Linux® Storage System Analysis for e.MMC With Command Queuing

Linux is a widely used embedded OS that also manages block devices such as e.MMC, UFS and SSD. Traditionally, advanced embedded systems have focused on CPU and memory speeds, which outpaced improvements in storage speed. This article describes at a high level how Linux deals with I/O requests from the user space and explores the parameters that impact access performance. We'll particularly pay attention to system performance limitations in the Linux® e.MMC subsystem and bottlenecks caused by the operating system rather than device speed.

Linux Storage I/O Request Flow

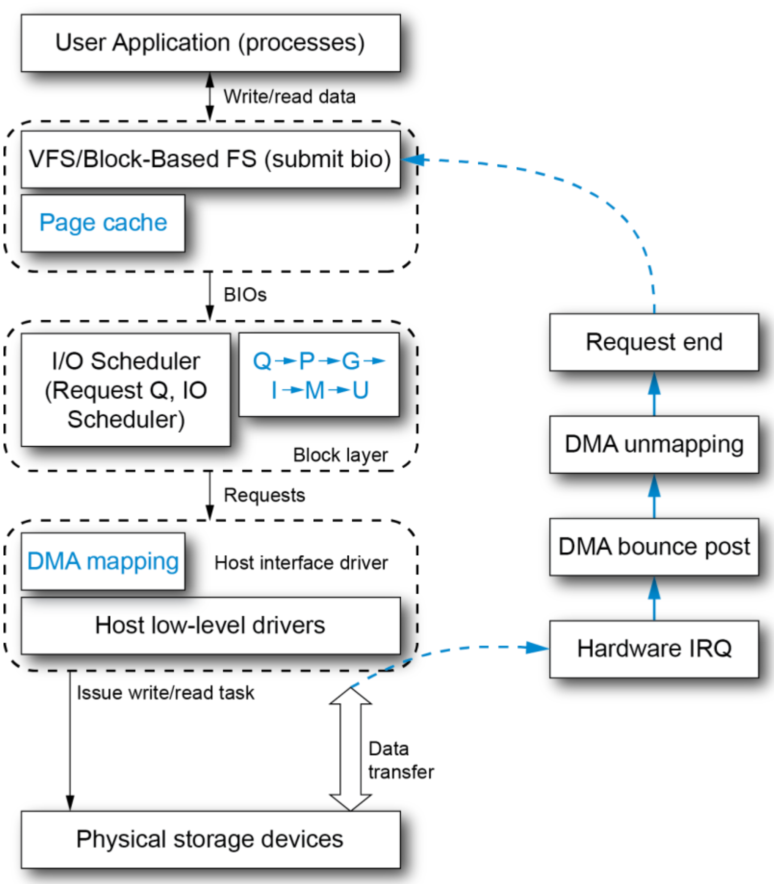

Figure 1: I/O Request Flow

Figure 1 depicts the Linux storage I/O request flow for a single user-space request on the way to the storage device.

The first layers of the flow are the virtual and block-based file systems, VFS and FS, respectively. During these stages, a data buffer is allocated for write requests and a MEMORY COPY operation is executed from the user space to the kernel space, which takes processing time. For a READ operation, a cache lookup is executed first to check whether the cache already contains the required data. If data is in the buffer cache, a hit condition, then the data is returned from the buffer cache, and the READ command will complete and return. For a cache miss condition, the data associated with the read request must be retrieved from the storage device, and the read request must traverse the full I/O stack. A data buffer is requested, which will be used to buffer the data returned from the storage device. Because no memory copy is executed, the initial stages for READ commands will consume less time than those for WRITE commands.

Allocated data buffer size depends on data length. For example, if 128KB data is programmed into the e.MMC device, the system will apply for 32 pages because the minimum page size is currently 4KB in the Linux memory management system. One page is requested at every COPY operation, and if a page is available, 4K is copied. After allocating a buffer the FS maintains the buffer information as a buffer header, this header is later passed to the page cache. The buffer header is formatted into the block I/O (bio) structure; one BIO structure represents an in-flight block I/O operation. The buffer header is then passed to the block layer by the block-based FS calling submit_bio().

After the FS calls the submit_bio(), the process moves out of the complex VFS and FS layers and into the slightly less complex block layer. The block layer checks the validity of the BIO, and if valid, the BIO will be turned into a new request using generpric_make_request(), or if the sector address requested in BIO is contiguous with a previously existing request sector, this BIO will be merged into an existing request based on the adjacent sector. The block layer then passes the request(s) to the I/O scheduler.

The I/O scheduler can re-order the requests to optimize storage operation execution time. There are several I/O scheduler designs and selection of a suitable I/O scheduler can make a difference in storage performance. For NAND-based block devices, such as e.MMC and UFS, NOOP is the recommended I/O scheduler.[3] Before dispatching I/O requests to the lower block driver layer, all I/O requests are "plugged," which means requests are not immediately dispatched to the low device driver, but are held by the "plugging mechanism." This enables additional requests to arrive so that large sequential requests can be built by a MERGE operation. This maximizes I/O throughput but may delay a single I/O request and increase the inter-request latency (defined as the time between request completion to a new request arrival). Requests will wait in this plugged list until either the list is full or a process becomes blocked by waiting on an I/O request. Either of these two conditions will trigger the I/O requests to be unplugged and dispatched.[1] As shown in Figure1, the request will go through these stages: Q (enqueue request) to P (plug request) to G (get request) to I (insert request) to M (merge) to U (unplug request). The block layer then forwards the I/O requests to the block host interface driver, such as the one in the MMC submodule layer for example.

