Design of a Giga-bit Hardware Accelerator for the iSCSI Initiator

Chung-Ho Chen[a], Yi-Cheng Chung[ab], Chen-Hua Wang[a], and Han-Chiang Chen[b]
chchen@mail.ncku.edu.tw, carbooky@casmail.ee.ncku.edu.tw
elvik@casmail.ee.ncku.edu.tw, jolly@itri.org.tw
Department of Electrical Engineering and Institute of Computer and Communication Engineering
National Cheng-Kung University[a]
Taiwan, R.O.C.
Network & Communication Technology Center, Computer and Communication Laboratories
Industrial Technology Research Institute[b]
Tainan, Taiwan, R.O.C.

Abstract

We present the design of an iSCSI hardware accelerator for the initiator subsystem of a host bus adapter (iSCSI HBA). By analyzing the UNH-iSCSI open source code, first we evaluate the software performance and present a general methodology that transforms the software C code into the hardware HDL implementation. For the hardware module, the datapath design maximizes the concurrent accesses achievable within a clock cycle by using a dual-port descriptor memory. The synthesizable iSCSI hardware accelerator achieves 100 MHz speed and costs about 85K gates in the 0.18u technology. The design is able to meet the requirement of 1Gbps network when the average iSCSI PDU size is greater than 125 bytes.

1. Introduction

The Internet SCSI (iSCSI) is a newly developed network protocol that specifies the access to the SCSI storages over the TCP/IP network [1-4]. Until now iSCSI is mostly appeared in the solution of software implementations, the source codes released by Intel, IBM, UNH, etc., can be installed on the Linux operating system [5]. Several works [6][7] investigate the iSCSI protocol performance by comparing with other technology related to storage networks, such as the Fiber channel, NAS, NFS, etc. A software implementation of the iSCSI protocol bears the advantage of flexibility; however it consumes the host CPU utilization, especially for high bit rate networks. For instance, the CPU utilization for the iSCSI initiator during a WRITE operation reaches about 36% [8]. There are numerous developments of iSCSI offload engine that offloads the iSCSI protocol from the host CPU to the Host Bus Adapter (HBA). The methods are to use an embedded CPU on the HBA to run the iSCSI protocol. In [9], it proposes a method to build a switch and route subsystem to dispatch iSCSI PDUs to the processing end nodes. In [10], the work has proposed a method to extract the iSCSI PDU header from the TCP buffer.

Since the iSCSI is a data-intensive protocol that involves heavy checking for the data integrity during transmission [11], an effective hardware design that speeds up the process and relieves the burden imposed on the host processor is quite attractive, especially for Giga-bit storage systems. Our design addresses these issues and develops schemes to speed up the iSCSI protocol processing for the read/write operations, including iSCSI PDU creations, on-the-fly CRC checking, and PDU decompositions. The rest of this paper is organized as follows. Section 2 presents the performance profiling tool. Section 3 presents the initiator hardware architecture and a C-to-HDL translation methodology. Section 4 discusses the verification and performance of the design. This paper is concluded in Section 5.

2. Performance Profiling

To profile the performance, first the instrumented codes are added into the frequently used top-level functions. Two check-points (enter and leave) are added at the beginning and the end of every function. They generate messages into a log file when the monitored functions are called as shown in Fig. 1. When the execution flow meets the "check point," it invokes the "printk" function and the "do_gettimeofday" function to obtain the current time ticks measured in micro-second. Then it prints out the information as kernel messages to a log file.

An analyzing tool called *analyzer,* implemented in perl is designed to scan the log file line by line. An instance of one line in the log file looks as follows:

47755560->Enter Rx_data @1

where the leading number is the time ticks when dumping this log message. This number comes from a timer since the time the machine is booted up. The following "Enter" is the log type (it can alternatively be a "Leave" type). "Rx_data" is the function name of the logged function. The number appeared after '@' symbol is the referenced counts for this function. It is used to trace the Enter and Leave pair of the same function.

The analyzer checks each line in the log file, calculates

the average interval time for every invoked function, and counts how many times the function is used in the entire flow. This mechanism enables us to trace the program easily and determine the most frequently-used and time-consuming functions in the C code. Some functions are called iteratively in certain operation scenario. The analyzer obtains the iteration counts of the function used, the average time interval when calling a function, and the maximum and minimum iteration time interval the function is called.

