IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2022.Doi Number

Research and Design of Nand Flash Array from Host to Flash Translation Layer

BOYANG DING[1], SONGYAN LIU[2], TAO LV[2], YAO LIU[2], and WENBIN LU[2]
School of Electronic Engineering, Heilongjiang University, Harbin, Heilongjiang 150080, China

Corresponding author: Songyan Liu (liusongyan@hlju.edu.cn)

ABSTRACT Given the inherent limitations of flash memory, solid-state storage devices require a host controller and a flash translation layer (FTL) to address two major conflicts: the conflict between the limited erase endurance of flash memory and the expectation of longer usage time and the conflict between the insufficient per-die bandwidth of flash memory and the exponential growth in data throughput.This paper presents a hybrid architecture implemented with FPGA logic and embedded processors. FPGA hardware acceleration is utilized to meet the requirement of high bandwidth, while the Host-FTL flash translation layer architecture is used to address the varying workload demands. By separating the storage device from the flash translation layer, the host manages the flash channel using the command and message units provided by the system.The design of Host-FTL not only implements conventional software algorithms such as address mapping, wear leveling, and bad block management but also uses a "pipeline" strategy for regular writes and a "parallel page group" strategy for large file writes, after analyzing the bandwidth bottleneck of the system. The channel-level RAID array enhances data security, and the localized wear leveling increases the total amount of written data in the solid-state disk array.

INDEX TERMS NAND Flash; HOST-FTL; FPGA; Wear leveling; Solid-state storage.

I. INTRODUCTION

In solid-state drives, the firmware running in the solid-state controller manages all the flash chips, creating a virtual management layer called the Flash Translation Layer (FTL). With the development of solid-state storage technology, FTL has multiple implementation methods, and its structure is becoming more modular and diverse[1, 2]. The traditional implementation method is Device-FTL, which implements FTL internally in the device. However, due to the resource constraints of solid-state storage devices, FTL is seeking new solutions[3, 4]. Host-FTL is a novel implementation method for FTL, which implements many flash management tasks on the host side. It is currently a new direction in flash management and the most suitable solution for high-capacity solid-state storage[5].

This article uses FPGA as the controller to manage the flash channel, optimizes the performance of the flash translation layer, and presents it in the form of Host-FTL, which is beneficial for the implementation and management of large-capacity storage systems and has important implications for the promotion of all-flash arrays.

This project mainly focuses on the implementation of a new high-speed solid-state storage disk, which includes internal instruction processing, command allocation, status feedback, and data protection[6]. The Host-FTL algorithm involves address mapping, parallel instruction pipelining, wear leveling, and so on. Based on the actual storage hardware platform, a processor softcore is built inside the FPGA to allocate tasks to the processor, and the instruction flow and data replication process inside the solid-state disk are analyzed and studied. Finally, in combination with the operating system, the performance of the designed solid-state disk is tested, and improvement proposals are put forward.

The flash memory management mechanism of Host-FTL includes the address mapping mechanism, optimization process of the read-write pipeline, wear-leveling algorithm, write allocation mechanism applicable to Host-FTL, and software-level data protection. Host-FTL delegates the flash memory's read-write interface to the driver program, enabling the host to complete the FTL work, and only requires the implementation of ECC error correction, command response, and flash channel control within the solid-state disk.

II. NAND FLASH Array Controller Design

This system utilizes an actual solid-state drive hardware platform, where the traditional FTL is divided into a driver layer and a firmware layer[7]. The firmware layer is responsible for handling address mapping transactions and receiving commands from the upper layer, while other FTL transactions are executed on the host side. By leveraging the host's powerful computing capabilities and ample memory resources, the storage performance of the solid-state drive can be fully optimized. The internal resources of the solid-

state drive are thoroughly analyzed, and a combination of write allocation strategy on the host side and internal

The original functional division diagram of the system is shown in Figure 1. The FPGA is divided into a processor section and a logic section, with the logic section responsible for constructing the control circuit of the NAND array, configuring the hardware RAID structure, and managing some peripheral devices. The processor section handles command parsing, data packet transmission and reception, and other tasks. The system is connected to the host via a PCIe interface, and a driver with FTL functionality is loaded on the host to control the device.

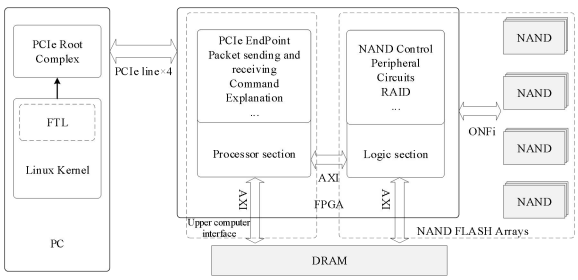


FIGURE 1. **System function partition block diagram**

The system uses Micron's MLC NAND flash memory chips, with a capacity of 256Gbit per chip or 32GB per chip and a total of 32 flash chips across four channels on the solid-state drive. The flash chip specification includes a page size of 17600 bytes, with a data area of 16KB and 512 pages making up a block. Additionally, 1024 blocks comprise a plane. Depending on the capacity of the flash chip, a die (also known as a logical unit or LUN) can contain 2 to 4 planes, and a chip package can contain 2 to 4 dies. Within each plane, there are two high-speed buffer registers that can be used to pipeline advanced commands such as write-back and re-read, increasing the degree of parallelism of the operations[9]. The system architecture is illustrated in Figure 2.

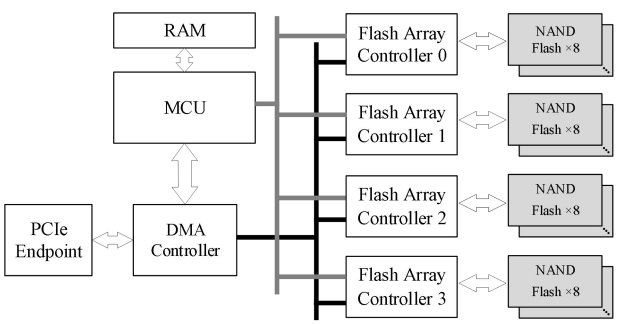


FIGURE 2. **Schematic diagram of system connection 123**

The system consists of several main modules, including a PCIe controller, a DMA controller, a message unit, global registers, four NACs (NAND Array Controllers) corresponding to four channels, and an MCU (Master Controller Unit) for processing commands. The NACs operate at a clock frequency of 160MHz, while the MCU operates at a clock frequency of 120MHz. A temperature sensor is connected via SPI to measure the temperature of the

command scheduling in the firmware layer enables read-write optimization without compromising data integrity[8].

junction, and system-related information is stored in NVRAM. The various modules are connected via an AHB-Lite bus operating at a clock frequency of 78.125MHz with a 32-bit data width, and external DDR memory is attached to the FPGA. All data paths are connected using a 256-bit bus operating at a clock frequency of 78.125MHz, with data transfer being accomplished via DMA[10]. The internal hardware design of the hard disk controller is shown in Figure 3.

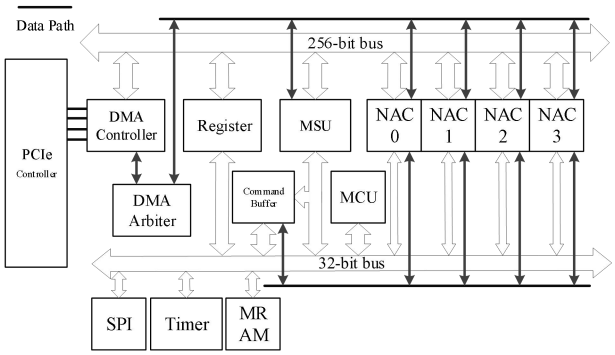


FIGURE 3. **System processor design block diagram**

A. Design of PCIe Interface Unit

The SSD is connected to the Linux PC via the PCIe bus interface[11], creating a 4-lane PCIe physical link on which DMA operations are performed to obtain the maximum bandwidth boost[12]. PCIe devices want to communicate with the Host, mainly through a packet called TLP (Transaction Layer Packet) for data exchange. We build a BAR (Base Address Register) space in the device. After power on, the system software reads these BARs, allocates the corresponding system memory space for them, and writes the corresponding memory base address back to the BAR. The address of BAR is actually the address of the PCI bus domain, and the address of the memory domain is accessed by the Host; when Host accesses the PCIe device, it needs to convert the bus domain address into the address of the memory domain. We can access the registers in this space by configuring read/write TLP.