The block host interface driver, which is related to a specific storage hardware interface, translates one block I/O request into a corresponding storage device request and executes direct memory access (DMA) mapping through arm_dma_map_sg(). DMA mapping creates the software structures used in controlling DMA operations. The host interface driver prepares DMA descriptors and triggers a hardware transfer. The real data transfer on the data bus is managed by the DMA engine. I/O requests will go directly into a lower-level driver, which will perform the relevant READ/WRITE operations. After the data transfer between the DMA engine and the storage device completes, an interrupt occurs that triggers a software interrupt handling function, and DMA unmapping arm_dma_unmap_sg() and request post-processing is then executed. A READ operation is expected to take longer than a WRITE operation because an additional step, a memory copy, from the kernel space to the user space is executed.

e.MMC Command Queuing Workflow

The command queuing (CQ) feature was introduced in JEDEC's e.MMC standard v5.1. Command queuing enables commands within e.MMC to be optimized and sorted.[2] Compared to the original single-threaded command execution, access efficiency can be increased and random access performance improved. CQ enables the host to dispatch multiple commands to the e.MMC device, permitting the device to increase parallelism and optimize command processing, thus achieving a higher overall performance.

A command queue engine (CQE) can be added to support the CQ feature. The CQE executes the functions related to CQ, including processing task information issued by software, communicating with the device using the bus protocol for issuing tasks and ordering task execution, copying data to or from the system memory, and generating interrupts. These functions can be managed in the hardware CQE or by software.

Figure 2 shows an I/O request flow for an e.MMC system with CQE. The VFS, FS and block layer are the same as in the general flow shown in Figure 1. The differences are in the MMC sublayer and the CQE driver. The MMC sublayer has one kernel thread that oversees fetching new I/O requests from the upper block layer request queue and dispatching them to the CQE driver. The e.MMC host driver performs bounce buffer copy and DMA mapping before passing the I/O requests to the CQE driver.

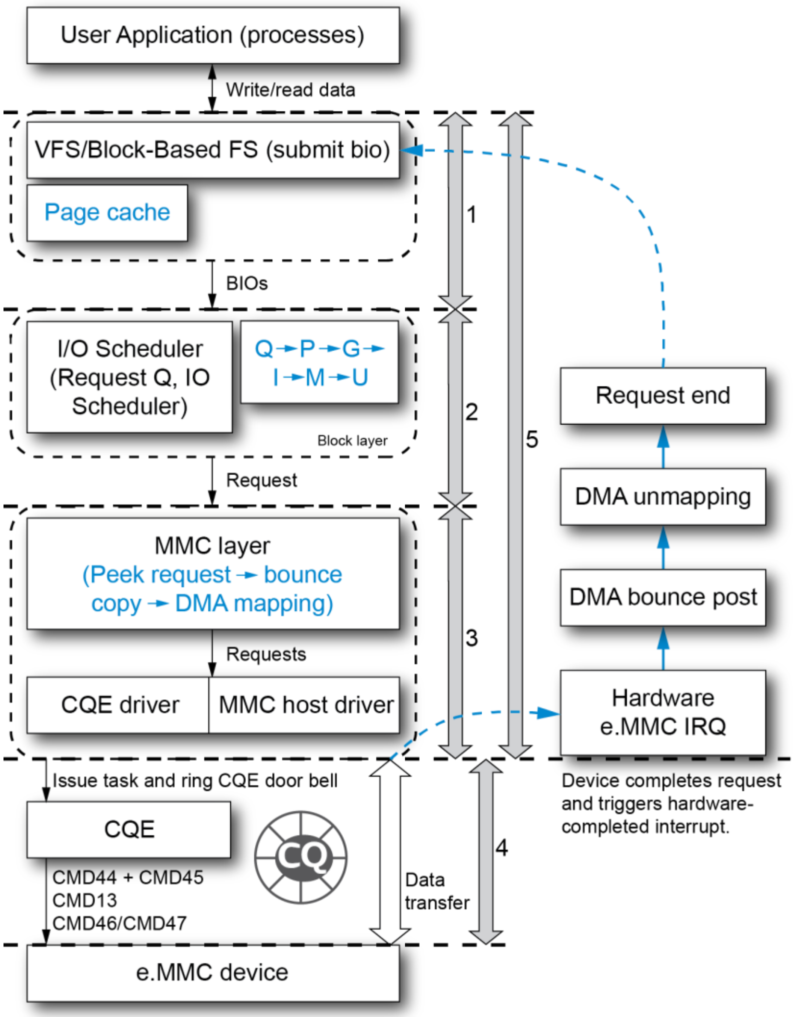

Figure 2: I/O Request Flow With e.MMC CQE

1. Preparing page cache or memory mapping in VFS/FS layer
2. BIOs plug/unplug and I/O schedulers in block layer
3. Request fetching, issuing and DMA mapping in MMC sub-layer
4. Hardware data transfer
5. Request-post

At this point, the CQE driver and CQE start executing (shown in Figure 3). The CQE driver prepares the task descriptor required by the CQ and triggers the hardware CQE to add the I/O request task to the e.MMC queue, after which the CQE continues to track and check the status of the I/O request tasks until the queue is empty. During CQE status checks, the CQE can continue accepting new I/O request tasks and adding them to the e.MMC queue if the queue is not full. When a task is ready to execute, the CQE starts the data transfer phase. After a data transfer is completed, a data transfer finished hardware interrupt is triggered by CQE, and the software interrupter handler starts to handle the request-post.

e.MMC With CQ Performance Evaluation

As shown in Figure 2, each I/O request will go through five spans, and each span will introduce overhead and impact I/O performance. We used a standard Linux tool, Ftrace, to measure the impact of each layer. We measured several test cases corresponding to various workloads and each test case was run at least five times. The reported values were then calculated by averaging the results of the five runs. The observed deviation from the mean was roughly 2%.

Ftrace is an internal tracer tool in Linux that can trace event, function and latency. The trace log stays in the kernel ring buffer.[4] To remain as close as possible to real data. Only several key functions and set tracepoints were chosen to measure span durations.

All performance data in this paper is based on the two platforms detailed in Table 1.

