An Implementation of Special Purpose SSD Device

Songyan Liu(✉), Shangru Wu, Ting Chen, and Cheng Zhang

Electronic Engineering College, Heilongjiang University,
Xuefu Road 74, Harbin 150080, China
liusongyan@hlju.edu.cn,
{2141258,2151302,2161419}@s.hlju.edu.cn

Abstract. Under the background that SSD is more and more popular, this paper shows an efficient implementation of an SSD device designed for special function and interface on Xilinx SoC platform. The Device uses MLC NAND Flash as storage chips and uses Xilinx's Zynq-7000 series SoC as the processor. The device adopts the method of multi-channel parallel data transmission and pipeline operation to achieve high performance. Enhanced ECC checking ability is provided to against the flash internal errors. The storage system also uses the RAID5 architecture to improve reliability significantly. Finally, the test results show that the designed SSD storage device reaches the expected performance and reliability.

Keywords: SSD · DMA · Linux · FPGA

1 Introduction

Data storage plays an important role in today's information society. With advantages in performance, low-power and reliability, SSD (Solid-State Disk) is more and more popular in data storage domain. As the price drops, the storage scheme based on NAND Flash is becoming increasingly attractive relative to traditional mechanical disk scheme in vast application area. Under this background, more and more needs of customization of various application and interface happen.

Authors in reference 1 propose a RAID0 array storage scheme to improve the transmission performance of the solid-state drives at the interface of SATA II [1]. Authors in reference 2 and 3 combines SSD with the PCI Express bus, PCI Express basics and different PCI Express SSD architectures are reviewed. Finally, they present an overview on the standardization effort around PCI Express [2, 3]. Authors in reference 4 put the SSD device together with the AHB bus and an 8 channels transmission mechanism was put forward to improve the transmission speed [4].

This paper shows an SSD device which is optimized for volume, power consumption and none general purpose computer dependency, with an additional board to extend various interfaces like Ethernet, USB and Rocket IO. Linux DMA driver implementation is emphatically discussed.

The rest of this paper is organized as follows. Section 2 gives the hardware design of SSD device and the software design for the device in Sect. 3. Tests and results of the device are showed in Sect. 4. Finally, conclusion is in Sect. 5.

2 Hardware and Architecture Design

The hardware contains two boards: an SSD board and a download board. The SSD board focusses on optimized volume and power consumption for embedding into the special purpose device. The download board is designed to download data from SSD and forward it to various bus interfaces like network and USB. Download board is also a test platform for testing the SSD. Two boards connect through the FMC (FPGA Mezzanine Card) interface. The overall architecture of the system is shown in the Fig. 1.

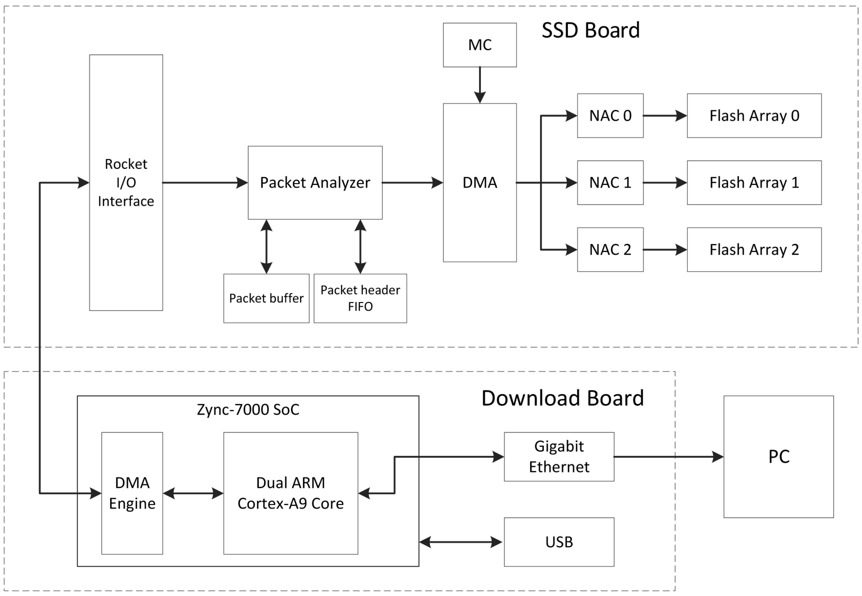

Fig. 1. The overall system diagram.