The LECS space includes power management, PCI Express, and MSI (Message Signaled Interrupt) function structures. The power management function works with the temperature sensor on the chip to reduce power consumption and speed up heat dissipation during overheating, and the temperature sensor uses the SPI bus to connect to the FPGA. The power management information is configured through the 32-bit PMC (Power Management Capability) register and the PSCMR and DATA registers.

Host writes the corresponding data to the MSI configuration register, a way to send interrupts to the device, mainly containing task completion interrupts, temperature alarms, internal fatal errors, etc. MSI control is in the PCI

space domain, where the mask bits, message data, and high/low message addresses are controlled by the Host. PCIe function, on the other hand, records the parameters and capabilities of the device in the register and is Host's entry point for obtaining configuration information and adjusting storage disk parameters.

B. Messaging and Control Unit

In the system, although the end device does not deal with flash memory management, it must have a set of message and control units for fast information exchange with the Host to pre-process Host's commands. As shown in Figure 4, the message unit transmits command frames and status frames, and the end device can also return its own situation through the corresponding interrupts.

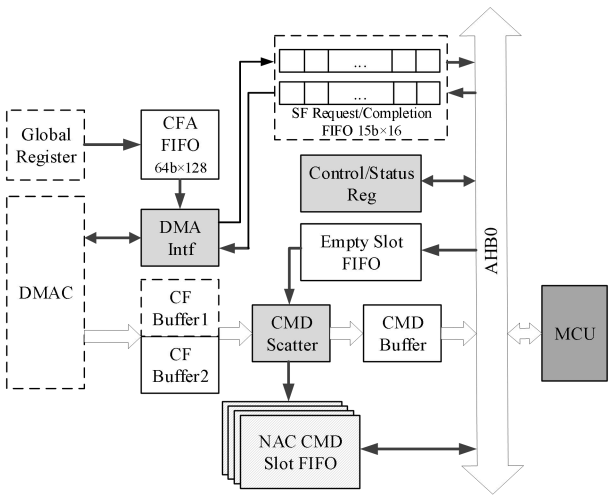


FIGURE 4. MSG Unit connection diagram 123

The connection between Host and the device mainly consists of commands and status information, which are transmitted by the MSG unit. Commands are transferred from the Host to the device by the MSG unit, and status information is pushed directly from the device to the Host by the MSG unit. The transmission pattern between the Host and the device is shown in Figure 5.

To improve transfer efficiency, more than one command or status item is usually transferred at a time, and command and status information is transferred in a batch mode. Commands are merged into a structure called Command Frame, and status entries are merged into a structure called Status Buffer and then transferred to the Host, both of which require a transfer operation via MSG. Here, a status entry corresponds to a command. However, a status frame does not always correspond to a command frame, and status entries in the same status frame may come from different CFs. In addition, the device will generate an interrupt to the Host to notify that a new status frame has arrived. All encodings in the system are based on the small-end format, with Byte0 being the least significant byte. The transmission pattern of CFs is shown in Figure 6.

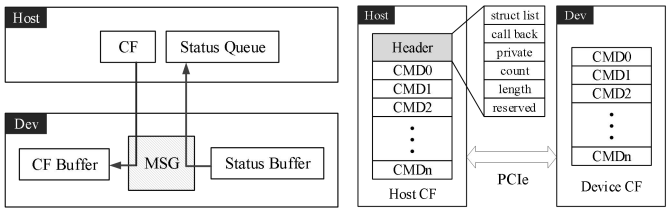


FIGURE 5. Host and device transmission interface （left）

FIGURE 6. CF transmission mode （right）

The CF host side contains a Header header and some commands, the header is for software use only and is not transmitted to the device side via PCIe even the physical address between the header and the command may be discontinuous. Therefore, there is no header at the device side. Commands can have different command ids, read, write, erase functions, etc. and can be of different sizes, with a default configuration of 512B at initialization. for Header, all CF headers have the same structure and therefore have the same size, 32 bytes or 64 bytes. Where struct list is a chain table for queue management call back is a function registered by FTL, specifically for FTL transmission; count is the total number of CF transmission; length is the total size of this CF, excluding the CF header, is a multiple of 8 bytes.

To transfer the CF to the device, the host must write the CFA (Command Frame Address) command frame address to a special PCIe register in the format shown in Figure 7. Each CFA is 8 bytes long, and the command frame address is the high 56-bit physical address in the CF. The lowest 8 bits in the CFA must all be zero and then be reused by the Length field. Therefore, the CF must be aligned with 256 bytes.

Byte0	Byte1	Byte2	Byte3	Byte4	Byte5	Byte6	Byte7
Length	Command Frame Address						

FIGURE 7. The CFA format

See the CFA FIFO in Figure 4, the Command Address FIFO, which is written by the Host through the global registers. Each command frame is written through Host and contains the host address and command frame length. When the CFA FIFO is not empty, the MSG unit sends a read request to the DMAC. The DMAC completes the request and writes the command frame directly to the CF Buffer. The CF Buffer is divided into two buffers of 512B each for ping-pong operations.

If an error occurs during a command frame transfer, the message unit sets the CF_ERR bit in the Control/Status register and stops after the transfer. Only a reset can re-enable the transmission, but all unprocessed CFAs will be lost.CMD Scatter pops an entry from an empty command slot FIFO of 64 levels of depth containing the slot index, calculates the address based on the slot index, and copies the command entry from the CF Buffer to the CMD Buffer.4 NAC Slot FIFOs with 64 levels of depth.MSG unit simply analyzes the command and pushes the completed command slot index to the corresponding NAC Command Slot FIFO. Hardware speeds up the MCU's command distribution efforts.

To send a status frame, the MCU writes the start address, frame length, and host status frame index into the SF Request FIFO, which is 16 levels deep and 15b in size. The MSG unit calculates the host address based on the FIFO content and sends a request to the DMAC, which reads the data directly from the status buffer and sends it to the Host.

After sending the frame, which also contains the update status, if an update is also enabled, the MSG unit pushes a status to the SF Completion FIFO, and the MCU knows the result of the request by reading the FIFO. This FIFO is also 16 levels deep. If there is an error in the status frame during transmission, the MSG unit pushes the error status into the SF Completion FIFO and stops any other requests.

C. Command dispatch and status return

The operating mechanism of the PCIe hardware link and message unit of the storage device is introduced above, as shown in Figure 8. Based on this operating mode of the FPGA logic mechanism, the on-chip processor needs to run two versions of firmware on the MCU and NAC , respectively, to further process the information transmitted by the FPGA logic. This section will elaborate on the way of command dispatching and optimization from the software side, i.e., how the MCU uses the above mechanism to realize the command response and internal scheduling, and finally, the process of returning the system processing status to the Host.

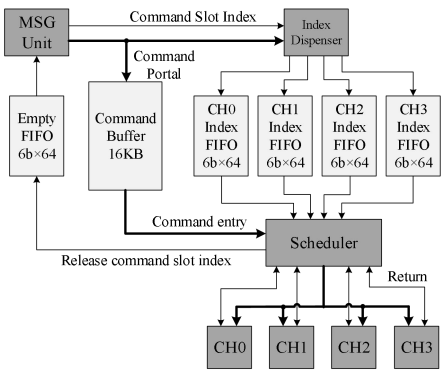


FIGURE 8. Command to distribute structure diagram

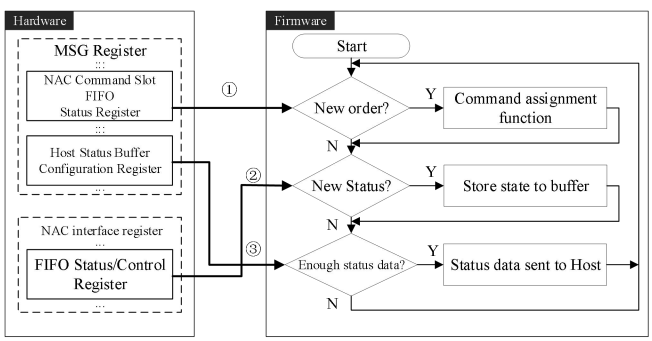


FIGURE 9. Schematic diagram of main loop flow

Figure 9 shows the flow of the MCU when processing commands, which contains three main steps.