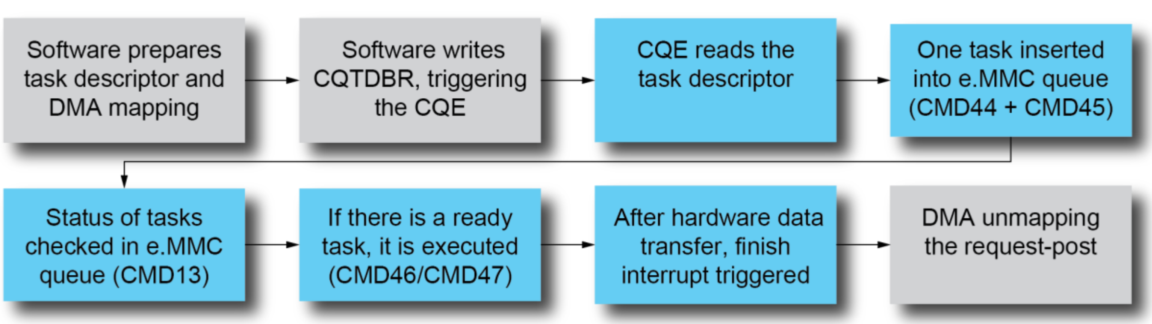

Figure 3: CQ Command Sequence

Parameter	Xilinx Zynq Zed Board	NVIDIA Jetson TX1 Board
CPU type	ARM Cortex A9	ARM Cortex-A57 MPcore
Core number	2	4
L1 cache	32KB L1 I-cache and 32KB L1 D-caches with parity per core	48KB L1 I-cache per core; 32KB L1 D-cache per core
L2 cache	512KB (unified cache)	2MB (unified cache)
CPU frequency	667 MHz	1.73 GHz
DDR type	DDR3 (32-bit)	LPDDR4 (16-bit)
DDR frequency	533 MHz	1600 MHz
OS	Linux 4.1	Linux 3.19
CQE	Hardware CQ engine	Software simulation CQ
e.MMC	32GB@HS400	8GB@HS400
Block FS	Ext4	Ext4

Table 1: Specification of Target Platforms

An I/O request latency is broken down into five spans (see Figure 2). To simplify the data collection and analysis, the second and third spans were combined into one, thus four spans were used to evaluate system latencies instead of five (see Figures 4 and 5). The first span included the VFS and ext4 operation latencies (memory allocation for BIO and memory copying [write] to submit the BIO). The second span included the block layer I/O scheduler latency, MMC subsystem request fetching and issuing to CQE latency. The third span was storage hardware latency, which was measured from when one request was enqueued in the CQE to when the corresponding data transfer completion interruption was received. It is a purely a hardware latency and depends on the storage device. The fourth span, called post-duration, included interrupt handling, DMA unmapping, memory copy (read) and request ending latencies. Figure 4 depicts the latency bars for the NVIDIA® Jetson™ TX1 platform with eight storage-stack configurations, and Figure 5 depicts the latency bars for the Xilinx® Zynq® Zed platform with eight storage-stack configurations.

Because the two hardware platforms we tested differ in their corresponding DMA engine designs and bounce buffer size, different data chunk sizes were used. The Jetson TX1 e.MMC controller uses an advanced DMA (ADMA) engine with scatter lists that have a maximum capacity of 128KB. The Zynq Zed board uses single operational DMA (SDMA) that can only support a single scatter list with a maximum buffer size of 64KB. In addition to the DMA MAPPING operation, the SDMA engine requires a BOUNCE BUFFER COPY operation prior to issuing a request to the CQE. This operation happens in the Block_MMC stage. As shown in Figures 4 and 5, the Block_MMC span takes longer on the 64GB Zynq Zed board than it does on the 128KB Jetson TX1 platform. This is related to their respective DMA, CPU and DDR capabilities.

