Design and Implementation of A Mobile Storage Leveraging the DRAM Interface

Sungyong Seo, Youngjin Cho, Youngkwang Yoo, Otae Bae, Jaegeun Park, Heehyun Nam, Sunmi Lee, Yongmyung Lee, Seungdo Chae, Moonsang Kwon, Jin-Hyeok Choi, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang

Memory Business, Samsung Electronics Co., Ltd.

ABSTRACT

Storage I/O performance remains a key factor that determines the overall user experience of a computer system. This is especially true for mobile systems as users commonly browse and navigate through many high-quality pictures and video clips stored in their device. The appetite for more appealing user interface has continuously pushed the mobile storage interface speed up; emerging UFS 2.0 standard provisions a maximum bandwidth of as high as 1,200 MB/s.

In this work, we propose, design, and implement a mobile storage architecture that leverages the high-speed DRAM interface for communication, thus substantially expanding the storage performance headroom. In order to effectively turn the existing DRAM interface into a storage interface, we design a new storage protocol that runs on top of the DRAM interface. Our protocol builds on a small host interface buffer structure mapped to the system's memory space. Based on this protocol, we develop and fabricate a storage controller chip that natively supports the LPDDR3 interface. We also develop a host software stack (Linux device driver and boot loader) and a host platform board. Finally we show the feasibility of our proposal by constructing a full Android system running on the developed storage device and platform. Our detailed evaluation shows that the proposed storage architecture has very low protocol handling overheads and compares favorably to a UFS 2.0 device. The proposed architecture obviates the need for implementing a separate host-side storage controller on a mobile CPU chip.

1. INTRODUCTION

Users expect that a new generation computer system would give higher overall performance than previous generation machines. In the past several decades, continued miniaturization of transistors has fueled the performance growth of CPUs. By comparison, performance of the most common storage device—hard disk drive (HDD)—has virtually stagnated. However, the inception of solid-state, NAND flash memory storage devices (solid-state drives or SSDs) changed this trend completely; the storage interface speeds are advancing fairly aggressively. For example, recent server class NVMe SSDs tap four or eight PCIe (Gen.3) lanes, touting a maximum bandwidth in excess of 3 or 6 GB/s [1]. These speeds are not needed for hard drives capable of transferring well less than 300 MB/s.

Storage I/O performance is equally important for mobile computers. Prior work has shown that storage performance is critical for ensuring overall mobile system performance, as well as the quality of user experience [2]. In fact, storage performance may have a larger impact felt in mobile systems than in server or desktop systems. First, most users operate their mobile system in an interactive manner, with actions that fetch and display large multimedia files. Slow or uneven storage response times will directly deteriorate the quality of user experience. Second, it may be hard to hide low storage performance with common techniques like write buffering and caching in DRAM, because mobile systems typically have limited resources than a server or a desktop system. Led by desires to make mobile experiences more pleasing, the speed of eMMC, the most common storage interface for mobile devices, has reached 400 MB/s [3] to rival desktop SSDs. Also emerging is a new mobile storage standard called UFS. The interface speed of UFS 2.0 tops 1,200 MB/s using two 6 Gb/s lanes [4]. When a new storage interface standard comes out, it will likely be even faster.

Traditionally, the main memory interface (e.g., DDRx) and the storage interface (e.g., SATA, SAS) are two separate traces of technical focus. The former must support byte (or cache block) level access efficiently at a very high transfer rate (tens of GB/s per channel). Access scheduling to take advantage of DRAM bank level parallelism is typically done in hardware. Access latency and timings are strictly deterministic. In the storage interface, by comparison, a storage block (or sector) is the granularity of data transfer and access scheduling is taken care of by software. Latency to storage medium is inherently variable.

In this work, we propose and develop a new mobile storage architecture that takes the DRAM interface as its physical communication layer, which departs from the current dichotomy of the memory and storage interface. There are several benefits to this approach. First, the bandwidth potential of the DRAM interface is very high. For instance, a 64-bit LPDDR3 channel gives over 12 GB/s peak bandwidth; compare this to the 1,200 MB/s bandwidth limit of the currently fastest UFS 2.0 standard. There is no need for defining a new storage interface standard to increase bandwidth headroom.

978-1-4673-9211-2/16/$31.00 ©2016 IEEE

Second, it is not needed to keep dedicated resources to implement a storage interface in a mobile CPU chip. Today, in order to interface with an eMMC or a UFS device, a mobile CPU must incorporate a host controller on chip and dedicated I/O pins for physical connection. Lastly, the proposed architecture could evolve into a unified memory architecture where a memory subsystem offers both memory and storage functions in a synergistic manner.

This work makes the following contributions. First, we prove the feasibility of a mobile storage architecture that leverages the DRAM interface. Especially, we discuss in detail its design (Section 3) and implementation (Section 4 and 5). Our work is the first mobile storage device architecture reported, that successfully builds on the standard DRAM interface. Second, our implementation work includes a prototype storage controller chip fabricated using an advanced 28 nm process technology (Section 4). We present evaluation results obtained on a real system platform (Section 6). Our results demonstrate that the proposed storage architecture achieves substantially low overheads in protocol handling, compared to a state-of-the-art storage device. The developed storage device is shown to perform especially well in terms of random access throughput. Third, we discuss and suggest changes to the existing mobile CPU architecture to make the proposed storage architecture more practical in future mobile platforms (Section 7).

2. BACKGROUND

This section builds the technical background of our work by discussing several topics centered around the storage and the main memory interface technology. Because our work is the first reported attempt to design mobile storage architecture on the DRAM interface, there is little directly comparable work in the literature. However, we mention prior related work in this section and throughout this paper. We also compare our design with existing ones in later sections.

2.1 Mobile Storage Interface Trend

Embedded multi-media card (eMMC) is the dominant mobile storage standard today. The most recent specification defines the maximum bandwidth of 400 MB/s [3]. One of the reasons for its widespread adoption is that eMMC has a simple architecture. eMMC interface uses only 12 pins for interfacing with the host; 8 pins for data, 1 pin for command, and 3 pins for the rest including clock. In addition, eMMC provides features particularly useful for mobile systems, such as *boot mode*. A host can boot solely from an eMMC storage device using this mode, with no help from any other device like NOR flash memory or EEPROM. In order to increase storage performance by fully utilizing parallelism of the storage medium (i.e., multiple NAND flash memory chips), the eMMC specification supports *command queueing* from version 5.1. Command queueing permits multiple host I/O requests to be executed concurrently.

Universal flash storage (UFS) is an emerging mobile storage standard, and was successfully commercialized (found in Samsung's Galaxy S6 phone [5]). The UFS 2.0 standard defines the fastest mobile storage interface to date, and offers up to 1,200 MB/s bandwidth using two 6 Gb/s lanes [4]. It also supports features like boot mode and command queueing. Unique to UFS, the *unified memory extension* feature allows a UFS device to access memory on the host side (e.g., DRAM) [6]. This feature helps the firmware on the UFS overcome limitations due to the small memory capacity of a given storage controller and boost the storage performance [7]. Thanks to its performance potential and rich feature set, UFS is supported by increasingly more mobile CPUs [8, 9].

Is the maximum throughput specified by a state-of-the-art standard like UFS 2.0 sufficient for future mobile systems? Probably not, if the history tells us something; a new standard would appear, and it will likely define a new higher level of performance limit. The approach we take in this work is that we simply remove the limit by tapping the fastest system interconnect—the DRAM interface.

