ICFPT 2012*, Seoul (2012).

[21] Akagic, A. and Amano, H.: High Speed CRC with 64-bit generator polynomial on an FPGA, *The International Workshop on Highly Efficient Accelerators and Reconfigurable Technologies* (*HEART*), Imperial College, London, UK (June 2011).

[22] Sawyer, N. and Defossez, M.: Quad-Port Memories in Virtex Devices, Application Note XAPP228 (v1.0), available from ⟨http://www.xilinx.com/support/documentation/application_notes/xapp228.pdf⟩ (accessed 2002-9-24).

[23] ML605 Hardware User Guide, v1.8, available from ⟨http://www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf⟩ (accessed 2012-10-2).

[24] Dual SFP+ FMC Module, available from ⟨http://hitechglobal.com/FMCModules/FMC_SFP+.htm⟩

[25] Xilinx ML605 Board Accessories, List of FMC Modules, available from ⟨http://www.xilinx.com/products/boards_kits/board_accessories.htm⟩

[26] Seelam, R.: I/O Design Flexibility with the FPGA Mezzanine Card (FMC), Xilinx WP315 (v1.0) (Aug. 2009).

[27] Xilinx LogiCORE IP 10-Gigabit Ethernet MAC v11.2, Xilinx UG773 (Oct. 2011).

[28] Xilinx LogiCORE IP XAUI v10.2, DS266 (Jan. 2012).

[29] Virtex-6 FPGA Configuration, UG360 (v3.2) (Nov. 2010).

Amila Akagic received her Dipl. el. Ing. and M.Sc. in Computer Science from Faculty for Electrical Engineering, University of Sarajevo, Bosnia and Herzegovina in 2006 and 2009, respectively. She is currently a Ph.D. student with Amano Laboratory at Keio University. Her research interests include computer architectures and reconfigurable systems.

Hideharu Amano received his Ph.D. degree from the Deparment of Electronic Engineering, Keio University, Japan in 1986. He is currently a professor in the Department of Information and Computer Science, Keio University. His research interests include parallel architectures and reconfigurable systems.

(Recommended by Associate Editor: *Shinsuke Kobayashi*)