1) Command check. The device first checks the hardware status. If there is a new command, it assigns the command to the appropriate NAC, depending on whether the host has a command to process.

2) Status check. The MCU checks if the status is returned from the NAC, and if so, stores the status in the buffer.

3)Send the status data to the host. If there is enough status data or a certain time is reached, it sends the status to the host through the DMA engine.

If no command is processed, the main loop function checks only the registers associated with the command dispatch, the status data from the NAC, and enough data bytes. It will check the registers nine times, spending a total of 112 cycles. Since there are 4 NACs, four registers associated with the status check are used. There is one register associated with the new command check. Another register is used to determine if there is enough status data.

An important parameter of SSD performance is latency. The main reason affecting MCU firmware latency is the time spent copying data to the command pool of NAC 0-3, so it is necessary to optimize the copying data process.

Since different commands have different formats and lengths, it is important to consider using loops to perform copy operations controlled by different length parameters. However, in this way, the loop control will consume CPU cycles and thus prolong the latency time. As shown in Figure 2-10, the cyclic lookup needs to consume more time to make a judgment about whether this HAM (Host Address Message) slot has the required data, and if so, then go for replication, which is R1 time. Moreover, the stack management operation is also required during the interval of each replication, i.e., the time from b to c.

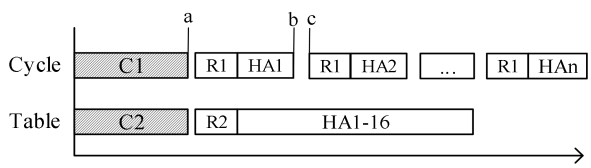


FIGURE 10.Loops and lookups indicate intent

The optimization results are shown in Figure 11. The assembly command operation has nearly ten times improvement, and the overall performance of the data moving operation has about 145% improvement. The declaration in the assembly function has three parameters: the first parameter is the source address stored in the R0 register; the second is the target address stored in the R1 register; and the third is the length of the data stored in the R2 register.

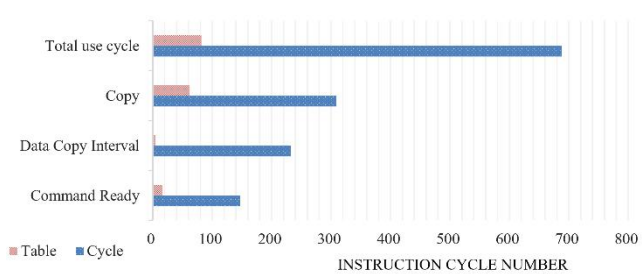


FIGURE 11.Total usage time test comparison

Based on the command format, each Host Address (HA) field requires 8 bytes of space. These HA entries are then copied to the SRAM section of the NAC (Network Access Controller) 0-3 command pool. The maximum number of HA entries that a command can hold is 14. Therefore, the maximum space for HA entries is 14x8=112 bytes. The maximum number of data bytes copied to the register portion of the NAC 0-3 command pool is 64 bytes. So the maximum number of data bytes to be copied by the firmware is 112 bytes. This is a small block of data to be copied. The assembly instruction processing flow is shown in Figure 12.

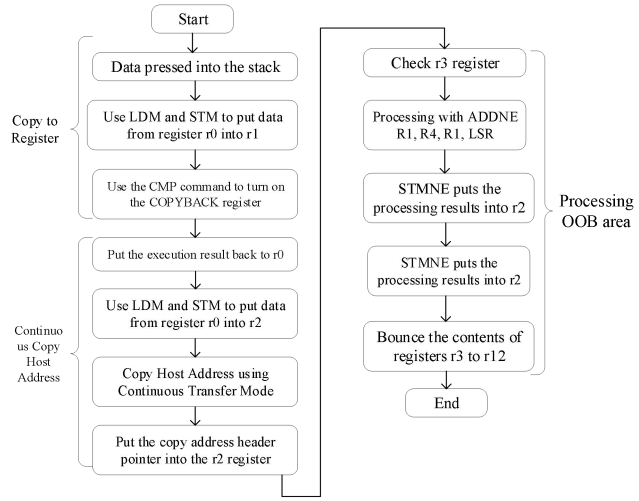


FIGURE 12.ASM process flow

There are 14 tags used in the firmware, each tag corresponding to 8 bytes. These 14 tags are arranged in a table, and the firmware looks up this table based on the length parameter input, thus improving latency performance. Compared to the cyclic control method, the lookup table method saves time for cyclic control but takes up more code space. Since the maximum data block is only 112 bytes, this is justified for latency performance reasons.

NAC0~3 will each enter the state of command processing after the command is taken out from the FIFO. If it is a time-consuming write-and-erase command, it will take up a long time, and at this time, the MCU just needs to continue to prepare the command according to the above process. When the NAC processing is completed, it will involve feeding the processing results to the Host, as shown in Figure 13.

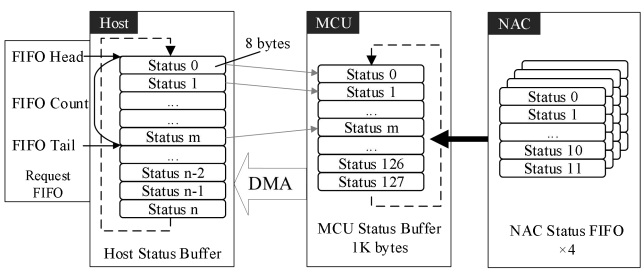


FIGURE 13.Status return process

Four NAC status FIFOs: Each NAC has a status FIFO containing 12 entries. The MCU and NAC exchange status entries through the status FIFO. When a command is completed, the NAC pushes a status entry into the FIFO, which can then be read by the MCU.

MCU Status Buffer: There is a 1KB status buffer in the MCU. 1KB status buffer can hold up to 128 status entries. It is a dual-port SRAM that can be written by the MCU or read by the DMAC simultaneously, cycled, and maintained by the MCU's firmware.

Host state buffer: There is a state buffer in the host memory, usually allocated as DRAM.

MCU firmware collects status items from the NAC status FIFO and stores them in the status buffer. Each operation to read the port registers pops up a status message called a status frame and puts it into the NAC status FIFO.

The firmware will prepare a status frame for transmission if the number m of status entries collected reaches the threshold set in the SFQ field of the Status Frame Configuration Register. A frame transmission will also be prepared if a timeout event occurs when not enough entries have been collected. However, when the status frame request FIFO is full, the actual number is greater than the SFQ if the status frame transmission is blocked and the firmware delays the transmission and requests again by merging subsequent status items.

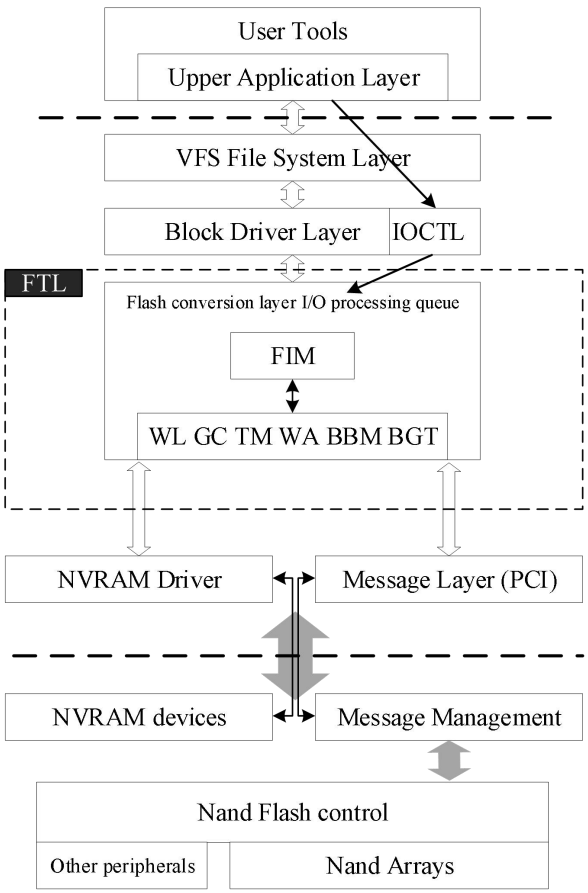
Therefore, a status frame is a variable-length structure. To manage the status frames, a request FIFO is used. Since the minimum length of a status frame is 8 bytes, the maximum number of status frames in a 1K buffer is 128. the request FIFO consists of a 128-byte array to store the frame length, header pointer, and length variables. Although the offset address of the status frame is also required for the status DMA transfer, it can be left out in the following steps. So the offset address is not stored in the FIFO. The MCU firmware can only use the length information of the status frame to ensure that both the MCU status buffer and the Host status buffer can avoid overflow. By controlling the FIFO header and counting information, the status of FIFO can be recycled.