2.2 DRAM Interface Characteristics

DRAM interface is the highest performance interface in a computer system in terms of both throughput and latency. In case of DDR4, for example, a single 64-bit DRAM channel provides up to 25.6 GB/s bandwidth [10]. Mobile systems typically incorporate the low power DDR (LPDDR) DRAM technology rather than a regular DRAM, in order to reduce power consumption.

The standard DRAM interface splits signal pins to separate *command* and *address* transfer from *data* transfer. This separation enables pipelining of information exchange and hence reduces the latency seen by the host system when sending or receiving data. On obtaining a read request, the DRAM returns the requested data to the host after a predetermined number of clock cycles. Similarly, all other timings when accessing a DRAM device, like the time needed between one transaction to another, are governed by a handful of configurable, yet fixed parameters. Basically, these parameters are known to both the DRAM controller and the DRAM device, and the DRAM controller can completely track and control the internal state of the DRAM device as various operations are performed. Working in the deterministic timing principle, the DRAM interface does not allow variable latency access to memory. This is the single most important difference between the DRAM interface and a storage interface. As we will discuss in the next section, we overcome this gap by introducing a layer in our architecture that seamlessly emulates the deterministic DRAM behavior, while allowing variable latency storage operation.

Another important DRAM characteristic for our work is its byte-level access granularity (typical operational access granularity is a cache block). This is in stark contrast with a storage interface, where a 512 byte sector is the most common minimum granularity of access. We observe that it is feasible to embed a block-oriented protocol on the byte-oriented DRAM protocol, but not vice versa.

2.3 Emerging Non-Volatile DIMMs

There are on-going discussions to define new dual in-line memory module (DIMM) types that incorporate non-volatile memory within JEDEC[1]. There are three *non-volatile DIMM*

[1]JEDEC (joint electron devices engineering council) is an industry group that develops open standards for microelectronics. DDRx DRAM, eMMC, and UFS are standards published by JEDEC.

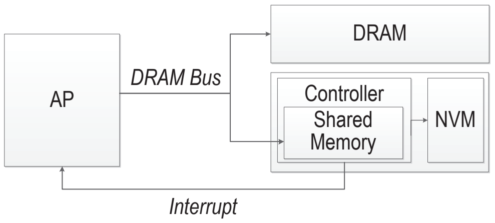

Figure 1: A high-level system architecture.

(NVDIMM) types being conceptualized: NVDIMM-N, -F, and -P [11]. An NVDIMM-N device is a byte accessible DRAM DIMM whose data are backed to on-DIMM NAND flash memory automatically, when the system experiences sudden power loss. It resorts to an auxiliary power source like super capacitor or battery for its back-up operation. When power is restored, the data saved in the NAND flash memory are copied back to DRAM. In effect, an NDVIMM-N device is a fast, DRAM-speed persistent memory. Known applications of NVDIMM-N include fast journaling, write buffering, and caching.

NVDIMM-F is for a block storage device with some non-volatile memory (presumably NAND flash memory) as storage medium. From the viewpoint of application software, an NDVIMM-F device is not distinguishable from a traditional block device like SSD. A device driver abstracts the NVDIMM-F device (on the DRAM bus) as an SSD. Currently, there is no standard command protocol defined for NVDIMM-F. Notice that the approach our work takes shares its basic functional definition with the NVDIMM-F proposal.

NVDIMM-P is for a hybrid memory system where both byte addressability and large capacity are sought in a single DIMM. Discussions around NVDIMM-P are still premature and there are no concrete technical components yet.

The above standardization efforts coincide with industry projects. First, there are already NVDIMM-N products in the market [12–14]. Second, Diablo Technologies has an NVDIMM-F like product with a technology called "memory channel storage" (MCS) [15]. Their architecture defines a bridge chip that connects the DRAM bus and two conventional SSD controllers. Details of the bridge chip architecture are not available. Lastly, Intel has recently announced a DIMM comprised of certain persistent memory [16]. Based on a proprietary DRAM protocol and new functions in the memory controller, they implement a hybrid memory subsystem that combines the available DRAM capacity and the persistent memory DIMM capacity. Again, few details are known about the actual architecture of the memory controller and its inner working.

3. STORAGE INTERFACE DESIGN

3.1 Physical Layer Overview

Figure 1 depicts the overall architecture of our system. The proposed storage device communicates with the host CPU (denoted application processor or AP) through the DRAM bus and an interrupt signal. Data movement between the storage and the DRAM is handled by host software.

The host interface of the storage device behaves as if it were a DRAM device; this ensures that no change is required on the AP to interface with the proposed storage device. In order to achieve this, we introduce a small shared memory buffer as the key storage interface. This buffer is accessed by both the AP (using regular load and store instructions) and the storage controller. When both the AP and storage controller try to access this buffer, the shared buffer arbiter gives higher priority to the AP. Therefore the shared buffer can guarantee the timing parameters as same as a DRAM device to the AP (more details in Section 4.1).

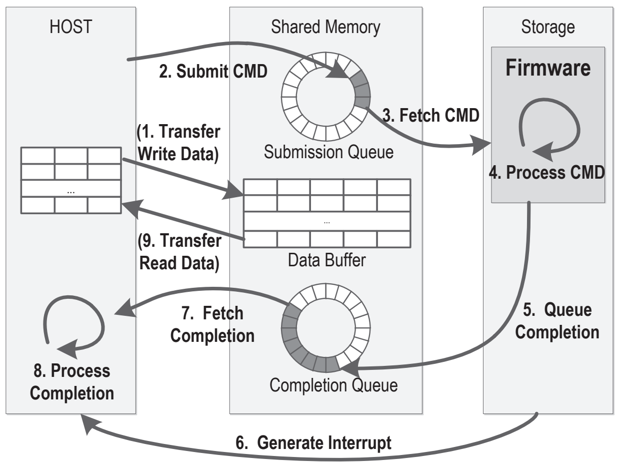

Figure 2: Protocol execution flow.

The shared memory buffer is memory per se, and its content is not interpreted by hardware. Instead, there are a number of special function registers that are exposed through the memory map. These registers allow the host software to deliver certain control information to the storage device and vice versa. The outlined physical mechanisms provide sufficient flexibility in embedding a storage protocol on the DRAM bus.

3.2 Protocol Design

Let us now elaborate our storage protocol. The shared memory buffer (Figure 1) is divided into three regions: Submission queue (SQ), completion queue (CQ), and data buffer (DB). The SQ is where the host sends commands, and the CQ is where the storage device returns command completion results, respectively. The DB stores read and write data temporarily. It is allocated and released in sectors. Figure 2 depicts this logical view of the shared memory buffer.

A submission queue entry consists of *command ID*, *opcode*, *start sector*, *sector count*, and *buffer index*. Command ID uniquely identifies a specific command among several outstanding commands. Opcode denotes the actual command, such as read and write. Start sector and sector count are the start sector address and sector count of this command. Buffer index is the start index in the DB, which is equivalent to a buffer address. I/O data for the command are stored contiguously from this buffer index.

A completion queue entry consists of *command ID* and *completion status*. Command ID is the command ID of the completed command, which is previously given in an SQ entry by the host. Completion status indicates whether this command was completed successfully or not.