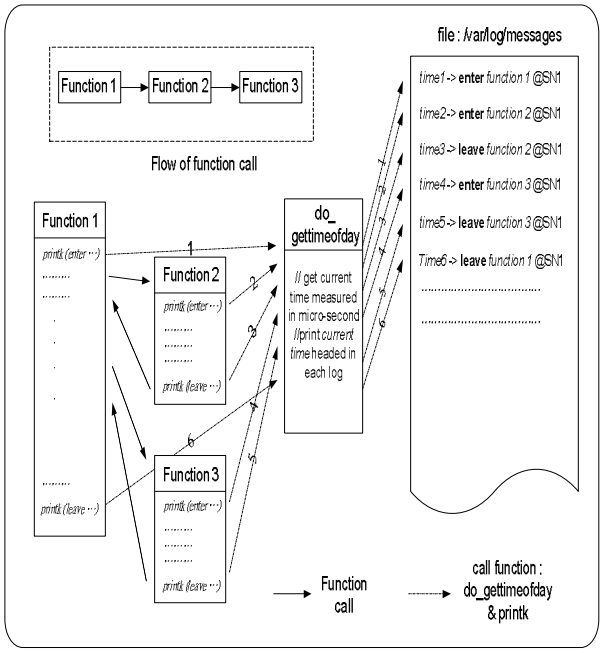

Fig. 1. Profiling of the function calls.

3. iSCSI Accelerator

Fig. 2 depicts the architecture of the implemented iSCSI initiator. The iSCSI initiator system comprises the following elements:

- Host interface and TCP interface
- Descriptor RAM
- Command management module
- Tx module
- Rx module
- Lookup tables

The descriptor RAM is used to keep the state information about sessions, connections, SCSI commands, pending commands, and iSCSI protocol data unit. The command management module deals with the process of transformation from SCSI command to iSCSI PDU, and maintains several data structures in the descriptor RAM. An iSCSI session uses one Tx module that performs all the PDU transmissions from the initiator to a target. The Rx module performs the PDU receptions from a target to the initiator. The details of these units will be addressed later. The other functions such as operations in the login phase are assumed to be executed by the host CPU or the embedded processor in the TCP/IP offload engine.

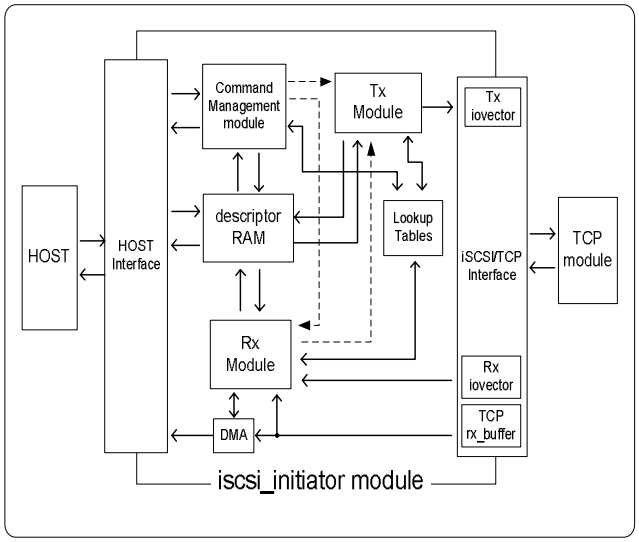

Fig. 2. Architecture of the iSCSI initiator.

There are three top modules in the design of the initiator accelerator. Each of the top modules comprises several submodules and each submodule is implemented in the same architecture as shown in Fig. 3. After the enable signal from the controller is asserted, the activated submodule is controlled by the step incrementor (or step counter) that functions similarly to an instruction pointer. The address generators generate the addresses for accessing the dual-port descriptor RAM from two pairs of the base and offset values through the multiplexers. This datapath and control design maximizes the concurrent dual access rate to the descriptor RAM in a clock cycle whenever possible. The rationale behind this design is to make memory accesses out of the bottleneck in the PDU process while keeping within a reasonable cost.