III. HOST-FTL ALGORITHM DESIGN

The original purpose of FTL was to complete the conversion of logical addresses to actual physical addresses, allowing hosts to use solid-state disks as normal disks through the file system[13]. Considering the problems of the flash memory, such as erase first and write later, the different granularity of read and write erase, data retention, and limited lifetime, other features such as wear leveling, garbage collection, and power management have been added.

The purpose of Host-FTL is the same as Device-FTL. With the support on the hardware and firmware of the device described above, we can achieve the control of flash chips and flash channels through software, instead of just black box read and write operations.

Figure 14 illustrates the complete Host-FTL design stack for this system. Host-FTL makes the flash disk a standard high-performance disk drive for the operating system, maximizing the benefits and minimizing the drawbacks of flash memory. Applications can still use standard disk drive access methods to read and write to Flash without any modifications. The host block driver receives block I/O requests from the OS, packages them, and sends them to the FTL. After the FTL processes the request, it calls a callback function registered with the host block driver to complete the



request. The host block driver can be integrated into multiple platforms, including Linux and Windows[14].

FIGURE 14.Host-FTL software architecture

A. Address mapping mechanism

From the address mapping above, it is clear that the mapping can be divided into three types: page-level mapping, block-level mapping, and hybrid mapping[15], the latter two appearing originally to solve the problem of insufficient RAM capacity in solid-state storage devices. On the contrary, in this system, there is a large amount of RAM resources available, and using page-level mapping is exactly the appropriate TM solution[16].

A TM table exists in the host memory, as shown in Figure 15, which holds the correspondence between logical and physical addresses. The logical address area stores the index of addresses, and the physical address area holds the index of flash channels, block groups, and pages. We also store the logical sector number, ECC, and status information in the out-of-band space[17].

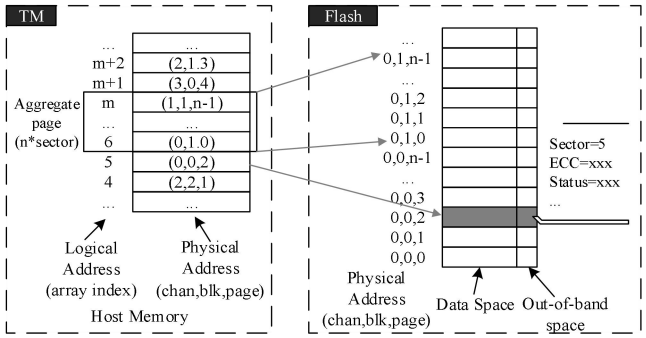


FIGURE 15.Page-level mapping structure

The granularity of the mapping is 4KB, which matches the usual sector size of the most popular file systems (such as EXT4 and NTFS). For a 1T storage capacity of this system, the RAM footprint is 1T/4K*4B = 1GB. However, the size of a flash page varies from 4KB to 16KB, and we offer the mode of aggregating multiple logical pages into a single physical page and then putting them together into pages with contiguous physical addresses, a job that requires a maximum of 4*N KB of cache. Aggregating and amplifying data blocks through software allows data to avoid fragmentation, effectively reducing the frequency of garbage collection and thus extending flash chip life. The aggregated pages require contiguous physical space addresses provided by a resource pool maintained by garbage collection, and since flash memory is erased in blocks, we can even aggregate the data into a block size before writing. In Device-FTL's common hybrid mapping, log blocks and cache management actually exist to aggregate more "hot pages," i.e., pages with more frequent data changes, leaving such hot pages in the responsive log pages and choosing the right time to put the corresponding "cold pages The corresponding "cold pages" will be put into the flash memory at the right time.

In this system, we also design PCP (Parallel Composite Page), i.e., parallel composite page, for eight chips of one flash channel. We aggregate the pages of all its Plane with the same address together so that the physical addresses of these pages keep the logical addresses consecutive and multiple PCPs can form a block group[18]. This is done because the flash chip can only program one Plane at the same time, and during this window period, the flash chip cannot respond to other programming instructions, so it is used to accelerate the write operation by programming all flash channels together.

When addressing groups, the Die index and page index is kept the same in all channels, and the Die group number is the same as the corresponding Die index in the channel. For example, Die group 0 of channel 0 consists of Die 0 of channel 0 on all channels, and in a given block group, page group 0 consists of page 0 of that block group. The only

difference in the block group address is the block index part; this is so that bad blocks can be easily decoupled from the block group and reduce the impact of bad blocks. This is also to match other flash chips for consideration so that when replacing another model of the flash chip with a different number of blocks and pages, it can also correspond. After grouping, the page group size is 4KB * 8 = 32KB, and the block group size is a multiple of the block size. Because the block indexes in the group may be different, a mapping table is necessary. This table is called BGT (Block Group Table). At first startup, the BGT is constructed to exclude any bad blocks in the block group, which is an array of 8 block indexes. If any block becomes bad after the operation, the whole block group will be marked as bad, and the block members will be separated. After that, a good block is found on this channel to replace it to maintain the block group balance. Equation (1) shows how the block group table is calculated, and in this system, the block group table size is 4096×8Die×4chan×2×8lane×2B=8MB

$$S_{BGT} = \left(N_{block\frac{groups}{die}} \times N_{die} \times N_{chan} \times N_{lane}\right) \times 2B \qquad (1)$$

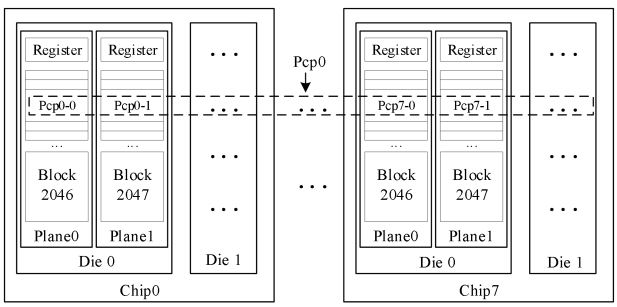


FIGURE 16.PCP combination mode

The BGT will be divided into dynamic and static, with active updated in memory in real-time and static will be stored in Flash. To ensure the consistency of the BGT, the last valid BGT address is always held in NVRAM to prevent the BGT from being corrupted by a power failure during the update.

PCP and block groups are only suitable for writing large files with cold data. The combination pattern of PCP is shown in Figure 16, where the number of Dies and Plane within the chip is subject to change with different flash chips.

B. Wear equalization algorithm

As mentioned earlier, wear-leveling algorithms are an important means of maintaining balanced consumption of flash block life within an SSD. The main metrics to evaluate the wear balancing performance are the standard deviation of erase count, maximum erase count, additional erase count, and hardware resource consumption. The standard deviation of the erase count is defined in equation (2).

$$D(X_t) = \sqrt{\frac{1}{n}\sum_{s=1}^{n}(X_t^s - \overline{X_t})^2} \qquad (2)$$

Formula (2), X_t^s that the sth storage block at the moment t erasure number, $\overline{X_t} = \frac{1}{n}\sum_{s=1}^{n} X_t^s$ for the average number of erasure of all storage blocks at the moment t, $D(X_t)$ represents the concentration of the number of erasure within the solid state disk. The maximum erase count, after the flash block erase count exceeds the expectation value given by the manufacturer, is considered a bad block and will be removed from the standard deviation calculation formula. The ratio of flash blocks exceeding the maximum erase count to the total flash blocks also reflects the overall lifetime of the SSD. The extra erase count is the number of erases to the flash block due to the SSD's spontaneous static wear equalization. The resource usage is also taken into account since performing wear-leveling takes up a lot of flash bandwidth and MCU resources.