Let's examine the execution flow of our protocol, as depicted in Figure 2. First, the host prepares and submits a command to the tail of SQ (#2). If the command to be issued is a write, the write data must be prepared in the DB ahead of command submission (#1). For notification of new command arrival to the storage, the host may write current tail index to a special register similar to the doorbell register of the NVMe specification [17]. Second, the storage device checks the SQ from the head and fetches the newly submitted command (#3). Third, the storage device executes the command (#4). If the command is a read, the storage device transfers read data to a designated location in the DB. Fourth, the storage device returns the command execution result by enqueuing a command completion entry to the tail of the CQ (#5) and notifying the host of the event (#6). Finally, the host finds a newly arrived command completion entry from the head of the CQ (#7), and processes the command completion event (#8). If the completed command is a read, the host transfers read data from the DB to a host buffer (#9).

In the protocol execution flow, actual events are comprised of memory accesses issued by the host-side device driver (from the host to the device) and the interrupt signaling (from the device to the host). By design, data transfer between the host memory (typically DRAM) and the shared memory buffer (#1 and #9) is performed by the host. Data transfer between storage medium and the shared memory buffer is done by DMA engines in the storage controller.

Finally, using queues to communicate between two parties is a common design pattern in storage and communications systems. Queues naturally capture and realize multiple outstanding messages (commands and responses) in a system. This aspect is especially important for storage devices where parallelism (e.g., multiple NAND flash memory dies) and asymmetry in access time (e.g., access time is a function of input and the current device state) are present. While Figure 2 has a single pair of SQ and CQ, there is nothing that prevents us from implementing multiple queue pairs. In fact, the emerging NVMe standard supports a plurality of queue pairs so that multiple CPUs can have their own I/O queues for lock-less device interaction [17].

3.3 Design Considerations

Completion notification delivery. A conventional storage interface has a completion notification mechanism. For example, eMMC (host controller) generates an interrupt for completion notification while hiding the interaction between the eMMC host controller and the device controller. In another example, NVMe SSDs leverage the underlying PCIe interrupt mechanisms (legacy signaling or message-based). However, the DRAM interface does not provide an equivalent host notification mechanism, as a DRAM is a deterministic and passive device (Section 2.2). There are two strategies to overcome this gap. The first strategy is to utilize polling in the host device driver. More specifically, once a command is submitted, the device driver may periodically check the status of the CQ to see if the submitted command was fulfilled and how. Another strategy is to borrow a notification mechanism available in the system. In a typical mobile system, the mobile AP comes with GPIO pins that can be configured as external interrupt signals. In order to save precious host CPU cycles, we take the second strategy in this work. Note that with an extremely high I/O rate, polling may prove more efficient than interrupt [18]. It is a classic topic in I/O processing and describing in detail the trade-offs between the two strategies is not warranted in this paper.

Command submission integrity guarantee. Correct execution of the protocol processing flow of Figure 2 depends on the memory consistency model. Take steps #1 and #2 as an example; the host system must guarantee that the write data are written to the shared memory buffer before the corresponding write command is issued to the SQ. If this ordering is violated, the storage controller may inadvertently write stale data to the storage medium. Therefore, the host system implementation must utilize proper mechanisms to guarantee ordering of memory accesses that belong to two ordered protocol phases. In the target platform we use (Section 5.1), complications are caused by L1 and L2 caches. We treat the shared memory buffer space as non-cacheable and apply proper memory ordering primitives in our device driver to guarantee command submission integrity. As we will discuss next, main data transfer (to and from the DB) is done with a hardware mechanism in our implementation.

Data transfer offloading. Unlike a conventional storage interface that builds dedicated control logic on both the host and the storage, our storage interface architecture does not have a control logic on the host side (i.e., the DRAM bus is directly connected to the AP). Software-based memory copying (a series of load and store instructions) consumes many host CPU cycles [19] and results in prohibitively large overheads at high I/O rates. Fortunately, a modern mobile AP generally comes with general DMA controllers that can be used for memory to memory data movement [20]. Our device driver implementation aggressively utilizes DMA facilities available in a mobile AP. Section 5.4 further discusses this consideration.

4. STORAGE IMPLEMENTATION

This section describes our prototype controller chip implementation as well as the firmware that runs on the controller.

4.1 Controller

Figure 3 shows the overall architecture of the prototype controller. The controller consists of three main blocks: *host interface*, *core backbone*, and *flash interface*. The blocks are integrated with an AXI interconnect, which provides a high speed data bus. *The host interface block* receives I/O commands, and returns command execution result, via a shared memory buffer which is an emulated DRAM device. *The core backbone block* facilitates firmware operations by providing computation, memory, and data movement IPs. *The flash interface block* enables NAND flash memory operations with high performance and reliability, using multiple NAND flash memory controllers (FMCs) and ECC engines. The features of the controller chip are shown in Table 1.

Host interface. This block is the most unique element in the proposed controller compared with existing mobile storage

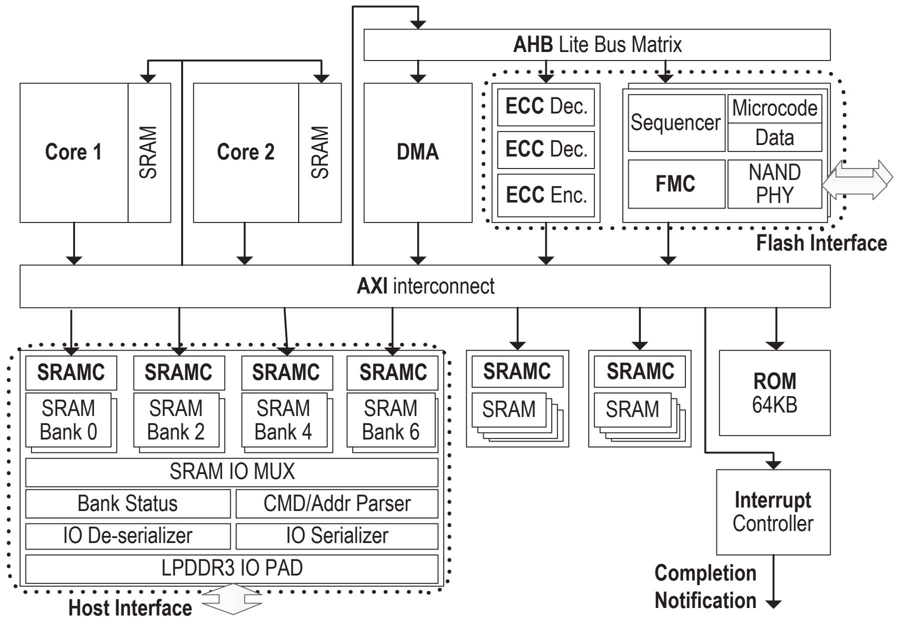

Figure 3: Storage controller implementation.

Table 1: Controller chip features. Power consumption includes the NAND flash memory's power.

Fabrication Process		28 nm CMOS
Controller Package Size		20.9 mm^2 (5.5 mm $\times$ 3.8 mm)
eMCP Package		11.5 mm $\times$ 13.0 mm $\times$ 1.2 mm
Clock	Core	266 MHz
	Host Interface	800 MHz (1,600 Mb/s)
	NAND Flash Interface	233 MHz (466 Mb/s)
Power Consumption		560 mW @ Seq. Read 450 mW @ Seq. Write

controllers. It provides a shared memory buffer as host interface, which is *an emulated DRAM device*. The host accesses the buffer in this block as if it were accessing a DRAM device. This DRAM emulated buffer is implemented by general SRAM and supporting logic. In order to interface with the host LPDDR3 bus, we use LPDDR3 I/O pads identical to those of a conventional DRAM device. Behind the LPDDR3 I/O pads, a command/address parser interprets the target address of a DRAM request. The address is forwarded to the bank status register, which keeps the row address for each bank, and generates a concatenated address of row and column for each request.