2.1 Module Partition of SSD Board

The SSD hardware can be divided into following modules:

Packet Analyzer. The data are transmitted in the form of packet. Packet analyzer is responsible to deal with the packet, extract packet header information and store it in packet header FIFO for subsequent processing of software. Packet buffer is a true dual-port SRAM, which makes it easy to move data between each end in parallel.

DMA Controller. The main task of DMA controller is data transmission, but it also handles transparent RAID function. Data that reach SSD disk is first deposited in the packet buffer. Data is transmitted from the packet buffer to the page buffer which controlled by NAC via DMA, and then wrote into the Flash arrays.

MC (Master Controller). It is responsible for the NAND Flash management works such as bad block management and wear-leveling. It also handles packets sent by the upper machine and forwarding to flash array with help of Packet analyzer and DMA. Data pipeline and parallel stream of every logical channel are organized in MC.

NAC (NAND Array Controller). The number of NACs depends on tradeoffs of capacity, performance and cost. Each NAC controls 8 Flash chips as a logical channel, so that it can store data in the Flash arrays in parallel efficiently [5].

2.2 Hardware Design of Download Board

The processor of download board is Zynq-7000 series SoC which produced by the Xilinx company. Zynq-7000 products incorporate a dual core ARM Cortex-A9 based Processing System (PS) and Xilinx Programmable Logic in a single device [6]. It contains a Xilinx IP AXI Direct Memory Access (AXI DMA) core, which provides high-bandwidth direct memory access between memory and AXI4-Stream target peripherals [7]. Its optional scatter/gather capabilities also offload data movement tasks from the CPU. Besides, this system has a serial port and a network port to debug and test procedures. There is a FMC port to connect with the SSD board.

2.3 Hardware Overview of AXI DMA

To achieve enough performance, this paper uses scatter/gather mode of AXI DMA to transmit data. Scatter/gather mode can put the discrete memory together as one descriptor used for transmission [8]. AXI DMA can be divided into MM2S (memory-mapped to stream) used to send data and S2MM (stream to memory-mapped)

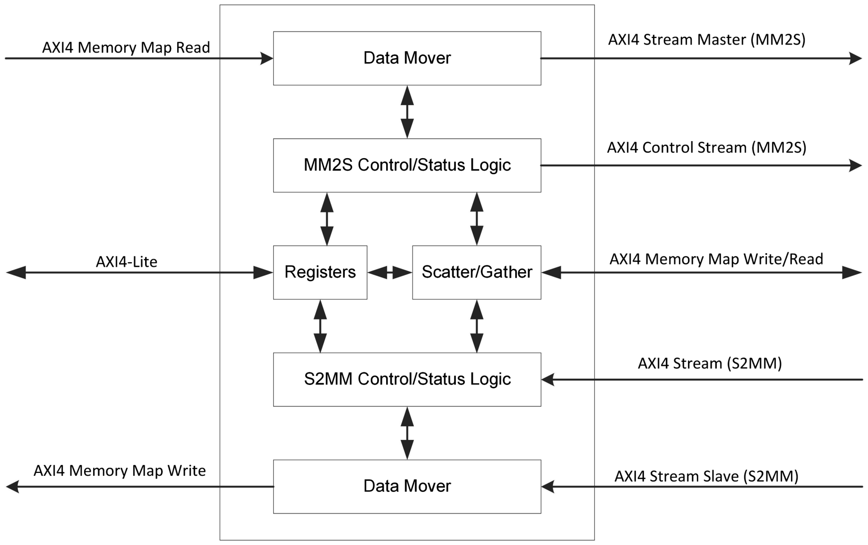

Fig. 2. The AXI DMA hardware block diagram.

used to receive data. To send data as an example, put the first section of memory physical address in MM2S_CURDESC register, and the last section of memory address in MM2S_TAILDESC register, then give '1' to the lowest position of MM2S_DMACR register to start DMA, data can be transmitted from download board to SSD board. Receiving data is similar. The block diagram of AXI DMA is shown in Fig. 2.

3 Software Design of Download Board

3.1 Linux Kernel DMA Framework

The primary components of the Linux kernel DMA framework include the DMA device control together with memory allocation and cache control.