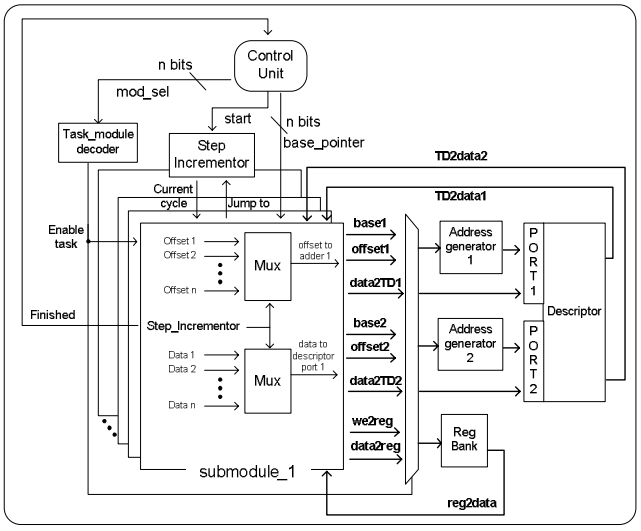

Fig. 3. Architecture of a single task module.

The job that a submodule has to do is to fill the required fields of data in the descriptor RAM. One submodule may communicate with each other by accessing the necessary data fields through the descriptor RAM. An entire C procedure is implemented in several submodules since the same submodule may be called in a particular scenario of the iSCSI operation, and may not be called in another one. Also, individual submodule can be shared by two different top modules.

3.1 C-to-HDL Translation

The way we implement the operations in a specific submodule is directly based on those implemented in the corresponding software function. Fig. 4 shows an example of translating the software operations into the corresponding hardware implementation. Except for observing the dependency, the statements in a C module are executed in parallel in a cycle, if possible, by the provided multi-cycled hardware datapath. The amount of the hardware parallelism to provide is determined by the total execution cycles constrained by the intended network throughput. Translating the functions in the software into the hardware submodules uses the following rules:

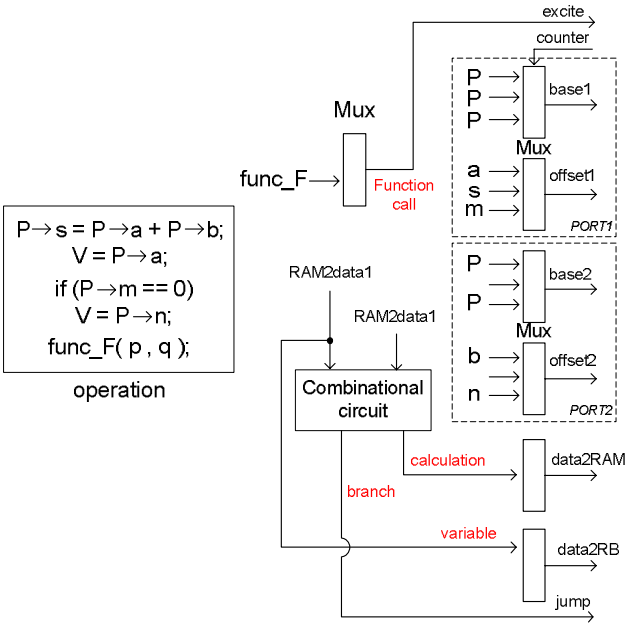

Fig. 4. C-to-HDL translation.

Mapping a variable to a register
A register bank that can be shared among submodules is used for keeping temporary values. Whenever we find the use of variables in the software implementation, these temporary values are allocated properly to registers. In our design, modules are classified into several levels. Each level may use up to five registers in its life time.

Mapping a pointer to a base value with offset
In the C version, a data structure is often used to describe the entity of a defined type. To translate a data structure, we allocate a "descriptor" space in the descriptor RAM to hold the same data structure. Whenever the access for an element (or a field) of the descriptor occurs, the address is generated by using the header of this descriptor (that is, the base) plus the relative position of the element (i.e., offset) in this descriptor.

Mapping a computation to a combinational circuit
The computations of two or more values are the fundamental operations found in the software implementation. Since it is expensive to instantiate extra ALUs, we deploy time-shared ALUs to perform the operations once the data are ready.

Mapping a branch to a jump
The statements in the source code may not be executed line by line due to conditional branches. This is resolved by notifying the step incrementor in the hardware to jump to the branch position whenever the executing flow encounters a branch.