The memory area is made up of eight SRAM banks, each of which is a number of SRAM macros. Each SRAM macro is single-ported (rather than dual-ported) to minimize the SRAM logic area. Since this buffer is shared by both the host and the storage controller itself, access conflict to this buffer can occur. We give higher priority to the access from the host, and preempt the access from the controller itself. This preemption is inevitable since the storage controller must emulate the behavior of a DRAM device, which should operate in a fully synchronized manner with the host memory controller. Note that, however, this preemption will not incur significant performance degradation. We use a dedicated SRAM controller for each SRAM bank, thus the conflict probability is lower than $1/8$. Finally, the accessed data from the SRAM macro are delivered to data signal lines through the serializer.

Core backbone. This block provides basic facilities for operating storage device such as CPU core, SRAM, and DMA controller. It has two symmetric CPU cores, each of which has its own scratchpad memory for storing instruction and data. Thanks to the scratchpad memory, the core gets predictable high performance even without a cache subsystem.

There are two additional SRAM modules for general purpose, which are controlled by dedicated SRAM controller for achieving high bandwidth. The memory mapping logic of the AXI interconnect makes memory access to the SRAM modules to be processed in an interleaved manner, so the bank conflict frequency is reduced. A general DMA controller is employed to transfer data efficiently under software control. In addition to the data transfer function, it has predefined computational logic such as bit-wise AND, XOR, and counting, which can be exploited for accelerating time-consuming data processing such as data encoding and decoding.

There are also peripheral IPs such as interrupt controller, timers, and GPIOs. The interrupt controller takes an important responsibility for I/O command completion notification to the host. It generates an interrupt signal to the host whenever an I/O command completes, and removes the necessity of command completion status polling at the host side.

Flash memory interface. This block controls NAND flash memory chips while ensuring data integrity using sequencer and FMC. The sequencer is a special purpose microcontroller which generates a high-level control signal sequence. Then the FMC generates primitive control and data signals for operating NAND flash memory chips according to the high-level signal sequence given by the sequencer. It also transfers user data between internal SRAM and NAND flash memory. The NAND PHY, implemented as a hard macro, enables high speed data transfer between the FMC and a NAND flash memory chip. It has delay-locked loops (DLLs) and delay control logic to find the center of a valid data window and to generate center-aligned data strobes. Finally, there are two shared ECC decoders and one ECC encoder. We adopt this structure to accommodate asymmetric and variable throughput performance of the main ECC algorithm that is based on a *low-density parity-check* (LDPC) code.

4.2 Firmware

Our firmware is comprised of three layers: Host interface layer, flash translation layer, and flash interface layer. The role of each layer is briefly described below.

Host interface layer (HIL). This layer is the top layer of the firmware stack and interacts with the host system using the shared memory buffer according to our storage protocol. It receives new commands and returns command execution results from and to the host. In order to find a newly submitted command entry, it checks the head of the SQ. If there is a new entry, it fetches the SQ entry, interprets the command, and converts the corresponding submission entry into an internal data structure for further processing by the FTL. Finally, it sends a request to the FTL. When handling a command completion event, the HIL is notified by the FTL that the command execution has completed. Then the HIL prepares a CQ entry. Finally, it enqueues a prepared CQ entry to the tail of the CQ.

Flash translation layer (FTL). When a read request arrives from the HIL, the FTL looks up its logical to physical mapping information to identify the actual NAND flash memory physical address where the corresponding data are stored. Once this look-up is completed, it delivers the request to the FIL. The FIL will fetch the requested data from NAND flash memory, then the FTL notifies the HIL that the data are ready. In case of a write request, the FTL determines the destination address in a NAND flash memory block and delivers the write request with the destination address to the FIL. When the FIL notifies that programming was successful, the FTL updates its logical to physical mapping information with the new physical address. Finally it notifies the HIL that the write request is completed.

Flash interface layer (FIL). The FIL is the bottom layer of our firmware stack, which interacts with the FMCs. It receives a request (and data, in case of a write command) from the FTL, converts the request into a corresponding NAND flash memory command, and issues the command to a NAND flash memory. When a NAND flash memory command completes, the FIL checks the completion status. Then it notifies the request completion to the FTL.

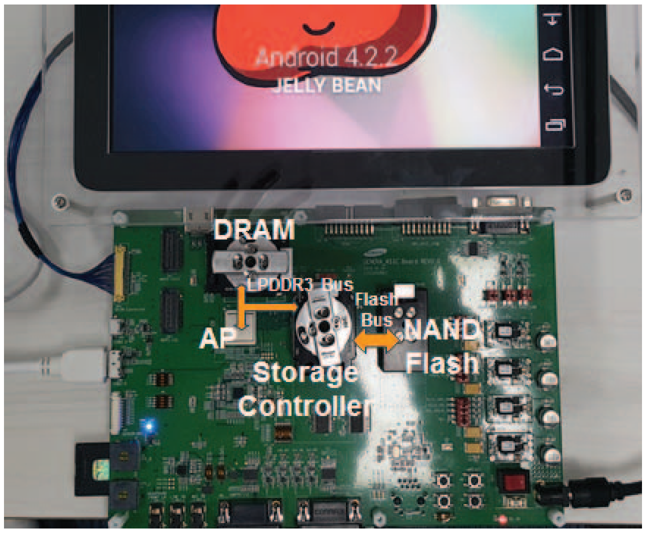

Figure 4: Host platform board.

5. HOST SYSTEM IMPLEMENTATION

The proposed storage interface architecture does not require any hardware change on a given mobile AP. This section first describes our platform hardware that incorporates a commodity AP and our controller chip. We also discuss how we implemented the boot loader and the storage device driver.

5.1 Prototype Platform Hardware

We built a host platform board to prove the concept of the proposed storage interface architecture and the performance of our storage device implementation. We take an off-the-shelf mobile system reference board [21] as the starting point and modestly changed its structure. The changes reflect the high level system architecture sketched in Figure 1.

Figure 4 is a photo of our platform. The application processor on the platform is Samsung's Exynos 5250; this mobile AP was made available from 2012 and adopted in several commercial products including Samsung's Chromebook and Google's Nexus 10.[2] It has two separately controllable LPDDR3 DRAM channels. One channel is reserved for actual DRAM chips. Our storage controller chip is connected to the other channel. Four 64 Gb NAND flash memory chips are integrated, for a total capacity of 32 GB. Table 2 lists the specification of the platform.

5.2 Boot Loader

We use *universal boot loader* (U-Boot) [22] for early-stage system initialization and OS loading. During system initialization, the DRAM subsystem is configured. Since the proposed storage interface appears to be a DRAM to the host system, the system must initialize the shared memory buffer as if it would do to a regular DRAM. We modified U-Boot to reflect this arrangement. The modification is relatively

[2]The mobile AP chip that we use is a bit outdated. We chose this platform and the mobile AP primarily because these were available for use and modification.

Table 2: Summary of the developed platform.