Figure 17 illustrates the mechanism of the area wear equalization by ranking the block erasure counts from largest to smallest; reaching the maximum number of erasures is marked as a bad block, and the index address of the bad block is recorded to the bad block area and simultaneously removed from the writable addresses that can be assigned[19, 20]. T is an adjustable window threshold, and the difference in the number of flash block erasures in the solid state disk at the same moment is a maximum of T.

Copy-back is a command provided by the flash chip for copying data between different blocks within the same die. Compared to the normal data migration process, i.e., read, transfer, process, write, which takes about 1 ms, the Copy-back step only reads and then writes, which does not take up bus bandwidth and takes about 650μs and can improve performance by about 35%.

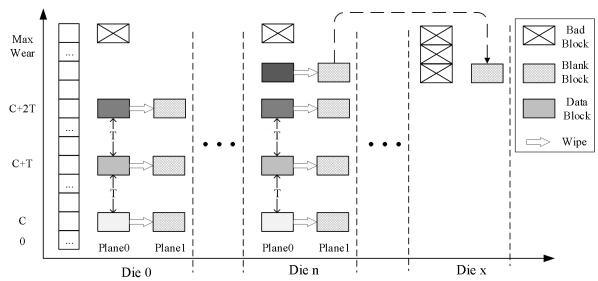


FIGURE 17.Wear leveling of area erasure times

In this system, due to the use of an FPGA controller, there is a special operation called intra-region copy because the eight flash chips connected together share the same SRAM region, so it is also possible to copy data between two Die in the same region without using bus bandwidth, taking about 850 μs. Taking advantage of Copy-back and intra-region copy, we wear out The equalization work is divided into regions on a channel-by-channel basis, and within each region, there are multiple forms (bi-directional chained tables): a blank block resource table, a garbage block table, a hot data block table, and a cold data block table. There is also a general form: the bad block table. If there are not enough available blocks in a region, blocks from other regions can be erased and added to this region, but this loses the advantage of Copy-back, which

only occurs when the solid-state disk is reaching the end of its life.

When a written request arrives, FTL needs to select the appropriate block to save the data. Firstly, the selection is made within the same region, and the block with the smallest current erase count is selected in the free block resource pool. Secondly, if the average erase count of the free blocks in this region has exceeded the average erase count of other regions by more than the threshold T, the free block in that region with the smallest average erase threshold will be selected for writing.

The main function of Regional Static Wear Leveling (RSWL) is to maintain the pool of free block resources and to migrate the cold data blocks in the region for recycling. After data has been written to flash memory for a long period of time without modification, dynamic wear balancing does not work anymore, and then RSWL is needed to migrate data by setting a variable value of erase count. Due to the different frequencies of hot data updates and cold data blocks in the internal partitions, RSWL is also divided into two types, i.e., intra-regional and inter-regional[21].

For the region, the first should be based on the region's cold and hot data form to determine the hot and cold data blocks; the most recently accessed logical address is saved and counted as a threshold window, and the current threshold window has the highest number of visits considered to be too much access to the hot data blocks. The block with the lower erase count in the cold data block table stores this data, and then the addresses of the two blocks are exchanged, and the hot and cold data block tables are updated[22].

For inter-region, we should use the same approach to make measure the wear and tear of the region. Set a percentage threshold D%, and when the $\bar{X}_t$ of a region is greater than the average of all regions P/E over D%, migrate some of the hot data blocks M in this region to the one with the smallest $\bar{X}_t$.

For sequential data writes to large files, we use the PCP page group policy so that the entire block of data is cold, making it easier for WL to proceed. The cold block table in the region is also able to ensure that cold data is written to certain flash blocks individually. Since WSWL generates additional writes and, thus, additional P/E cycles. This overhead must not be too high, or the benefit of wear balancing will be lost. It is recommended to keep the extra P/E cycles below 2‰.

In addition to the P/E cycle limit, flash memory has a read cycle problem that starts with the read interference effect of flash memory. As the data area is read by more and more people, the read interference will get worse. With MLC NAND, the general rule is to not read each block more than 100,000 times before updating the data; otherwise, it will result in uncorrectable data errors, which increases the likelihood of data loss. A straightforward solution to avoid excessive read cycles is to keep a read counter for each block, which increases with each completed read request. Once the counter reaches a predefined value, the data in the block is moved to another location, the previous block is erased, and the new block's read counter is reset to 0. However, it is not practical to write the read counter to flash memory because the overhead and performance impact is too great. So when power is lost, the read counter is lost as well. We can determine the severity of the read disturbance by the error correction return data from ECC. Even if the P/E cycle does not reach the threshold, if the ECC error correction bit count is too high, it means that this block is about to be damaged.

C. Data Protection

The Redundant Array of Independent Disks is an important means of data backup and protection, and we choose the RAID5 solution, which requires fewer disks. RAID5 does not back up data but stores parity check information from the same strip onto a certain strip[23, 24].

In this system, the eight logical channels are treated as eight storage areas using RAID5 striping. Implementing RAID can be done in software and hardware. Software processing of the data to calculate the checksum information will increase the load on the Host CPU and cause unpredictable latency. Therefore, we choose to add a bit XOR checksum generator in front of logical channel 7 in the DMA controller inside the FPGA, and logical channel seven stores the corresponding checksum information if RAID5 is on during data transfer. This hardware RAID scheme is implemented by the FPGA hardware and has no impact on the SSD speed. On write, when the destination address is a page buffer, and RAID is enabled, the incoming 128-bit data is reordered to 224-bit data + 32-bit RAID code. When reading, if RAID is disabled, all eight channels of data should be reordered to 128 bits wide and sent to host memory. If RAID is enabled, logical channel 7 for read data is the RAID checksum that should not be transmitted, and logical channels 0~6 data should be reordered to 128 bits wide.

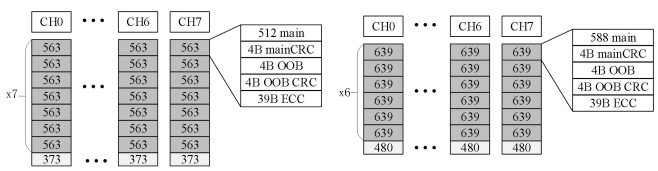


FIGURE 18.RAID organizational form

The RAID organization is shown in Figure 18. 8 logical channels are actually in one physical channel, corresponding to 8 flash chips and one NAC. One physical channel has a 4KB main area and 218B spare area; each page is divided into six sectors, and six pages are written at a time to make up one-page size data; the effective page group size is 6*4KB=24KB. In each sector, there is an ECC check code for this sector to ensure the correctness of data transmission.

IV. SYSTEM TESTING

A. Setting up the test environment

The SSD software system is divided into on-disk firmware and off-disk driver, which contains the necessary flash

conversion layer strategy and PCIe driver. A standard JTAG interface is reserved on the SSD board for debugging the FPGA chip, through which we can burn the firmware into the off-chip SRAM, and the FPGA can read the Bootloader at the corresponding address after powering up and then get the code segment needed by the MCU and NAC and load it to run. The driver is developed in a Linux environment and consists of several modules such as FTL, PCIe, data protection, and message queue. After the driver is loaded, a block device is generated under Linux /dev/block for read and write operations by the system. We can use a tool running at the user level for SSD management, mainly for formatting, temperature detection, status checking, etc. A more complete and stable CentOS distribution with kernel version 2.6.30 is chosen. Other distributions can be loaded normally, but make sure the kernel version is the same.

The solid-state disk card is linked to Host using a PCIe link. Host is a tower server that currently uses only one CPU and plugs the card into a standard port of PCIex16. The hardware architecture of this system is shown in Figure 19.