Mapping function calls to submodule calls
We map the function calls in the software to the hardware submodule calls. When a main module needs to activate a submodule, it just has to give a number that represents the submodule outside the main module.

Mapping a linked list to a lookup table
When a data structure needs to be added to a linked list in software implementation, a queuing table in hardware is used to keep track of the relationship among them.

3.2 Control

The general architecture of a top module is shown in Fig. 5. It comprises a main module which is the caller of all its submodules inside the top module. All the modules in the top module are controlled by the module controller, which is implemented as a finite state machine, FSM.

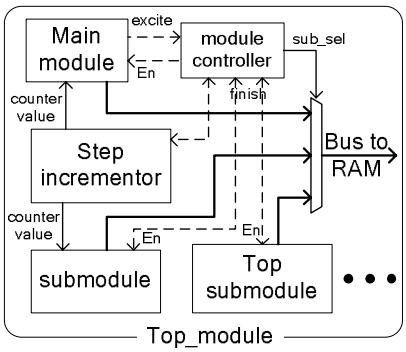

Fig. 5. Block diagram of a top module.

The FSM activates these submodules one by one according to the asserted signal from the main module. During the time the main module or a submodule is running, they all listen to an integer value from the step incrementor as

their program counter or sequence controller.

This step incrementor is duplicated in each top module design. All the submodules in the same top design share only one step incrementor, except the submodule that is also a top module instancing other low-level modules. The step incrementor performs the following jobs:

- The step incrementor outputs an integer value started from zero and incremented by 1 every clock period, to the activated submodule. The submodule performs a designated operation based on this integer value.
- When the step incrementor selects a pair of base and offset, an address is generated by adding up the base and offset.
- The step incrementor can issue a predefined integer value to a submodule according to the jump mechanism, and it can return to the state which is stored before the top design calling the submodule.

Fig. 6 shows the state diagram of the FSM for the control. The state machine stays at the IDLE state when the system is reset. Until the start signal is asserted, the state transits to the MAIN state. During the MAIN state, the main module is activated as the counter value in the step incrementor being advanced. The FSM won't transit to the next SUB_RUN state until the main module asserts the excite signal to indicate that a chosen submodule is responsible for doing the corresponding work. At the same time, the current counter's value is captured by a restore register. After the submodule has finished its task, the FSM proceeds to the RESTORE state to restore the step counter to the previous value before the main module calls the submodule. After the main module has reached the final step, the FSM goes back to the IDLE state which is the end of the top module.

An additional register bank that comprises 16 32-bit registers is used. This register file provides two ports for asynchronous reads and one port for synchronous writes. The purpose of this design is to minimize the amount of registers used if two different submodules can be arranged to occupy the same register at different time.

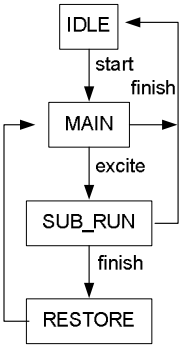

Fig. 6. State transition for the control.

3.3 Configurable CRC Implementation

With the increasing transmission rate to 1 Gb/s or 10 Gb/s, the CRC (cyclic redundancy check) computation becomes the performance bottleneck, especially when implemented in software. We design a configurable parallel CRC module that generates the CRC result by computing either 32-bit or 8-bit data within one clock period.

This parallel CRC module can deal with the input data of any length in byte. The CRC module can be attached to a data bus and generates the checksum for the data sequence at each clock period. This feature is referred to as *on-the-fly* CRC computation. The calculation is done when the data are in transmission. With this technique, no additional access is required when generating the CRC result. Fig. 7 depicts the implemented iSCSI CRC architecture.

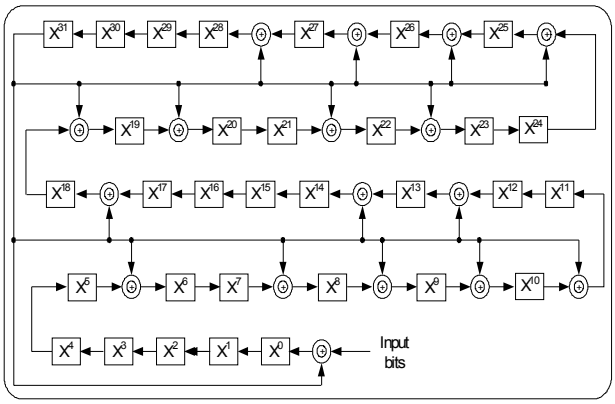

Fig. 7. iSCSI CRC architecture for G(x) = X32 + X28 + X27 + X26 + X25 + X23 + X22 + X20 + X19 + X18 + X14 + X13 + X11 + X10 + X9 + X8 + X6 + 1.