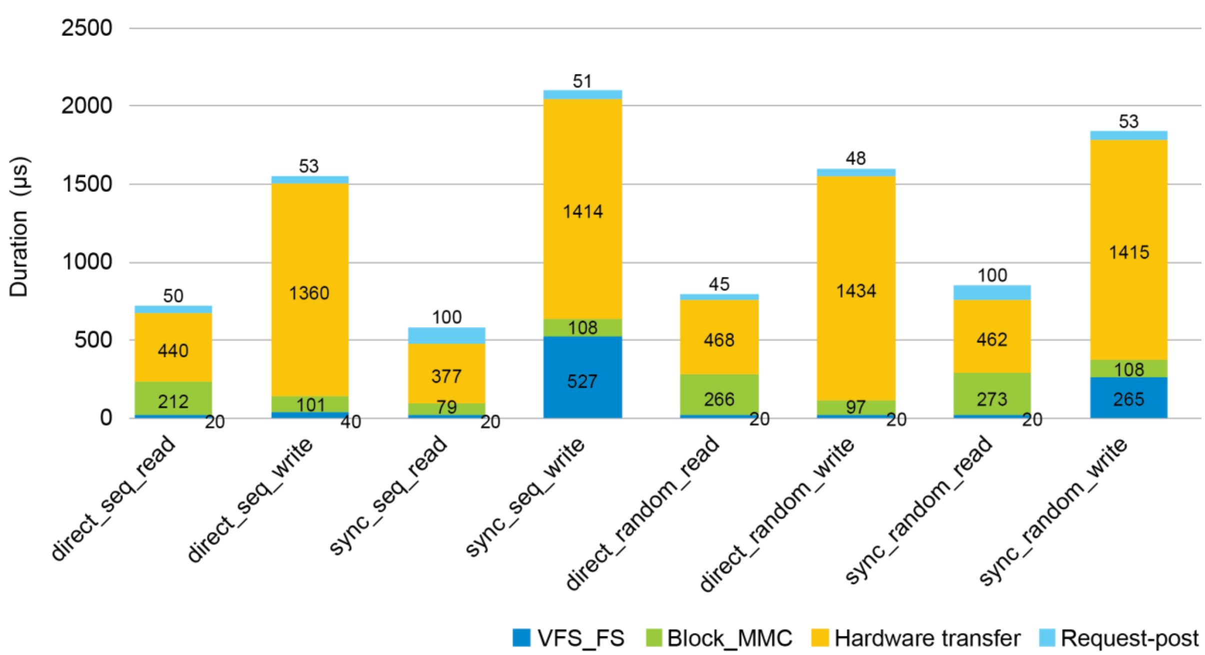

Figure 4: 128KB Data I/O Request Latency, NVIDIA Jetson TX1

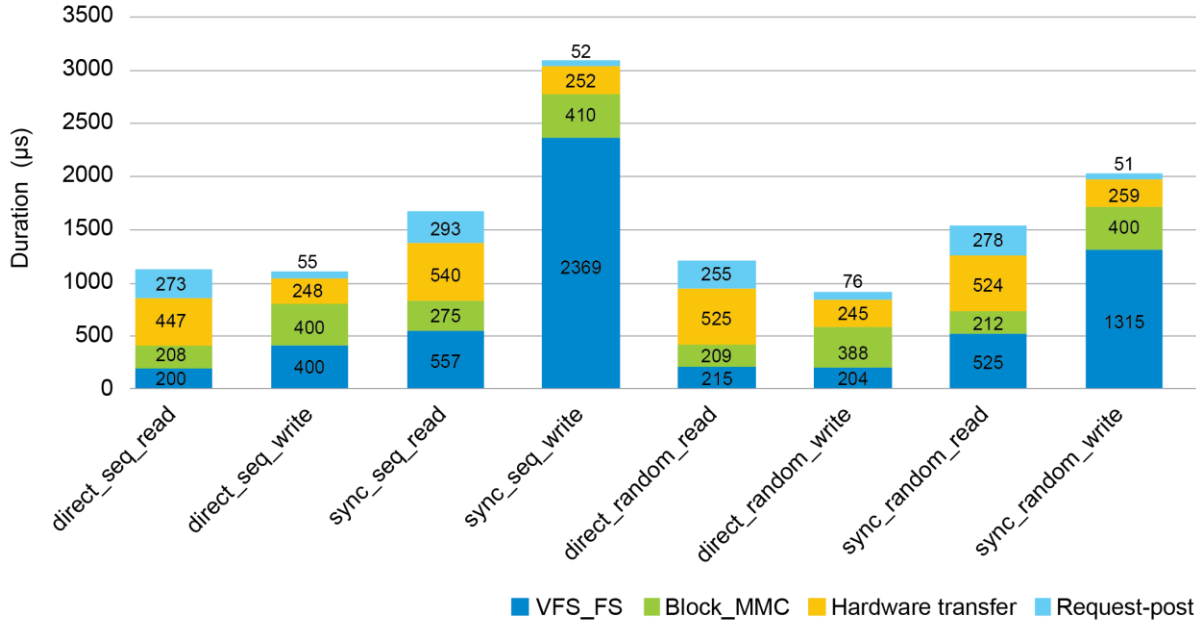

Figure 5: 64KB Data I/O Request Latency, Xilinx Zynq Zed

Another important factor regarding data size is the different CQE used on the two platforms. To better understand system-level limitations with small chunk size, refer to the latency for 4KB chunk size with random accesses in Figures 6 and 7.

Software System Overhead for the of NVIDIA Jetson TX1 Platform

- 128KB chunk size (Figure 4):
  23%~32% for sync writes
  10~15% for direct writes
  34~44% for reads in sync mode
  ~40% for reads in direct mode

- 4KB chunk size (Figure 6):
  ~80% for sync writes
  ~45% for direct writes
  ~40% for reads in both direct and sync modes

The data for the Jetson TX1 platform indicates that at the 128KB chunk size device performance is the dominant factor in system-level performance; however, system overhead is not insignificant at 10%–35%. At the 4KB chunk size system performance is limited not by the device but by system overhead, which takes up to 80% of the total time consumed on an I/O operation.

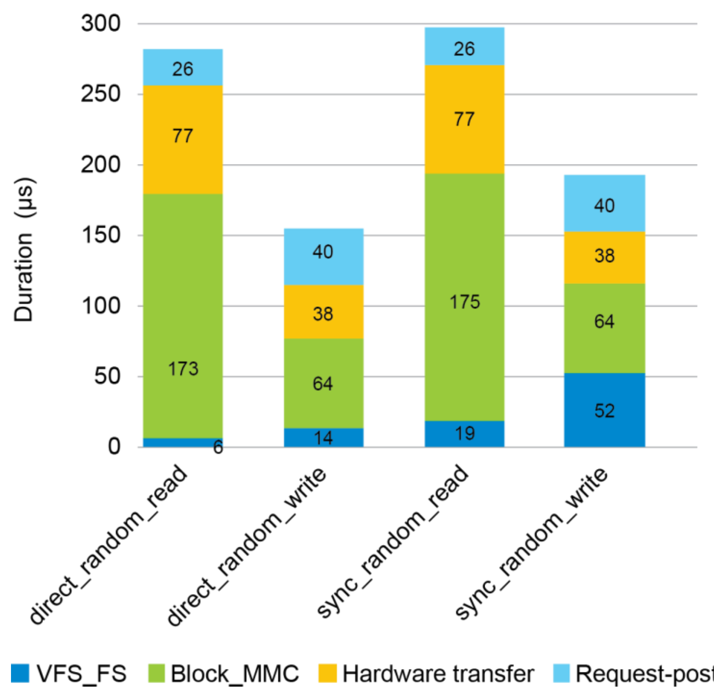

Figure 6: 4KB Data I/O Request Latency, NVIDIA Jetson TX1

Software System Overhead for the of Xilinx Zynq Zed Platform

- 64KB chunk size (Figure 5):
  87%~91% for sync writes
  ~75% for direct writes
  ~65% for reads in sync mode
  56%~60% for reads in direct mode
- 4KB chunk size (Figure 7):
  87%~91% for writes
  ~64% for reads

Unlike the Jetson TX1 board, system overhead is the dominant factor for the Zynq Zed board even at the smaller 64KB chunk size. This contributes to the cache management differences and the processor and DRAM speed differences.