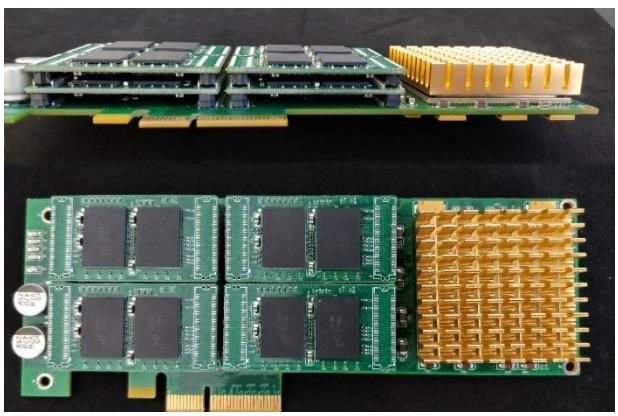


FIGURE 19.SSD card hardware

The test environment utilized for this SSD storage system includes a standard PCIex4 link connecting to the Host host. The complete test environment is illustrated in Figure 20, and the configuration details of the Host host can be found in Table 1.

TABLE 1
HOST TEST CONFIGURATION

Components	Parameters
CPU	Xeon E5-2420 v2 @2.8GHz
RAM	32GB ECC DDR3
Operating System	CentOS 6.5/Linux kernel 3.10
Host SSD	Intel 545s 512G
Tools	IOmeter
Test content	IOPS 、 Read/Write Bandwidth 、 Steady-state delay
Implementation Standards	SNIA

The performance of an SSD is influenced by multiple internal mechanisms, and solely emulating and testing an internal module is insufficient to fully capture the impact on SSD performance. iOmeter, originally developed by Intel for dedicated testing of storage disk and network I/O performance, allows you to set your own parameters, such as read/write data block size, queue depth, etc. In this paper, we choose the black-box test method to test the whole solid-state disk system as a whole.The test data encompasses metrics such as sequential read/write speed, random read/write speed, IOPS (Input/Output Operations Per Second), latency, and more. The value of IOPS can vary significantly based on the system configuration, including factors such as the read-to-write ratio, sequential versus random access ratio, configuration method, number of threads, access queue depth, and other parameters.

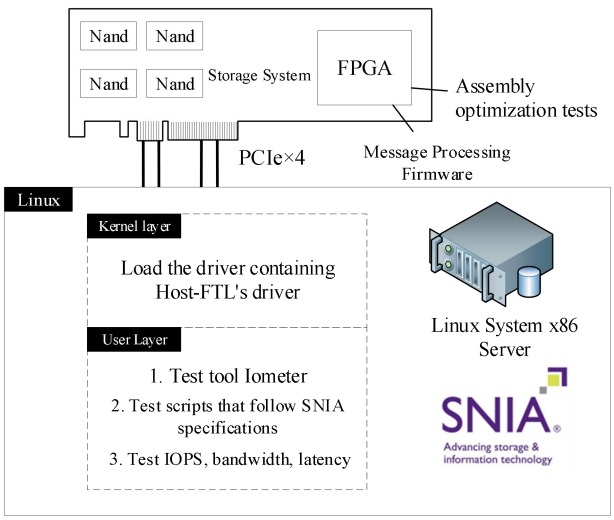


FIGURE 20.System test environment

The SNIA Solid State Storage Technical Working Group is a non-profit organization of more than 50 SSD industry companies, including SSD controller manufacturers, test houses, and vendors, that continuously research, test and measure the latest solid state products and technologies. SNIA's PTS (Performance Test Specification) is the industry's key standard for SSD testing.

The system is tested based on industry test standards to create test cases, where QD/Thread = 64/32/16/8/4/2/1 and Thread count = 4. The IOPS test case flow is shown in Figure 21. Where Purge does not simply format the drive but uses the way in the specification to eliminate the effects of other operations (reads and writes, other tests) prior to the test, leaving the SSD in a known blank state prior to each test. In the preparation condition, the test script reads the parameters and variables set for this test, including

queue depth, number of threads, write cache enablement in the system, etc.

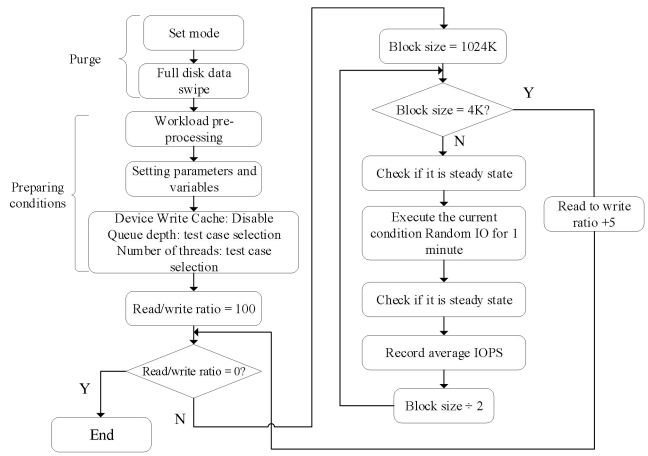


FIGURE 21.IOPS test case flow

In Figure 22, the IOPS values are depicted for various block sizes and five different ratios of read and write requests, representing different load cases. The measurements were taken with a queue depth of 16 and a thread count of 4. In all other cases, the IOPS remains stable within the range of 75 to 100K. For larger block sizes, the IOPS demonstrates an inverse relationship with the block size. This is because the system employs block group management, and the total bandwidth is not reduced as the total number of bytes in one operation increases, aligning with the expected behavior according to the system design.

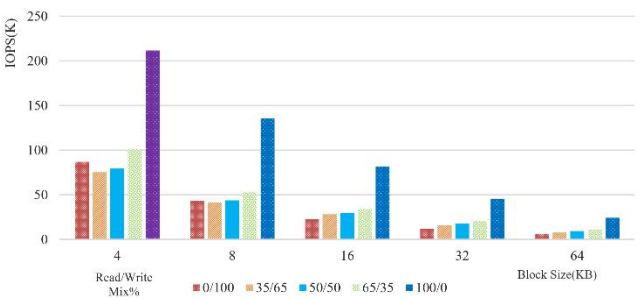


FIGURE 22.IOPS comparison of different block sizes

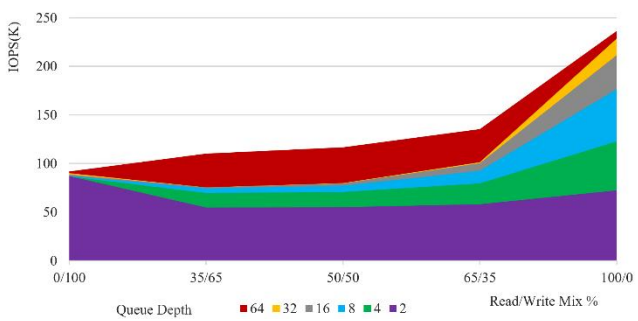


FIGURE 23.IOPS performance under different queue depths

The IOPS performance can vary at different queue depths. Figure 23 illustrates the IOPS performance across queue depths ranging from 2 to 64, focusing on a block size of 4KB and different read/write mixes. As the queue depth increases, the impact on IOPS improvement becomes less significant. However, in the case of all-read requests, higher queue depths can still lead to improvements. This is primarily because the execution speed of read requests surpasses that of write requests, allowing the queue to process and receive feedback at a faster rate.

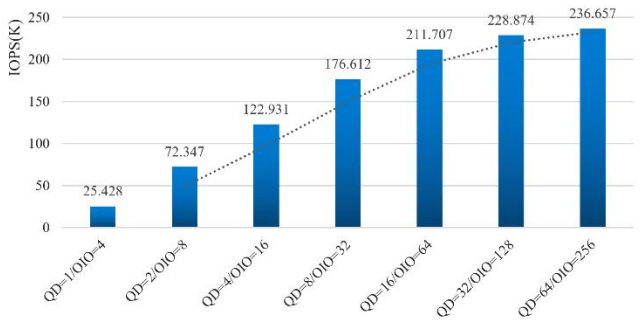


FIGURE 24.4KB writes to IOPS under different OIO conditions

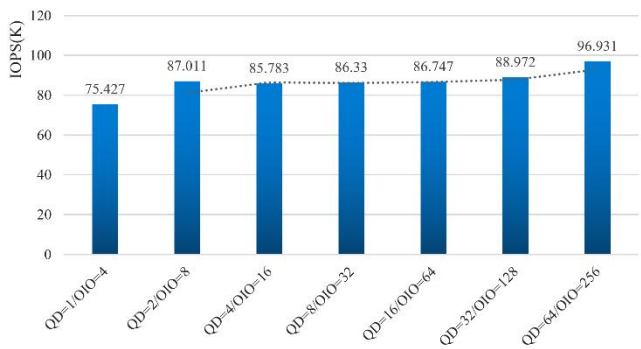


FIGURE 25.4KB read IOPS under different OIO conditions

For the case of OIO=8 to OIO=128, the 4KB write IOPS is not affected by the OIO parameter, all about 86000. the minimum IOPS for the case of OIO=4 is 75427, and the maximum IOPS for OIO=256 is 96931. the 4KB write IOPS is shown in Figure 24. And the 4KB read IOPS is affected by the OIO parameter. The minimum IOPS is 24529, and the maximum IOPS is 236657. 4KB read IOPS is below 100000 at lower OIO. 4KB read IOPS histogram is shown in Figure 25.