Application Processor	Samsung Exynos 5250 (ARM Cortex-A15 dual @ 1.7 GHz)
DRAM Interface	Two LPDDR3 channels @ 600 MHz
DRAM Capacity	1.5 GB (DRAM channel 0)
Storage Capacity	32 GB (DRAM channel 1)

straightforward, with two main tasks being: (1) preparing a proper memory map and (2) setting DRAM access timings.

5.3 Device Driver

We developed a Linux device driver (kernel version 3.4.35) to operate the proposed storage device. For initialization, the device driver maps physical addresses of the proposed storage interface to virtual addresses. For physical to virtual address mapping, it calls the `ioremap_nocache()` kernel API to mark the SQ and CQ areas as uncacheable. It calls `ioremap()` for the DB area. After the above address mapping is done, the device driver identifies capabilities of the attached storage device, such as capacity and supported primitives, by issuing an administrative command to the storage.

If it successfully fetches the identity of the storage, the driver registers the storage device as a block device to the Linux kernel by calling the `add_disk()` API. From this point on, the storage device is accessed just like any other block devices like eMMC and SATA SSD. It exposes the `make_request()` API, which accepts the `bio` (block I/O) structure as input; the upper layer software requests the storage device to process a storage I/O by calling the API.

Our device driver has two handlers for I/O request processing: *submission handler* and *completion handler*. When a storage I/O request arrives at the device driver, the submission handler is called. It converts the request into our storage protocol commands. It allocates some entries in the DB to the request. Then it submits a command to the SQ. In case of write, it also copies write data from the corresponding `bio` buffer to the DB before it submits a write command.

Storage I/O completion is handled by the completion handler. It is called by the interrupt handler of the GPIO pin registered as completion interrupt of our storage device. It checks the CQ entry and notifies completion of the matching `bio` request to the upper layer in the Linux kernel by calling `bio_end_io()` if the execution result of the CQ entry is successful. In case of read, it also copies read data from the DB to a previously allocated `bio` buffer.

5.4 Software Optimizations

We perform the following software optimizations to realize high performance.

Transferring data by DMA. In order to make data movement between the DB and a `bio` buffer efficient, we utilize a memory-to-memory DMA facility in the Exynos 5250 AP [20]. On platforms that do not provide a memory-to-memory DMA, the device driver will have to depend on software memory copy (like `memcpy()` in the Linux kernel). Software memory copy is inherently CPU heavy because it executes load and store instructions to move data. Our measurements show that small random I/O performance with the DMA is an order of magnitude higher than without the DMA. Unfortunately, the DMA in our prototype platform does not provide high enough throughput to show the full potential of our proposed approach. The limitations will be discussed in more detail in the next section.

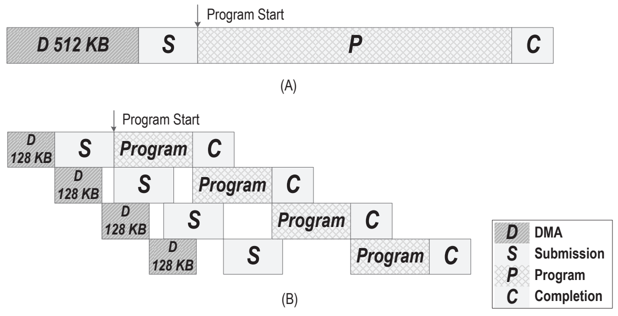

Figure 5: Split I/O.

Splitting I/O requests. Our device driver splits a `bio` request from upper layer into several storage commands with a smaller chunk size. There are two reasons for this. First, it can't submit a storage command unless it secures buffer space in the DB for the command. Moreover, the upper layer may submit a `bio` request whose I/O size is bigger than the entire DB capacity.

Second, splitting a large I/O may in fact increase performance. It is noted that our storage protocol permits command submission or completion only after corresponding data are prepared in the DB. Take the illustration in Figure 5 then; assume first that the device driver receives a `bio` request to write 512 KB data. It may submit the write command after transferring entire 512 KB data to the DB. On the other hand, the device driver may split this command into four 128 KB write commands. Comparing with the latter, submitting a large write command incurs smaller total protocol handling overheads. However, the time to start actual NAND flash memory operation (as indicated with "Program Start") is longer because time to transfer the whole requested data takes longer. Consequently, the idle time of the storage medium is longer. If the raw bandwidth of the storage medium is smaller than the interface throughput, it makes sense to keep the storage medium as busy as possible by reducing its idle time.

What is the right size of a split I/O request? In principle, a preferred size is the smallest one that makes the underlying storage medium all busy.

Controlling cache invalidation overheads. In order to maintain cache coherency, CPU cache is invalidated both before and after DMA data transfer. We call them *pre-invalidation* and *post-invalidation*. We measured that cache invalidation over a 512 KB area takes nearly 290 μs on our host platform, which is substantial. We reduce the cache invalidation cost by splitting a large DMA transfer into smaller transfers, and executing smaller cache invalidation operations in a pipelined manner.

Minimizing external fragmentation in the DB. As many, various storage I/O transactions proceed, the DB capacity may become fragmented. We must minimize (the negative impact of) external fragmentation for two reasons. First, in our design, a storage command (read or write) has a single buffer index within its SQ entry. Because our protocol does not currently defines or allows scatter-gather I/O over the DB region, the buffer space for a command must be contiguous. Consequently, if the DB is heavily fragmented, a large `bio` request should be split into multiple storage commands with smaller chunk size, which incur considerable command processing overheads. To make the process of buffer space allocation and deallocation efficient, we adapt the well-known buddy memory allocation algorithm [23, 24] in buffer management.

Besides the optimizations we described above, we paid attention to reducing overheads in handling primitive operations like enqueuing, dequeuing, and interrupt handling. For example, fine-grained locking is used when managing queues.

6. EVALUATION

6.1 Evaluation Setup

We evaluate the proposed storage device architecture using the real platform hardware described in Section 5.1. In the current platform, we have a separate, primary storage device (SD card) for OS booting as an existing AP does not support booting from our device. A positive side-effect of this arrangement is that unexpected I/O requests from the OS are directed to the primary storage and not to our storage device that is under careful experimental investigation. We create a single partition on the storage device and format it with the ext4 file system.

All results we obtain and report in this section are measured. We repeat measurements many times to eliminate noise and have high confidence. We employ a microbenchmarking method because our focus is on characterizing the device itself in this work (rather than studying application level consequences). In our method, we employ a thread that issues an I/O request of a specified size and blocks until the request is fulfilled. The thread then issues another I/O request and this process repeats. In order to impose higher bandwidth requirements, we employ more threads that can issue a request simultaneously. The number of threads we deploy is informally called *queue depth* (QD) in this work. Hence, QD4 implies that we have four threads that concurrently and independently issue I/O requests. To put another way, QD4 means that there are at most four outstanding commands being served by the storage.