Memory Allocation. Memory allocation is needed to provide data buffer for DMA, but also limited by some condition of continuity and consistency. There are various methods to allocate contiguous memory in Linux. The `kmalloc()` function allocates cached memory which is physically contiguous. It is limited in the size of a single allocation. The `dma_alloc_coherent()` function allocates non-cached physically contiguous memory. It uses a new feature of Linux kernel called Contiguous Memory Allocator (CMA) since version 3.5, and allows very large amounts of physically contiguous memory to be allocated. We adopt `kmalloc()` function to allocate memory, so the cache control should be considered that is described below.

DMA Cache Control. What modern CPU direct access is cache, but DMA direct accesses memory, this causes the cache consistency problem. So the data buffer that DMA used should be non-cached, or, cache must be flushed and invalidated at the right time to assure the validity of data. Linux provides DMA functions for cache control of DMA buffers. Function `dma_map_single()` is provided to transfer ownership of a buffer from the CPU to the DMA hardware. It can cause a cache flush for the buffer in memory to device direction. Function `dma_unmap_single()` is provided to transfer ownership of a buffer from the DMA hardware back to the CPU. It can cause a cache invalidate for the buffer in the device to memory direction.

DMA Device Control. The DMA driver is designed to be a multithreaded and asynchronous program to improve performance [9]. Sending thread uses the completion mechanism to know if a DMA transfer task is done. The tasklet mechanism is used to send the completion signal when DMA transmission complete interruption comes, which replaces older bottom half mechanisms for drivers. Some DMA slave API use a piece of opaque data called cookie to exchange communicating information. For example, a DMA cookie is returned from `dmaengine_submit()` and is passed to `dma_async_is_tx_complete()` to check for completion of a specific DMA transaction.

3.2 Linux DMA Engine Slave API

The integral flow of DMA operation using Linux DMA engine slave API is shown in Fig. 3:

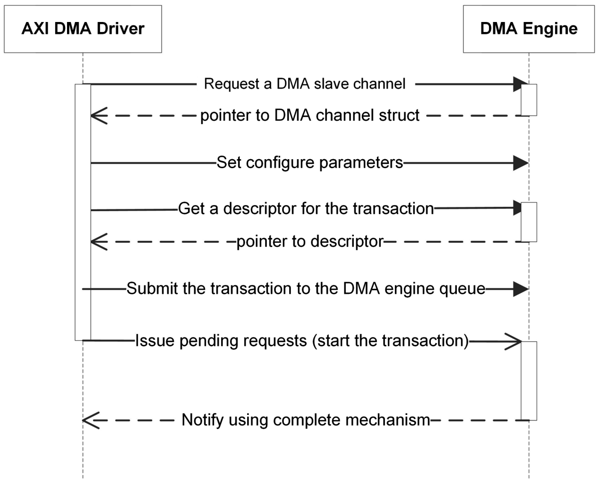
AXI DMA Driver
DMA Engine
Request a DMA slave channel
pointer to DMA channel struct
Set configure parameters
Get a descriptor for the transaction
pointer to descriptor
Submit the transaction to the DMA engine queue
Issue pending requests (start the transaction)
Notify using complete mechanism

Fig. 3. The operational flow of DMA engine slave API.

1. Use `dma_request_channel()` to request a DMA channel, for AXI DMA, the 1st channel is the transmit channel and the 2nd channel is the receive channel.
2. Set slave and controller specific parameters, include DMA direction, DMA addresses, bus widths, DMA burst lengths etc.
3. `dmaengine_prep_slave_single()` function gets a descriptor for a DMA transaction, which is the key data structure of a DMA transfer.
4. The `dmaengine_submit()` function submits the descriptor to the DMA engine to be put into the pending queue.
5. The `dma_async_issue_pending()` function is used to start the DMA transaction that was previously put in the pending queue.
6. Program waits for the DMA transfer done on a completion.

3.3 Data Stream in DMA Driver

Read Data. When the user reads the data, it will first send a packet include read command to SSD. When the SSD receives the command, host controller of SSD parses the command, read the NAND Flash chip on specific addresses, and sent it to the download board via DMA. After receiving the data, the data will be sent to the computer through the network port. The specific process as showed in Fig. 4.