3.4 Command Management Module (CMM)

The command management module transforms the SCSI command to the iSCSI PDU and maintains the related data structures in the descriptor RAM. Fig. 8 shows the procedure flow of the CMM.

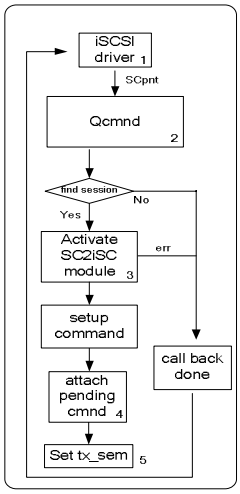

Fig. 8. Procedure flow in the command management module.

The iSCSI PDU is embedded in the iSCSI command data structure, which contains the necessary information to create the iSCSI PDU. The CMM module transforms the SCSI command to iSCSI command for further transmission by the Tx module.

3.5 The Transmit (Tx) Module

The performance of the iSCSI accelerator can be improved by means of duplicating the Tx module. Each iSCSI session occupies one Tx module that performs all the PDU transmissions from the initiator to a target. The Tx module can be activated by the CMM or the Rx module. For the first case, whenever an iSCSI command is properly filled in by the CMM, the Tx module is activated to check the pending command table for the newly setup command. If this command is a WRITE type command, the submodule "Build_Write_Cmnd" is called to fill the proper fields. Then, the "Rdy2Xmit" submodule is called to decide the transmit size. Finally, the "Do_Tx_send" is called to create a negotiation message format for the TCP layer, for this new PDU transmission. Fig. 9 illustrates the procedure flow for the TX module.

For the second case, if the Tx module is awoken by the Rx module, the "Rdy2Xmit" submodule is called to decide the transmit size according to the received "cookie." Then, the submodule "setup_dataoutPDU" is called to prepare the data which are solicited by the "cookie." Finally, the "Do_Tx_send" submodule is called. These actions are similar to those performed in the Tx_thread procedure.

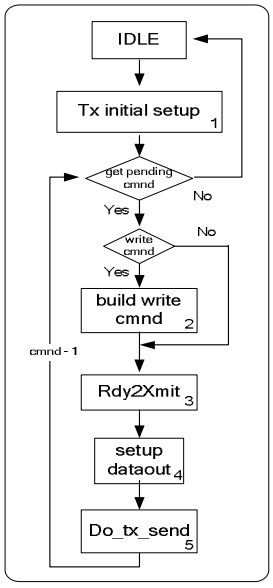

Fig. 9. Procedure in the Tx module.

3.6 The Receive (Rx) Module

The Rx module performs the PDU receptions from a target to the initiator. The iSCSI accelerator's performance can also be improved by using more of the Rx modules. When the Rx module is notified by the iSCSI driver that an iSCSI PDU is received in the TCP buffer, the Rx module is activated to receive just the size of a PDU header (by invoking recv_pdu_header) and check if the incoming PDU is a legal iSCSI PDU. Some checks (CRC, command SN, etc.) are done on the iSCSI header. After these routine checks on the PDU's header have been completed, an appropriate function, according to the opcode field in the PDU's header, is called to process the incoming iSCSI PDU. Fig. 10 shows the procedure for the Rx module.

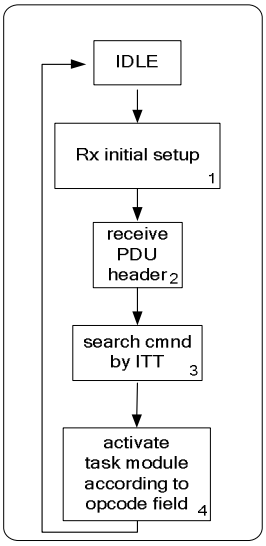

Fig. 10. Procedure in the Rx module.

3.7 Lookup Tables

There are four types of tables used for the communications among tasks. They are *session table*, *pending command table*, *pending cookie table*, and *Iovector table*.