6.2 Protocol Overhead

Let us first examine the efficiency of the proposed storage protocol based on measured QD1 performance. For quantification, we define *storage protocol overhead* as $1 - \frac{\textit{protocol bandwidth}}{\textit{interface bandwidth}}$. In case of the proposed storage protocol, *interface bandwidth* is determined by a data movement method used (i.e., DMA). We do not consider the raw LPDDR3 bandwidth as interface bandwidth because the practical data transfer rate is limited by the data movement method

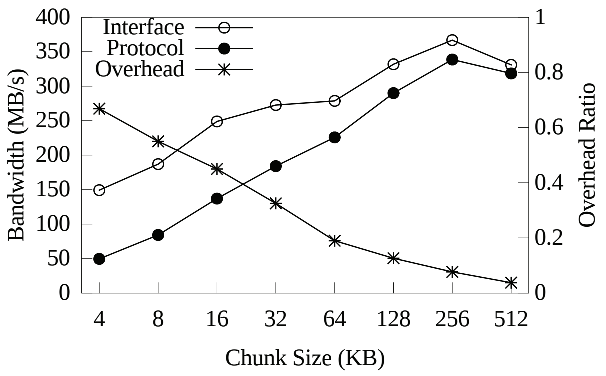

Figure 6: Interface bandwidth and protocol bandwidth.

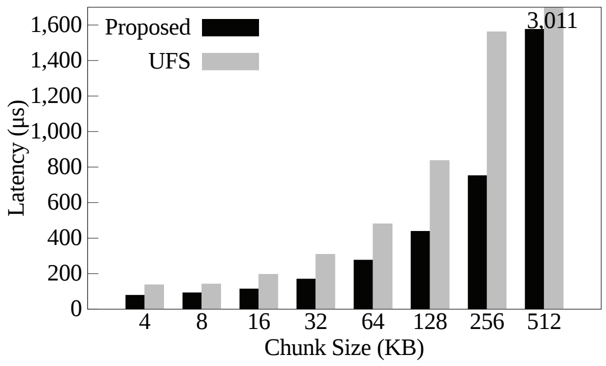

Figure 7: Protocol overhead latency: Proposed vs. UFS.

(which provides far lower bandwidth than the LPDDR3 bandwidth on our platform).[3] *Protocol bandwidth* includes processing overheads for command submission and completion, as well as data read and write. Obviously, the protocol bandwidth is lower than the interface bandwidth.

To obtain results, we collect an event trace in the device driver, and analyze both the interface bandwidth and the protocol bandwidth. Moreover, we instrument the host interface layer of our prototype storage device firmware to return immediately upon receiving a storage I/O request without NAND flash memory operation. In this way, we exclude the relatively long NAND flash memory access time so that we can more precisely extract overheads of the protocols under study.

The first result is presented in Figure 6. It shows both the interface bandwidth and protocol bandwidth of our storage implementation for each access chunk size. Under the current evaluation setup, interface bandwidth is 366.7 MB/s for a read command with at a 512 KB chunk size. It is shown that the relative overhead portion shrinks as we increase the access chunk size. Minimum and maximum relative protocol overhead was 3.8% (at a 512 KB chunk size) and 66.8% (at a 4 KB chunk size), respectively.

Figure 7 shows the same result in perspective, by plotting the protocol processing latency of our prototype and a commercial UFS implementation (see Section 6.5 for details

[3]The maximum DMA throughput on our prototype platform is only around 450 MB/s.

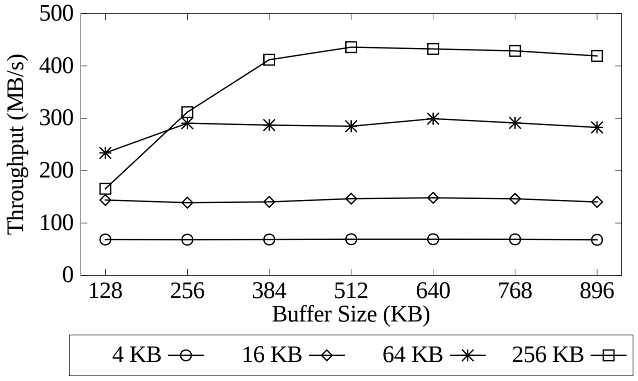

Figure 8: Maximum throughput at QD4 against buffer size.

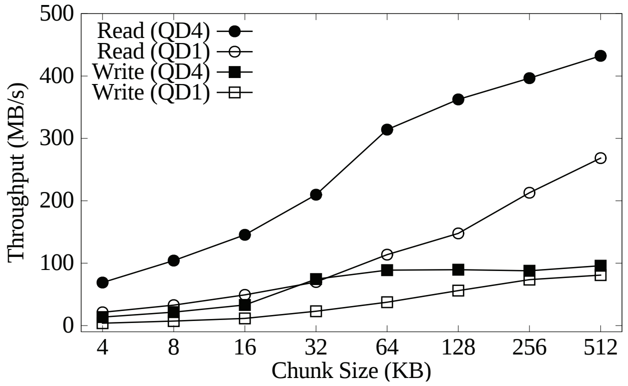

Figure 9: Throughput against access chunk size.

of it) next to each other. Our protocol processing latency is consistently smaller than that of the UFS device.

6.3 Interface Buffer Size and Throughput

Let us now turn to how the storage throughput changes when the interface buffer size is varied. The result is in Figure 8. We choose to study this parameter because it is a primary factor that governs the host interface bandwidth as well as an important cost factor. We measure the storage's read performance (including actual storage medium access) at QD4 as we vary the interface buffer size (X axis) and the access chunk size (different lines).

We get the peak performance of 436 MB/s with a 256 KB chunk size and the buffer size of 512 KB. At all chunk sizes we studied, there is no performance gain with additional buffer capacity over 512 KB. It is also shown that at a smaller access chunk size, peak performance is reached using a smaller buffer size.

In the result, the performance of 256 KB chunk access is lower than that of 64 KB when the buffer size is 128 KB. The reason for this unintuitive result is attributed to severe buffer contention (four threads vie for buffer space as each of them submits a 256 KB I/O to a 128 KB buffer). Further system level optimization would remove this cross point, but we believe the current result correctly predicts the performance trend against the interface buffer size. Note that we omit the trade-off between write throughput and buffer size because write performance is dominated by the NAND flash memory's write bandwidth.

6.4 Basic Performance

We measure the performance of our storage prototype by running a popular benchmark program called *IOZone file system benchmark* [25].[4] Figure 9 shows the result, measured in QD1 and QD4 configurations.

For both read and write, throughput grows as chunk size increases. The maximum read throughput is 432.2 MB/s at a chunk size of 512 KB and QD4. Similarly, a maximum write throughput of 95.8 MB/s is measured when the chunk size is 512 KB at QD4. The QD4 write throughput is nearly saturated at the chunk size of 64 KB.

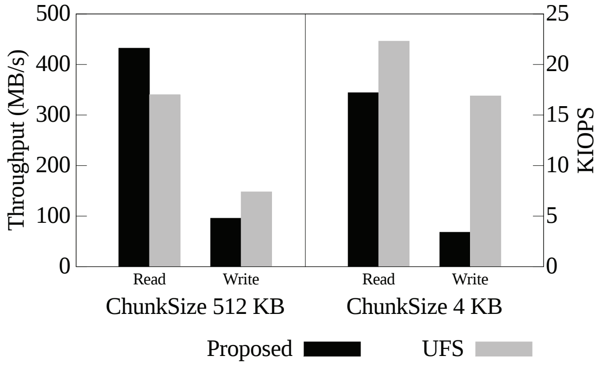

Figure 10: QD4 performance of our prototype and a UFS device.