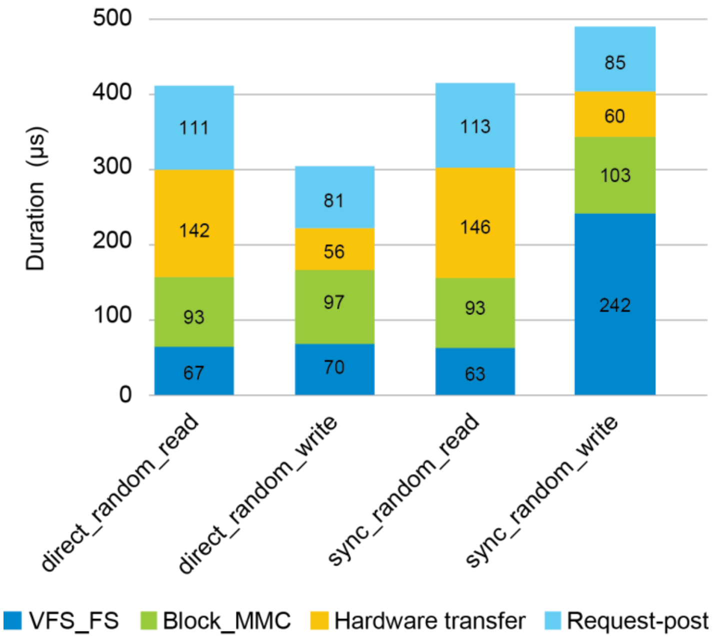

Figure 7: 4KB Data I/O Request Latency, Xilinx Zynq Zed

A comparison between sync and direct modes shows that on both platforms, regardless of READ or WRITE commands, VFS/FS always takes more time in sync mode than in direct I/O mode because direct I/O always bypasses the page cache, and the I/O operations pass directly to the underlying block layer. For a single I/O request in direct mode (as shown in Figures 1 and 2), the flow does not involve page cache preparation or memory copy. (Linux uses page cache to increase storage access.) This will largely reduce the first span's overhead. For READ commands, the VFS layer searches the requested data in page cache, if the data is found, the read request will not be passed down to the underlying block layer. Page cache can also be used for buffering data from read-ahead, in which adjacent pages will be read before they are requested.[5] This improves read performance but only for sequential accesses. For WRITE commands, page cache is used for write-back, which means that data is first copied to page cache, marked as dirty and then the write returns. The dirty data will be committed to disk later. This flow clearly increases write performance for user-space applications, but only exists in async I/O mode. Also, power loss will cause the buffered data to be lost. The current page cache mechanism was designed for conventional disk-based storage devices. For high-speed, flash-based block storage devices, it creates an overhead that limits the I/O access speed.

Figures 4 and 5 also show that a sequential write takes more time than a random write during the VFS/FS stage.

Theoretical System Performance Limitation

The five transfer spans of an I/O request discussed in the previous section can be simplified further into three simple spans. An I/O request can be considered a three-stage pipeline: request-pre, hardware transfer and request-post. The request-pre duration is the time the system takes to prepare one I/O request from the VFS layer to when it is queued to the CQE. The hardware transfer duration is the time it takes for the storage device to complete a command. The request-post duration is the time consumed from when the storage device has completed the I/O operation until the I/O is fully completed at the VFS level. This model allows us to consider the effects of the hardware duration on the system-level performance.

Jetson TX1 Read and Write Maximum Speed

The results discussed in the previous section were used to estimate the maximum system-level performance, assuming infinitely fast storage devices, and compare those speeds to the measured system-level speeds. The following table shows the estimates for four workloads: 128KB sync sequential writes, 128KB direct sequential writes, 128KB sync sequential read and 128KB direct sequential read. The table contains the theoretical maximum speed, which was calculated by assuming an infinitely fast storage devices with a zero hardware-transfer time, and the measured speed, which displays the actual user-level storage speeds. These numbers are useful in understanding how the actual system speed compares to the theoretical maximum speed.

Workload	Request-Pre (µs)	Hardware Transfer (µs)	Request-Post (µs)	Theoretical Speed (MB/s)	Theoretical Maximum Speed (MB/s)	Measured Speed (MB/s)
128KB sync sequential write	635	1414	51	60	182	36
128KB direct sequential write	141	1360	53	81	644	71
128KB sync sequential read	99	377	100	217	628	197
128KB direct sequential read	232	440	50	173	443	175

Table 2: 128KB Sequential Experimental Data, NVIDIA Jetson TX1

Figure 8 shows that direct I/O mode write has a significantly better performance than sync mode for the reasons discussed previously.