The *session table* is used for looking up sessions when the CMM receives a target ID from the iSCSI driver.

The *pending command table* is used for holding the pointers of outstanding command tasks. The pending command table is added by the command management module, and removed by the Tx module or the Rx module.

The *pending cookie table* is used for queuing the information received from the R2T PDU, and the Tx module can send the next "DataOut" PDU, according to the received cookie entry. The *Iovector table* is used for collecting the pairs of header and length of the data to be sent out from the iSCSI initiator system.

4. Verification and Performance Evaluation

To verify the functionality of our design, we compare the output results of the software version with that of the hardware version given the same inputs. We have synthesized our design using the UMC 0.18u technology. The design is able to operate at the speed of 100MHz in the system clock. The gate count of the respective module is as

follows:
top_Qcmnd module = 12’816.
top_TX module = 35’095.
top_RX module = 37’338.
The total area is 85’249.

The software iSCSI runs on a system illustrated in Table 1. We estimate the performance of each function call in the software implementation based on the performance profiling tool presented above. From this, the average time consumed when calling a certain function can be measured.

Table 1. Measurement Environment for Software iSCSI.

Initiator/Target	CPU: Pentium III 1GHz, 512MB memory, Linux Kernel 2.4.20-8.
Network	Ethernet 100Mbps
SCSI adapter/disk	TekramDC-390U3W/ Hitachi10000RPM/8MB/Ultra 320.

For the hardware accelerator, since the required clock cycles to finish a specific task module is a fixed value (e.g., the AtchPndCmnd module needs 15 clock cycles, HR2TC module needs 5 clock cycles, etc.), it is straightforward to obtain the cycle counts of the measured operation. The results of the three modules are shown in the Fig. 11 to Fig. 13 respectively. Due to the effectiveness of the translation, the hardware implementation even with a very low clock rate still performs much better than that of the software version.

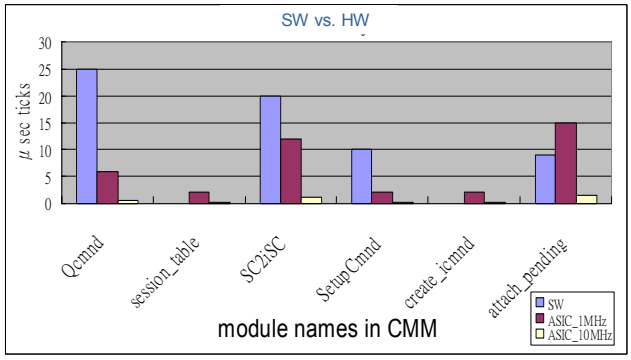

Fig. 11. Execution time for the CMM modules.

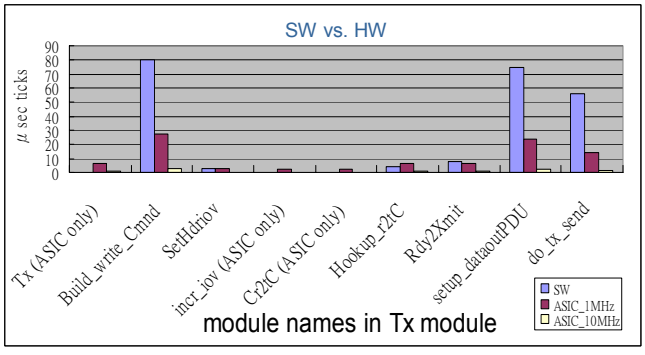

Fig. 12. Execution time for the modules in Tx.

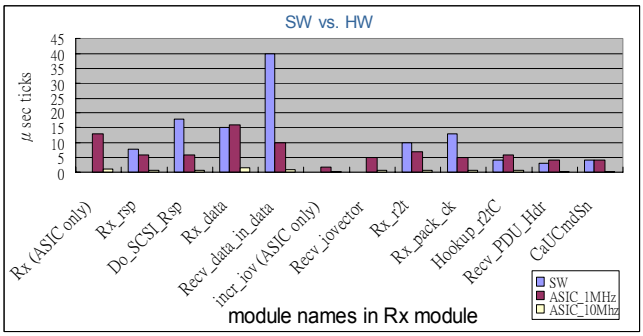

Fig. 13. Execution time for the modules in Rx.