6.5 Comparison with a UFS 2.0 Device

In this subsection, we compare the performance of our prototype storage and a state-of-the-art UFS device found in a recent commercial product. Table 3 has the relevant specification of the two platforms. It is clear from the table that the UFS platform has much higher performance than our prototype platform, in terms of CPU power, DRAM capacity and speed, the NAND flash memory capacity, and the level of engineering efforts put into implementation. Especially, the UFS device has twice the storage medium level throughput compared to our prototype (due to higher parallelism). We note that the experiment we perform in this subsection is not for apple-to-apple comparison of the two platforms and the storage devices per se. Rather, we would like to highlight the strength and weakness of our current prototype as well as other important factors that affect storage performance.

Figure 10 presents the result. Although our storage device has many critical disadvantages, it shows higher performance in case of sequential read. For both 512 KB write and 4 KB read, lower throughput of our prototype is expected and reasonable since our prototype has only half the capacity of the UFS device. Lastly, the 4 KB write throughput of our prototype device is slow mainly due to two factors, lower parallelism (fewer NAND flash memory chips) and lack of

[4]We run IOZone using the following parameters:
```
$ iozone -ecI -+n -L64 -S32 -r(CHUNKSIZE)k -s1024m
-i0 -i1 -t1
```

a *cached write* feature. With this feature, a storage device notifies the host immediately of command completion, after receiving write data into an internal buffer, hiding NAND flash memory write latency to the host. In our current implementation, we do not support this feature and that led to a relatively low random write performance of our prototype. Overall, the results we obtained so far are promising; if our platform had an equivalent DRAM throughput, NAND flash memory capacity and system level tuning, the proposed storage would perform noticeably better.

7. DISCUSSIONS

This section discusses technical challenges facing our approach that we feel important to consider, for successful development of a system with the proposed storage architecture. We also give possible solutions to the challenges and suggestions for further system improvements.

7.1 Technical Challenges

Interleaving. Modern computer systems commonly employ an *address interleaving* scheme to realize a high bandwidth memory subsystem [26]. When interleaving is applied, the main memory's physical address space is split between multiple DRAM channels and/or multiple DRAM ranks/banks in a granularity that is much smaller than the physical memory size. For example, given two memory channels, an interleaving scheme could map consecutive cache blocks (or memory pages) to two memory channels in an alternating manner. In essence, interleaving opts to increase the utilization of available memory and bus resources.

When the proposed storage device is applied, complications may occur as it appears in the DRAM's address space. For example, imagine that the storage is assigned a DRAM rank. While it complies with the DRAM protocol and is a DRAM in any sense from the viewpoint of the DRAM controller, it does not provide regular memory capacity as other co-located, real DRAM devices. Hence, the storage device can't participate in interleaving. Ideally, a well-designed system will have DRAM devices balanced across the buses and interleaved. Moreover, the storage device is kept out of that interleaved memory space. If the underlying AP is not flexible in its address interleaving support, this task can become a challenge.

Increased DRAM bus bandwidth consumption. The proposed DRAM interface storage architecture consumes more DRAM bus bandwidth than a conventional storage having a dedicated communication channel. To illustrate this, let's consider a read command processing scenario. When the read data are ready, the host CPU will fetch the data using a DMA *over the DRAM interface* consuming DRAM bus bandwidth (as data are moved from the storage to the DMA buffer). The DMA then moves the data in its buffer to a destination (a DRAM location), consuming DRAM bus bandwidth the second time. With a conventional storage device like eMMC and UFS, data transfer from the storage to the DMA buffer would not incur DRAM bus bandwidth consumption as there is a separate, dedicated data bus from the storage to the DMA.

This is a trade-off because our approach eliminates a dedicated on-chip storage logic and bus. Normally, additional DRAM bus bandwidth consumption during storage I/O does not pose a serious performance hit because peak I/O bandwidth requirements of typical applications are not high compared to the available DRAM bus bandwidth, and the applications issuing heavy I/O requests are naturally I/O bound. Under very heavy DRAM traffic (e.g., due to certain memory-bound computation), however, increased DRAM bus traffic due to storage I/O may impact system performance.

DRAM interface pin count. Our approach requires that the storage controller has a complete DRAM interface (to look like a DRAM). This may impact the cost of the storage controller because a DRAM interface typically has more signal pins than a storage interface. For example, eMMC has only 12 pins. UFS 2.0 needs 7 pins (1 lane) or 11 pins (2 lanes). By comparison, an LPDDR3 interface has 38 pins (for $\times16$) or as many as 60 pins (for $\times32$). This adds no overhead to the AP because it must have DRAM channels anyway, but may present a challenge to making the DRAM interface storage controller cost-effective (in terms of chip size). Indeed, the chip size of our controller was determined by the I/O pads rather than the core logic of the chip.

7.2 Suggestions

Flexible DRAM address mapping. We would like flexible DRAM interleaving support in mobile APs. Let's consider two scenarios: (1) The storage device participates in interleaving; and (2) The storage device does not participate in interleaving. In the first case, it is desired that the interleaving granularity is flexible. Suppose the granularity is fixed and very large. Then, an interleaving unit in the address space to which the storage device is mapped is not fully filled up. This is fine for the storage device as long as all its exposed address space falls within the mapped unit. However, a large physical address space is unused (wasted) in this case. Now, suppose that the granularity is very small (say a cache block). This implies that the memory capacity (buffer space and registers) exposed by the storage device is split and appears as scattered small capacities (cache blocks) in the physical address space. It won't be feasible to realize a device driver to work with the storage device then. Moreover, because the memory capacity contributed by the storage device can't be allocated for normal uses, the actual DRAM capacity that joins the same interleaving group as the storage device is not usable by the OS at all.

In the second scenario where the storage device does not participate in interleaving, there are again two cases to consider. In the first case, DRAM devices are connected to the AP in a manner that is balanced and symmetric in terms

Table 3: Host platforms.

	This Work	UFS (Galaxy S6)
Application Processor	Exynos 5250 (ARM Cortex A15 Dual @ 1.7 GHz)	Exynos 7420 (ARM Cortex A-57 Quad + A-53 Quad @ 2.1 GHz)
Main Memory	1.5 GB (LPDDR3 @ 600 MHz)	3 GB (LPDDR4 @ 1,555 MHz)
Storage	32 GB	64 GB

of bus connection and the number of ranks/banks. In this case, it is ideal to have the DRAM capacity interleaved in any way the system warrants, and the address space of the storage device appears linearly somewhere in the physical address space. In the second case, the DRAM configuration is asymmetric (e.g., two DRAM channels each see a different capacity). In this case, we would like that a good chunk of the total DRAM capacity is interleaved for performance, the remaining capacity occupies a linear space, and finally, the storage device presents a contiguous capacity in the address space.

In summary, capabilities of an AP to flexibly map physical address space among all DRAM devices for performance and usability are valuable and necessary for our approach to have more impact. Some APs we have seen do not offer such flexibility or are limited in their capability.

Memory transaction order guarantee. In addition to the data reordering requirement in the presence of L1 and L2 caches (Section 3.3), there is another source that may ruin command submission integrity: the DRAM controller in an AP; it may aggressively buffer and re-order memory transactions for performance reasons [27]. Hence, the CPU must provide a mechanism for the programmer to enforce end-to-end ordering of memory operations on demand in the presence of deep buffering and memory access scheduling. The x86 architecture has recently added instructions like `pcommit` and `clflushopt` to this end [28]. In the development platform that we used, we did not face correctness issues related with end-to-end memory access ordering, when we applied conventional memory ordering primitives like memory barrier.