B. Read/Write Performance

By selecting different queue depths and block sizes, the sequential read and write performance can be influenced. In our testing, we specifically opted for a thread count of 4. Figure 26 demonstrates that, when using larger data block sizes and implementing the super page policy, the sequential write speed is minimally impacted by the queue depth. This is because the write speed is primarily determined by the programming speed, and the upper layer can more effectively fill the entire write channel of the solid-state disk. Only when both the queue depth and block size are small does the issue of write speed degradation

arise, although the degradation is not significant.

FIGURE 26.Comparison of continuous write speeds at different queue depths

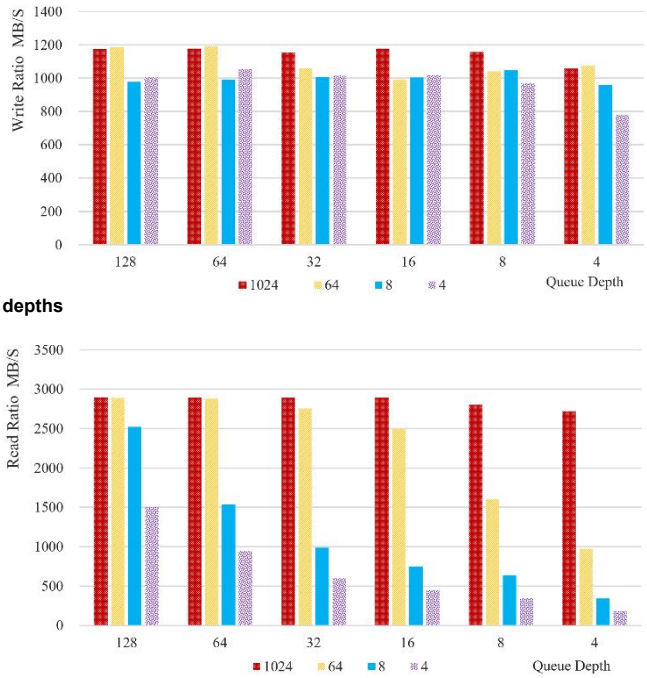


FIGURE 27.Comparison of continuous reading speeds at different queue depths

In Figure 27, the influence of various queue depths and block sizes on sequential reads is depicted. It is observed that with a sufficiently large block size, the sequential read rate remains at around 2800 MB/s, effectively utilizing the full bandwidth while accounting for space required for DMA operations and command operations, meeting the system design requirements. As the block size decreases, the read rate shows a gradual decline. For block sizes below 8KB and queue depths less than 32, there is a more significant drop in performance, as the read instructions are insufficient to establish sequential addressing at that point.

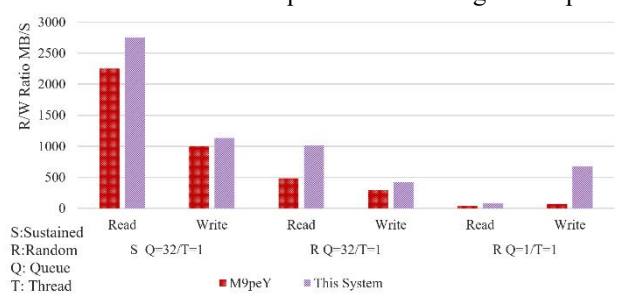


FIGURE 28.Read/Write Test of M9peY and this system

The literature [25] optimizes the number of threads and small I/O capacity on the prototype system based on OpenSSD, but its experimental system capacity and bandwidth number are too small, and the read/write bandwidth is low for comparison with this system. The other Host-FTL-based solid-state storage arrays are used by major data centers, and test data are not available. So the high-speed solid state disk with PCIe/NVMe interface on the Host host is chosen for comparison. In accordance with IOmeter's test method, 32-level deep queues and one thread were tested for read and write, and read and write were also divided into two forms: continuous and random. Since it is a normal device-level FTL SSD, the SSD is set to a block size of 4KB, which is consistent with it. In Figure 28, you can see that even at a logical block size, the performance of this system is still 20% to 100% higher. If provisioned to a larger block size, it would increase the storage array performance even higher.

C. Time delay

First, the latency within the SSD can be derived from the debug tool, and the number of command executions based on the clock cycles of the MCU is obtained. Figure 29 shows the number of MCU cycles for 4K read/write, PCP read/write, copy-back, no operation, and other (including erase, reset) commands under single-channel operation and 4-channel operation, respectively. As can be seen, under a 4-channel command operation, both read/write and other commands have a large improvement compared to single-channel, about 60%. The PCP read and write of the single-channel command is not much slower than the 4K read and write, so if the PCP write can be performed, the page group operation is given priority.

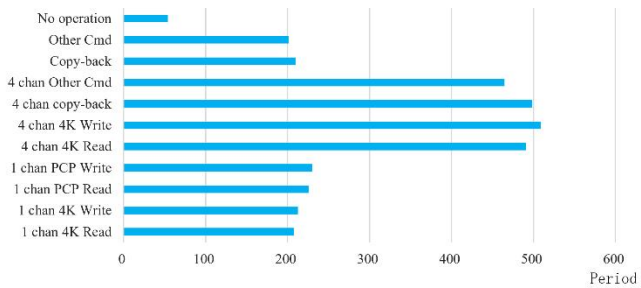


FIGURE 29.Delay contrast in solid state disk

The structure of the latency test case is basically similar to that of the IOPS test case. Under 4K conditions, a single-threaded random read/write method with a queue depth of 1 is used to measure the latency for block sizes of 16KB, 8KB, and 4KB to obtain the maximum and average values. The steady-state delay test of the system is shown in Figure 30. The curves in the figure show that after ten cycles, the system reaches a relatively steady state when the highest and lowest delays are in stability. Thanks to the optimization of FPGA hardware acceleration and instruction operation, the system does not have excessive fluctuations in delay time in the first ten cycles, and the whole system tends to be stable.

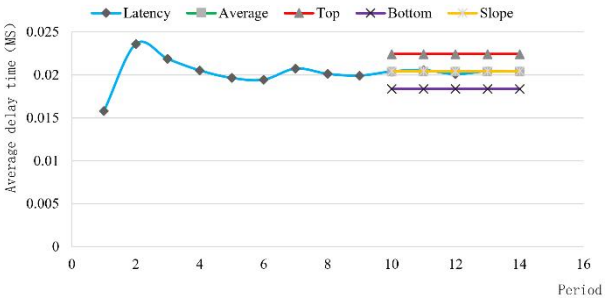


FIGURE 30.Latency Steady State Measurement

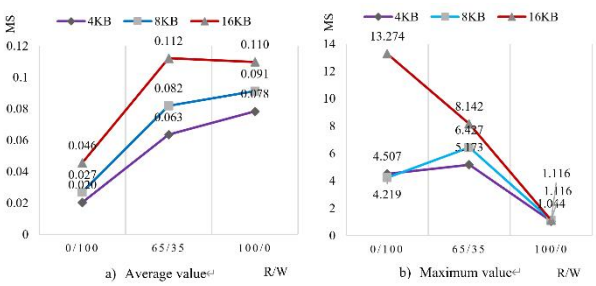


FIGURE 31.Average and maximum latency trends

Figure 31 shows the average latency and maximum latency of the system at 4K, 8K, 16K, and R/W at 0/100, 35/65, and 100/0, respectively. In terms of average value, the read latency for 4KB is about 78 μs, and the write latency for 4KB is about 20 μs. On the maximum value, the write latency at large granularity will be affected by various factors such as system process, host power consumption, etc. The maximum latency time is larger.