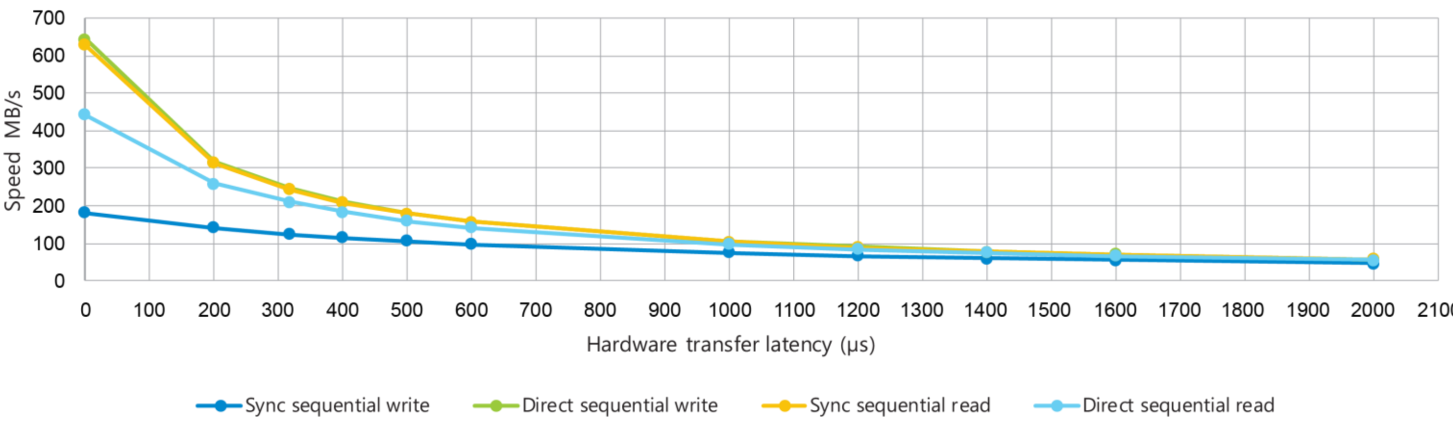

Figure 8: 128KB Sequential Experimental Data, NVIDIA Jetson Tx1

The measured speed for the 128KB chunk size sync sequential write is 36 KB/s with a hardware transfer time of 1414µs. If this hardware transfer time is decreased to zero, the maximum achievable speed will be around 200 MB/s due to system overhead. When the hardware transfer time is decreased to 700µs, the system speed remains at 90 MB/s — by cutting the hardware transfer in half there is an increase of 50%. In direct mode, the maximum achievable write speed with zero hardware transfer time is 644 MB/s, thus the overhead due to sync mode divides the achievable maximum write speed by a factor of three. The sync sequential read achieves 197 MB/s with a hardware duration of 377µs. Assuming a hardware latency of zero, the achievable speed is 628 MB/s, and the hardware latency results in a system speed loss of 431 MB/s. Direct sequential read is measured at 175 MB/s, and without the hardware latency the achievable speed is 443 MB/s — the loss is less than in sync mode.

This analysis confirms that improving device performance can significantly improve system-level performance for large chunk sizes. However, system-level write performance is capped at 200 MB/s for sequential sync writes, at 644 MB/s for direct writes, 442 MB/s for direct sequential reads and 628 MB/s for sync sequential reads.

Zynq Zed Multiple Thread Maximum Speed

CQ/One I/O Request Three-Stage Pipeline

An example of an ideal three-stage pipeline, where all spans take the same amount of time and the three stages can be fully parallelized, is shown in Figure 9.

With a depth of 32 commands, the total duration of processing N requests will take:
(N + 2) × D, where D is the duration of one transfer span.[1]

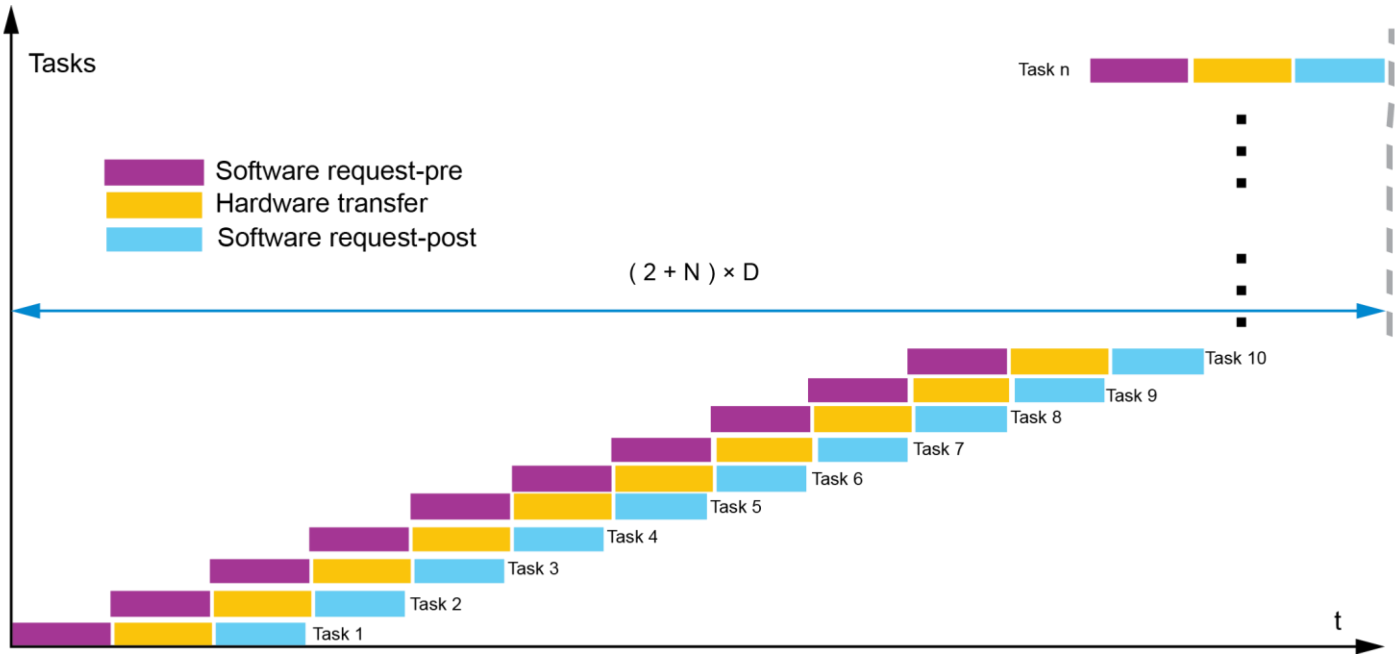

Figure 9: Ideal CMDQ Three-Stage Pipeline

The ideal case will not always be realized: Figures 10 and 11 show two types of pipeline, the first is a single-core condition and the second is a multiple-core condition. These figures show that while the spans differ in duration and parallelism, there is still considerable overlap.