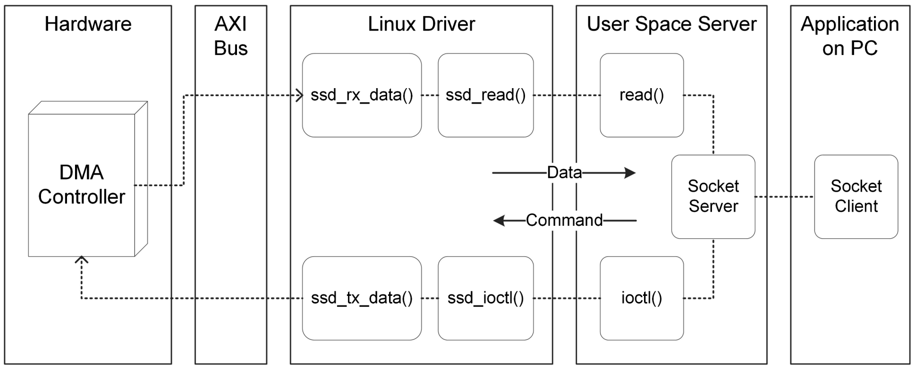

Fig. 4. The flow chart of read data.

1. Application on PC send the read command via Ethernet. The user space server calls the standard ioctl interface to send command and the read interface to read data.
2. DMA driver use `ssd_ioctl()` and `ssd_read()` functions to implement the read and ioctl interface, which invoke `ssd_tx_data()` function to send the command packet, and `ssd_rx_data()` function to receive data.
3. Start receiving DMA and then send a packet includes read command.
4. Command arrives at the SSD. SSD parses the command, reads data from Flash chips and send them via DMA.

Write Data. Compared to the read operation, write data operation is more simple. When the application calls the write function, the Download Board will prepare the data and send it to the SSD. SSD receives the data and puts it into Flash chips.

1. The application send write the command. Server use write interface to call the underlying `ssd_write()` function.
2. SSD_write function calls the `ssd_tx_data()` function.
3. Start DMA to send the prepared data to the SSD.
4. SSD receives the data and stores it into Flash chips.

3.4 Interrupt Handling

This function is so important because the processing will impact the performance of the system directly. The interruption of this driver can be divided into three types: the transmission complete interruption, transmission error interruption and transmission delay interruption. The device can only generate one interruption at a time. The 13, 14, 15 bits of transmission status register are the flag bits of these interrupts. When an interrupt generates, the driver determines if an interrupt is triggered by reading the status register and calls the corresponding function. Whether to send data or receive data, when a descriptor transmission is complete, a transmission complete interrupt will be triggered. When the complete interrupt happens, the driver executes interrupt handlers to judge whether there is a descriptor needs to transfer, if it does, the driver continues the transmission, if not, it ends the transmission. When hardware generates a transmission error interrupt, which indicates transmission error, the program will release resources and print warning information. Transmission delay interruption will happen within a certain time interrupt is merged into an interrupt notifies the user. The specific process is shown as Fig. 5.

1. The system generates an interrupt, read the value of the status register.
2. Use a spin-lock to mask other interruptions.
3. Determine the type of interrupt by reading the state of the register values.
4. Executes interrupt handler function [10].
5. Release resources and open spin-lock.

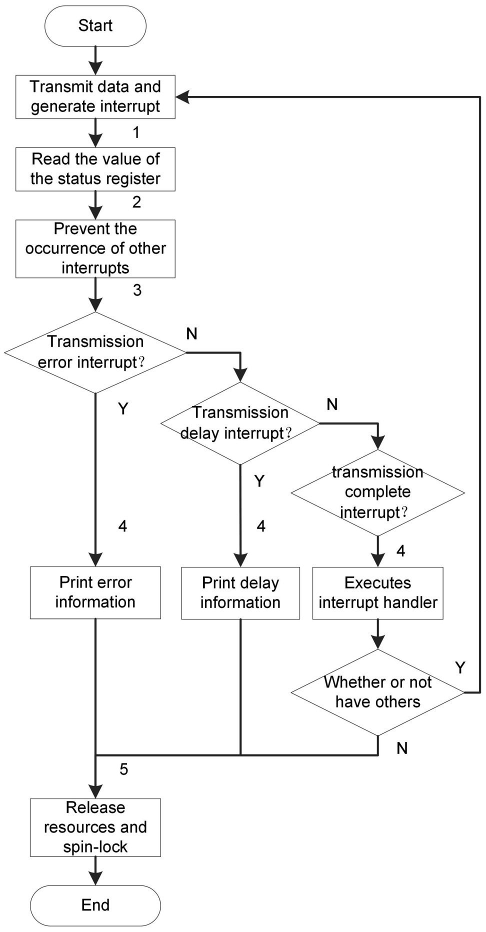
Start
Transmit data and generate interrupt
1
Read the value of the status register
2
Prevent the occurrence of other interrupts
3
Transmission error interrupt?
N
Y
Transmission delay interrupt?
N
Y
transmission complete interrupt?
4
4
4
Print error information
Print delay information
Executes interrupt handler
Whether or not have others
Y
N
5
Release resources and spin-lock
End