Note that the graph also shows which function requires the longest execution latency. For instance, in the Tx module, the execution time of the Build_write_Cmnd operation is the largest while in the Rx module, it is the Recv_data_in_data operation that takes the longest execution time.

To compare read/write performance without the impact of the network, Fig. 14 shows the required time spent in read/write operations assuming zero network latency. The read operation is the command issued by the initiator to read data from the target while the write operation is issued by the initiator which writes the data to the target.

The time evaluated for each bar is acquired from the summation of multiplying the iterations with the time consumed by every function (module) for the operation. Note that the hardware implementation has drastically reduced the execution time in comparison with the software version. For a one-Gbps network, sending the shortest PDU (52-byte header only) needs 416 ns. The current 100 MHz design with dual-port descriptor RAMs requires 1000 ns to prepare the PDU. For an IP storage system, the length of the transmitted PDUs is expected to be much larger than that of the minimum PDU. Thus, this design is able to meet the one-Gbps requirement when the average PDU length is greater than 125 bytes, including the header.

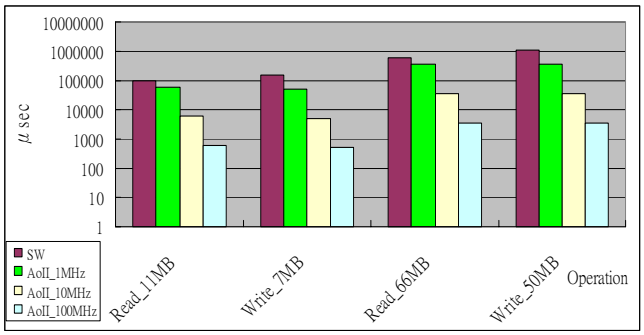

Fig. 14. Execution time in read/write operations.

5. Conclusion

This paper presents the design and analysis of an iSCSI initiator accelerator. We have described a C-to-HDL translation methodology to realize the design. First, a profile

analyzer is used to extract the execution time and frequency of the major procedure calls in the iSCSI C code. The most time consuming modules are identified and implemented in the hardware. For the hardware module, the datapath design maximizes the parallelism achievable within a clock cycle in using the descriptor dual-port memory. The iSCSI hardware accelerator achieves 100 MHz speed and costs about 85K gate counts in the UMC 0.18 technology. This design meets the requirement of one Gigabps when the average PDU length is greater than 125 bytes.

Acknowledgements

The work in this paper was in part supported by the National Science Council, Taiwan, under NSC 94-2220-E-006-004.

References

[1] John L. Hufferd, "iSCSI The Univeral Storage Connection," Addison-Wesley, ISBN 0-202-78419-X, 2002.

[2] Internet Small Computer Systems Interface (iSCSI), RFC 3720, http://www.ietf.org/rfc/rfc3720.txt.

[3] Kalman Z. Meth and Julian Satran, "Design of the iSCSI Protocol," Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'03).

[4] Mallikarjun Chadalapaka, "iSCSI State Diagrams," Networked Storage Architecture, NSSO, Rev 0.7, Jan. 07, 2002.

[5] UNH-iSCSI software code, http://sourceforge.net/projects/unh-iscsi

[6] Stephen Aiken, Dirk Grunwald, Andrew R. Pleszkun, and Jesse Willeke, "A Performance Analysis of the iSCSI Protocol," Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'03).

[7] Dimitrios Xinidis, Angelos Bilas, and Michail D. Flouris, "Performance Evaluation of Commodity iSCSI-based Storage Systems," Proceedings of the 22nd IEEE/ 13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'05).

[8] Anshul Chadda and Ro D. Russell, "Design, Implementation, and Performance Analysis of Session Layer Protocols for SCSI over TCP/IP," *Tech. Report TR 01-06*, University of New Hampshire, August 2001.

[9] William Todd Boyd, Douglas J. Joseph, Michael Anthony Ko, and Renato John Recio, "iSCSI Driver To Adapter Interface Protocol," US Patent # 20040049603.

[10] Shay Mizarchi, Rafi Shalom, and Ron Grinfeld, "iSCSI Receiver Implementation," US patent # 20030058870.

[11] R.J Glaise and X. Jacquart, "Fast CRC Calculation," IEEE International Conference on Computer Design, pp. 602-605, 1993.