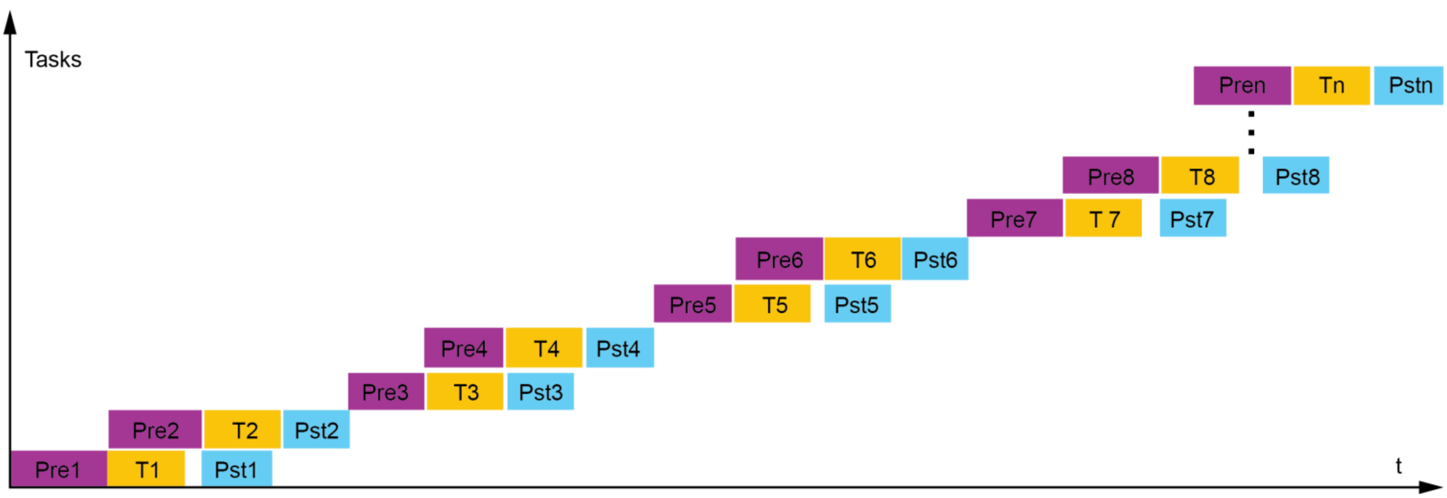

Figure 10: Real CMDQ Three-Stage Pipeline, Single Core

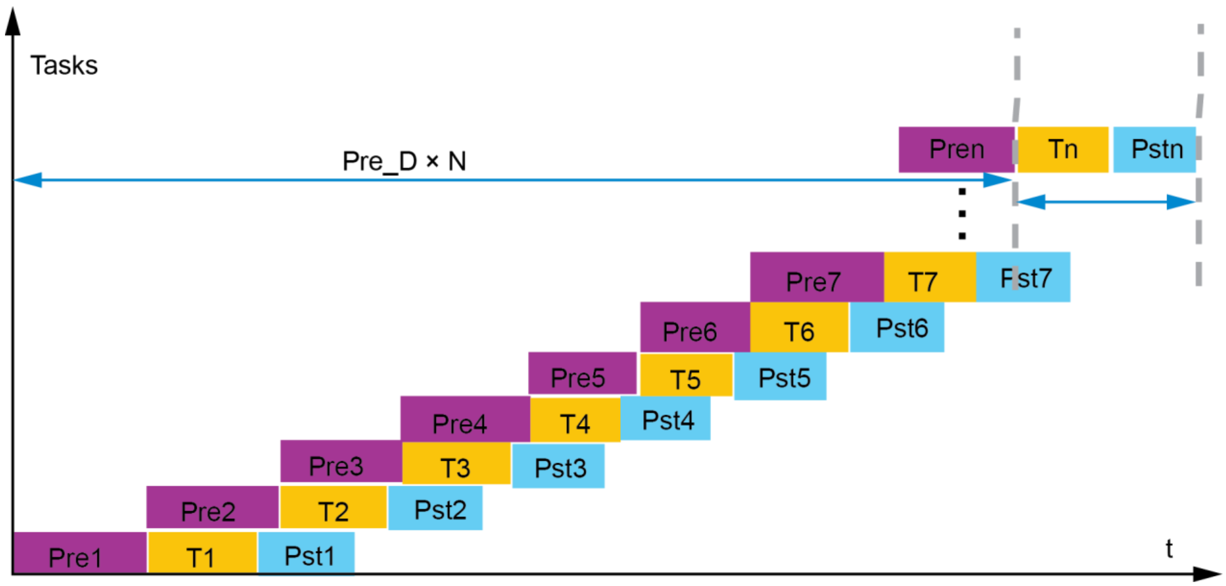

Figure 11: Real CMDQ Three-Stage Pipeline, Multiple Core

Figures 10 and 11 also show that there is overlap between the hardware transfer and software phases. However, in a single-core system the software phases never overlap. Assuming an infinitely fast storage device, the theoretical limit will be determined by the sum of the overhead durations: Pre + Pst stages. In a multiple-core system the software stages can overlap, as shown in Figure 11. The approximate total time cost of N requests with the same operation and data length is:

$Pre\_D \times N + T\_D + Pst\_D \approx Pre\_D \times N^2$

Where Pre_D is the request-pre span duration, T_D is the hardware transfer span duration and Pst_D is the request-post span duration.

Again — assuming an infinitely fast storage device performance will be determined by the largest span in a multiple-core system — the smallest span will be completely overlapped by the largest span.