Fig. 5. Interrupt service routine.

4 Tests and Results

The AXI bus connectivity, data communication and DMA functionality have been tested using Xilinx ChipScope Pro Analyzer. The software test can be divided into two aspects: correctness test and performance test. For correctness test, a large file contains random data can be prepared in advance. The file is written to SSD and read out again, and a byte-for-byte comparison of two times file will find any data corruption. For the performance test, operating any prepared data on SD card or network is too slow to test. And data transfer between user space and kernel space is also intolerable overhead. So, a test mode is implemented in DMA driver directly. Small amount of data/buffer is prepared in memory, and sent/receive to/from SSD in a cycle. Getting system time at the beginning and end of test, and performance is obtained by total data that sent/received divide total time. For configuration of two and three channel write, measured throughput between SSD and download board reach 227 MB/s and 357 MB/s. For configuration of two and three channel read, measured results reach 240 MB/s and 374 MB/s.

5 Conclusion

The needs for high speed storage become more and more extensive, SSDs designed for general purpose do not apply to any domain. This paper proposed a software and hardware co-design architecture of special purpose SSD device based on SoC. It's two board design is verified that can balance the customized requirements and extendability efficiently. We focus on the exposition of a DMA driver that controls a DMA engine implemented by Xilinx AXI DMA IP. The results of tests show an ideal performance is achieved. Since the architecture of this design is flexible, we will focus on performance improvement, more interfaces and more application scenarios in further works.

References

1. Eshghi, K., Micheloni, R.: SSD architecture and PCI express interface. In: Micheloni, R., Marelli, A., Eshghi, K. (eds.) Inside Solid State Drives (SSDs), pp. 145–148. Springer, Netherlands (2013). https://doi.org/10.1007/978-94-007-5146-0_2
2. Hung, J.J., Bu, K., Sun, Z.L., Diao, J.T., Liu, J.B.: PCI express-based NVMe solid state disk. J. Appl. Mech. Mater. **464**, 365–368 (2013)
3. Yu, Z.L., Hua, J., Feng, L.: The design and implement of SSD chip with multi-bus and 8 channels. J. Appl. Mech. Mater. **58–60**, 2592–2596 (2011)
4. Zhang, P., Wang, X.: SSD performance analysis and RAID 0 scheme design. J. Microcomput. Appl. (2016)
5. Zertal, S.: Exploiting the fine grain SSD internal parallelism for OLTP and scientific workloads. In: 2014 IEEE International Conference on High Performance Computing and Communications, 2014 IEEE International Symposium on Cyberspace Safety and Security, 2014 IEEE International Conference on Embedded Software and System, pp. 990–997. IEEE (2014)

6. Crockett, L.H., Elliot, R.A., Enderwitz, M.A., Stewart, R.W.: The Zynq book: embedded processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 all programmable Soc. Strathclyde Academic Media, Glasgow (2014)
7. Na, S., Yang, S., Kyung, C.: Low-power bus architecture composition for AMBA AXI. J. Semicond. Technol. Sci. **9**, 75–79 (2009)
8. Cutting, D.R., Karger, D.R., Pedersen, J.O.: Scatter/gather: a cluster-based approach to browsing large document collections. In: International ACM SIGIR Conference on Research & Development in Information Retrieval, pp. 318–329. ACM (1996)
9. Lian, P.P.: Multi-thread chain DMA data transfer method for SuperSpeed bus video transmissions. J. Adv. Mater. Res. **1046**, 277–280 (2014)
10. Lee, J., Park, K.H.: Interrupt handler migration and direct interrupt scheduling for rapid scheduling of interrupt-driven tasks. J. ACM Trans. Embed. Comput. Syst. (TECS) **9**, 1–34 (2010)