V. CONCLUSION

This paper completes the design of a high-performance, high-speed solid-state storage device controller, mainly the overall system architecture, instruction processing flow, and flash memory conversion layer design and implementation. The Host-FTL of this system addresses a wide variety of load requirements and changes the working mode of traditional solid-state disks from the bottom layer, striving to solve the contradiction between actual requirements and existing storage media. The system designed in this paper combines the flexible hardware acceleration capability of FPGA and the easy development flow of MCU and uses Host to manage flash channels directly, allowing the access time to be greatly reduced and making better use of the bandwidth resources at the bottom of the flash array. The work in this paper is mainly biased toward practical applications, and the conclusions obtained are as follows:

1) Completed the link establishment for PCIe, using a higher speed and secure interface to link the SSD to the Host, ensuring that the physical bandwidth will not be limited.

2) Designed and implemented an architecture for a hard disk controller with multiple processor cores interconnected and a multi-level cache and relied on hardware acceleration to design a message management mechanism and register management mechanism. It implements command transfer and status monitoring on the device side and host side and exposes all flash channel interfaces to the host side. The key data copying part is optimized at the assembly level, which effectively reduces the on-disk latency.

3) Implemented a set of Hots-FTL for direct flash management on the host side, using the "parallel combo page" and pipelining strategy for data and freeing up the bandwidth of the entire SSD by dividing logical channels for parallel pipelining. Through the area wear balancing strategy, the SSD life consumption is effectively reduced based on faster flash data operations in the same channel. It also uses channel-level RAID technology to improve the security of the data on the disk.

REFERENCES

[1] J. Kim, M. Jung, and J. Kim, "Decoupled SSD: Reducing Data Movement on NAND-Based Flash SSD," *IEEE Computer Architecture Letters*, Vol. 2, pp. 150-153, 2021.

[2] Y. Pan, Y. Li, H. Zhang, et al., "GFTL: Group-Level Mapping in Flash Translation Layer to Provide Efficient Address Translation for NAND Flash-Based SSDs," *IEEE Transactions on Consumer Electronics*, Vol. 3, pp. 242-250, 2020.

[3] Y. Yao, M. Yan, X. Kong, et al., "An Adaptive Read-Write Partitioning Flash Translation Layer Algorithm," *IEEE Access*, Vol. 7, pp. 179063-179073, 2019.

[4] Y. Jun, J. Park, J.-U. Kang, et al., "Analysis and Mitigation of Patterned Read Collisions in Flash SSDs," *IEEE Access*, Vol. 10, pp. 96997-97009, 2022.

[5] H. Qin, D. Feng, W. Tong, et al., "QBLKe: Host-side flash translation layer management for Open-Channel SSDs," *Journal of Systems Architecture*, Vol. 119, pp. 102233, 2021.

[6] S.J. Kwon, "Address Translation Layer for Byte-Addressable Non-Volatile Memory-Based Solid State Drives," *IEEE Access*, Vol. 7, pp. 73207-73214, 2019.

[7] P. Forouhar and F. Safaei, "Increasing the Lifetime of Flash Memory Based SSDs by Improving the Merge Operation in Flash Translation Layer," *IEEE Access*, Vol. 8, pp. 134324-134333, 2020.

[8] N. Toulgaridis, E. Bougioukou, M. Varsamou, et al., "Real-time emulation and analysis of multiple NAND flash channels in solid-state storage device," *Microprocessors and Microsystems*, Vol. 74, 2020.

[9] R. Micheloni, S. Aritome, and L. Crippa, "Array Architectures for 3-D NAND Flash Memories," *Proceedings of the IEEE*, Vol. 9, pp. 1634-1649, 2017.

[10] L. Rota, M. Caselle, S. Chilingaryan, et al., "A PCIe DMA Architecture for Multi-Gigabyte Per Second Data Transmission," *IEEE Transactions on Nuclear Science*, Vol. 3, pp. 972-976, 2015.

[11] Y. Ou, N. Xiao, F. Liu, et al., "Gemini: A Novel Hardware and Software Implementation of High-performance PCIe SSD," *International Journal of Parallel Programming*, Vol. 4, pp. 923-945, 2017.

[12] X. Liao, Y. Lu, Z. Yang, et al., "Efficient Crash Consistency for NVMe over PCIe and RDMA," *ACM Transactions on Storage*, Vol. 1, pp. 1-35, 2023.

[13] X. Cui, L. Shi, and K. Wu, "Towards trustworthy storage using SSDs with proprietary FTL," *Microprocessors and Microsystems*, Vol. 55, pp. 82-90, 2017.

[14] E. Senn, J. Boukhobza, and P. Olivier, *Flashmon V2: Monitoring Raw NAND Flash Memory I/O Requests on Embedded Linux*. 2013.

[15] Q. Wu, Y. Zhou, F. Wu, et al., "Understanding and Exploiting the Full Potential of SSD Address Remapping," *IEEE Transactions on Computer-Aided Design of Integrated Circuits*

and Systems, Vol. 11, pp. 5112-5125, 2022.

[16] D.B. Yeo, J.Y. Paik, and T.S. Chung, "Hierarchical Request-Size-Aware Flash Translation Layer Based on Page-Level Mapping," *Journal of circuits, systems and computers*, Vol. 7, pp. 1950117.1-1950117.22, 2019.

[17] K. Han and D. Shin, "Remap-Based Inter-Partition Copy for Arrayed Solid-State Drives," *IEEE Transactions on Computers*, Vol. 7, pp. 1640-1654, 2022.

[18] V. Taranalli, H. Uchikawa, and P.H. Siegel, "Channel Models for Multi-Level Cell Flash Memories Based on Empirical Error Analysis," *IEEE Transactions on Communications*, Vol. 8, pp. 3169-3181, 2016.

[19] H. Zheng, H. Zhang, S. Xu, et al., "Adaptive Mode Transformation for Wear Leveling in Nonvolatile FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 11, pp. 3591-3601, 2022.

[20] L. Jiang, Y. Du, B. Zhao, et al., "Hardware-Assisted Cooperative Integration of Wear-Leveling and Salvaging for Phase Change Memory," *ACM Transactions on Architecture and Code Optimization*, Vol. 2, pp. 1-25, 2013.

[21] S.J. Kwon and T.-S. Chung, "Hot-LSNs distributing wear-leveling algorithm for flash memory," *ACM Trans. Embed. Comput. Syst.*, Vol. 1s, pp. Article 62, 2013.

[22] B. Zhou, S. Wan, and C. Xie, "Isolation: Inexpensively separating cold data via garbage collection to improve the lifetime and performance of NAND flash SSDs," *Concurrency and Computation: Practice and Experience*, Vol. 15, pp. e5460, 2021.

[23] A. Thomasian and M. Jai, "RAID5 performance with distributed sparing," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, pp. 640-657, 1997.

[24] C. Qiong, Z. Jiang-ling, and F. Dan, "One method for improving RAID5 performance," *Wuhan University Journal of Natural Sciences*, Vol. 3, pp. 289-292, 2000.

[25] M. Bjrling, J. Gonzalez, and P. Bonnet, "LightNVM: The Linux Open-Channel SSD Subsystem," Vol. 1, 2017.

BOYANG DING was born in Harbin City, Heilongjiang Province, China in 1997. He is currently pursuing a master's degree in Electronics and Communication at Heilongjiang University in Harbin, China.
His research interests include embedded systems and file systems.

SONGYAN LIU was born in Harbin City, Heilongjiang Province, China in 1969. He received his M.S. and Ph.D. degrees in computer organization and system architecture from Harbin Institute of Technology, Harbin, China in 2000.

He is currently an associate professor in the School of Electronic Engineering, Heilongjiang University, Harbin, China. His research interests include embedded Systems, SoC, and file system.

TAO LV born in Linfen City, Shanxi Province, China in 1996. He is currently pursuing a master's degree in Electronics and Communication at Heilongjiang University in Harbin, China.
His research interests include embedded systems and file systems.