Deviations from optimal cases are expected in a real system, but the general conclusions will apply. To measure the benefits of multiple-queuing systems, we measured the performance on real systems using tiotest.

Xilinx Zynq Zed Multiple-Thread Performance Based on Tiotest Benchmark

Tiotest is a file system benchmark specifically designed to test I/O performance with multiple running threads. To fairly investigate multiple-thread performance on e.MMC and avoid date merging, we only conducted tests for random access with sync mode.

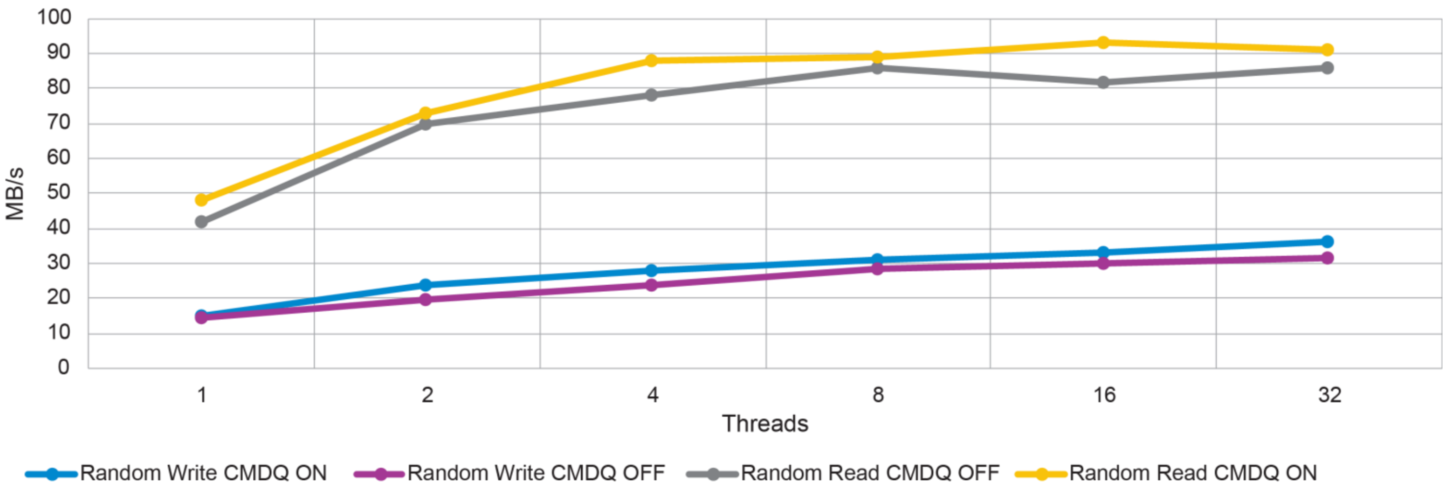

Figure 12: 64KB Random Experimental Data Based on Xilinx Zynq Zed

Item	Request-Pre (µs)	Hardware Transfer (µs)	Request-Post (µs)	Single-Thread Theoretical Speed (MB/s)	Multiple-Thread Theoretical Speed (MB/s)
Random read	737	524	383	38.01	84.8
Random write	1715	259	92	30.23	36.58

Table 3: 64KB Random Experimental Data Based on Xilinx Zynq Zed

Figure 12 and Table 3 show that the random write's theoretical speed (discussed previously) is matched with the measured data. In multiple-thread mode, the 64KB chunk size random write maximum speed for the Zynq Zed board is 36.58 MB/s; however, reads show a slight gap with measured data. The CQ disable and enable comparisons clearly show that CQ improves I/O performance, but maximum improvement is approximately 10% for random write and random read.

Conclusion

Conventional thinking considers the storage device a bottleneck in system I/O performance, driving a need for storage devices with faster and faster I/O speeds. Flash-based storage devices can support high throughputs: e.MMC can reach up to 400 MB/s and UFS can reach up to 1 GB/s. Our study analyzed how much of these high throughputs can be realized by the embedded platforms tested and what the limitations might be.

The current Linux storage stack is based on legacy HDD, and some of its mechanisms benefit legacy storage devices. Our tests show that new-generation, flash-based storage devices have a negative impact on I/O performance and that software overhead has increased as a proportion of system overhead.

Higher-performance storage devices translate to higher costs, and as we show in this study, the available bandwidth is often consumed by the storage software rather than being passed to the end user.

This study also indicates a need for optimization of the existing storage stack, including VFS optimizations,[6] and direct I/O and I/O scheduler improvements because the existing Linux I/O stack is an obstacle for realizing maximum performance from flash-based, high-speed storage devices.[7]

For further information, contact Bean Huo, beanhuo@micron.com, or Zoltan Szubbocsev, zszubbocsev@micron.com.

References

1. Jens Axboe, original contributor
   Explicit block device plugging
   https://lwn.net/Articles/438256
   Apr. 29, 2011

2. Embedded Multi-Media Card (e.MMC) Electrical Standard (5.1); JEDEC

3. Seolhee Lee, Sanghyuk Jung, Yong Ho Song
   Performance analysis of LINUX block I/O for mobile flash storage systems
   Presented: Network Infrastructure and Digital Content (IC-NIDC), 2014 4th IEEE International Conference

4. Kernel Tracing Using Ftrace
   http://blog.fpmurphy.com/2014/05/kernel-tracing-using-ftrace.html

5. Bhavin Turakhia
   Understanding and Optimizing Disk IO
   http://careers.directi.com/display/tu/Understanding+and+Optimizing+Disk+IO
   Oct. 19, 2012

6. Russell Joyce, Neil Audsley
   Exploring Storage Bottlenecks in Linux-based Embedded Systems
   Oct. 8, 2015; Mar. 15, 2016

7. Yongseok Son, Heon Young Yeom, and Hyuck Han
   Optimizing I/O Operations in File Systems for Fast Storage Devices
   Dec. 5, 2016