Serial DRAM interface. Currently DRAM interface standards define a parallel-fashion interface, requiring many signaling pins (pursuing short latency while achiving high bandwidth). In contrast, increasingly more storage devices adopt serial physical links (e.g., SAS, SATA, UFS, and NVMe). If the memory interface adopts a serial link technology, we could reduce the number of pins on the CPU chip [29]. Such an interface also reduces the number of pins in the proposed storage architecture, likely reducing the controller chip size and the fabrication cost.

Another important benefit of using serial links is its natural support for variable latency. We can conceptualize a hybrid memory system synergistically comprised of multiple memory components having different characteristics, like DRAM and NAND flash memory. Such a memory system may flexibly offer byte-oriented and block-oriented access models over the same physical interconnect.

Booting support. In a mobile system, system booting proceeds in three stages. In the first stage, a boot loader image is loaded from a storage device to a small SRAM within the AP by the boot ROM code embedded in the AP. Then, the boot loader initializes hardware including the DRAM and the storage device. Thereafter, these devices are available for remaining booting tasks. Finally, it loads the OS image from the storage device to the DRAM, and starts executing the OS by jumping to the entry point of the OS.

Because the boot ROM code has access to a small SRAM in the first stage, loading the boot loader image from the storage device must be achieved using a simple method. Typically, the boot ROM code can't utilize a full storage command set in this stage because the storage device driver has not been loaded due to lack of resources. In order to overcome this, some legacy mobile systems employed a NOR flash memory that is accessed like an SRAM that requires no initialization. An eMMC storage device provides a boot mode that is triggered simply by driving its CMD line low for 74 clock cycles [3]. Once this mode is triggered, an eMMC device sends the boot loader image to the host.

How can we support booting from a DRAM interface storage device? Because the storage device is a DRAM device from the viewpoint of the DRAM controller, initialization of the DRAM controller and the device must be done in the first stage. Then, a boot loader image should be retrieved from the storage device. There are two important ingredients that must be had in this step. First, the system has to understand where the storage device is located in the address space. This is not straightforward because platforms may have a different DRAM and storage device configuration while the boot ROM code in the CPU is the same. Second, the storage device will have to provide a simple method (like the boot mode of eMMC) without the full device driver.

High-performance embedded DMA controller. As previously described, the maximum bandwidth of the proposed storage device may be limited by data movement between the host memory (`bio` buffer) and the host interface buffer. In order to get the most out of the extremely high throughput DRAM interface, a high speed embedded DMA controller is needed. In that regard, the DMA facility in the experimental platform used in this work was unsatisfactory; its maximum throughput was only around 450 MB/s. We hope to see general-purpose high performance DMA engines that can saturate the throughput of a given DRAM bus. Certain server chipsets provide DMA controllers for general-purpose, high speed data movement [30].

8. CONCLUSION

In this paper, we described a new mobile storage architecture and its product-strength implementation results. This work gives the first reported mobile storage device architecture that builds on the standard DRAM interface, remaining a solid reference for future storage architecture design endeavors. Theoretically, the approach we took obtains the largest headroom to scale storage performance, because a DRAM channel is the highest performance interconnect between a CPU and the rest of the system. Our design work focused on defining a storage protocol that is translated into regular memory accesses on the DRAM bus. We showed through experiments that our design leads to cost-efficient implementation.

Solid-state storage medium technology will continue to evolve in terms of both capacity and performance. To adequately deploy the rapidly advancing storage medium technology, a clean-slate approach to architecting a storage device, like this work, will be of growing importance.

References

[1] Samsung Electronics, "Samsung Rolls Out Strong Line-up of V-NAND SSDs Primarily Geared to Enterprise

and Data Center Customers." http://www.samsung.com/semiconductor/insights/news/22663, August 2015.

[2] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," *ACM Transactions on Storage (TOS)*, vol. 8, no. 4, p. 14, 2012.

[3] JEDEC, "Embedded Multi-Media Card Electrical Standard (5.1), JESD84-B50.1." http://www.jedec.org/sites/default/files/docs/JESD84-B51.pdf, February 2015.

[4] JEDEC, "Universal Flash Storage (UFS), Version 2.0, JESD220B." https://www.jedec.org/sites/default/files/docs/JESD220B.pdf, September 2013.

[5] "The samsung galaxy s6 and s6 edge review." http://www.anandtech.com/show/9146/the-samsung-galaxy-s6-and-s6-edge-review/7, April 2015.

[6] JEDEC, "Universal Flash Storage Unified Memory Extension, Version 1.0, JESD220-1." https://www.jedec.org/sites/default/files/docs/JESD220-1.pdf, September 2013.

[7] K. Watanabe, K. Yoshii, N. Kondo, K. Maeda, T. Fujisawa, J. Wadatsumi, D. Miyashita, S. Kousai, Y. Unekawa, S. Fujii, T. Aoyama, T. Tamura, A. Kunimatsu, and Y. Oowaki, "19.3 66.3 KIOPS-random-read 690MB/s-sequential-read universal Flash storage device controller with unified memory extension," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pp. 330–331, IEEE, 2014.

[8] "The samsung exynos 7420 deep dive - inside a modern 14nm soc." http://www.anandtech.com/show/9330/exynos-7420-deep-dive/2, June 2015.

[9] Qualcomm Techonologies, Inc., "Snapdragon 805 processor specs." https://www.qualcomm.com/products/snapdragon/processors/805.

[10] JEDEC, "DDR4 SDRAM, JESD79-4." http://www.jedec.org/standards-documents/docs/jesd79-4, September 2012.

[11] "NVDIMM." https://en.wikipedia.org/wiki/NVDIMM.

[12] Agigatech, "Agigaram ddr3 nvdimm." http://www.agigatech.com/ddr3.php.

[13] Netlist, "Nvvault ddr3 nvdimm." http://www.netlist.com/vault-memory-storage/nvvault-ddr3.

[14] Viking Technology, "Non-volatile memory." http://www.vikingtechnology.com/nvdimm-technology.

[15] J. McFarland, "Memory channel storage (mcs) demystified." http://www.snia.org/sites/default/files/Memory%20Channel%20Storage%20Demystified%20-%203_10_14.pdf.

[16] Intel, "Chip shot: Intel unveils intel optane technology based on 3d xpoint." http://newsroom.intel.com/community/intel_newsroom/blog/2015/08/19/chip-shot-intel-unveils-intel-optane-technology-based-on-3d-xpoint, August 2015.

[17] NVM Express, Inc., "NVM Express Revision 1.2." http://nvmexpress.org/wp-content/uploads/NVM_Express_1_2_Gold_20141209.pdf, November 2014.

[18] J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, (Berkeley, CA, USA), USENIX Association, 2012.

[19] X. Jiang, D. Solihin, L. Zhao, and R. Iyer, "Architecture support for improving bulk memory copying and initialization performance," in *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, pp. 169–180, Sept 2009.

[20] Samsung Electronics, "Exynos 5250 User's Manual." http://www.samsung.com/global/business/semiconductor/file/product/Exynos_5_Dual_User_Manaul_Public_REV100-0.pdf.

[21] YIC System, "SMDKC520." http://www.yicsystem.com/products/smdk-board/smdk-c520.

[22] DENX Software Engineering, "The Universal Boot Loader." http://www.denx.de/wiki/U-Boot/WebHome.

[23] D. E. Knuth., *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Addison-Wesley, 3rd ed., 1997.

[24] M. C. Daniel P. Bovet, *Understanding the Linux Kernel*. O'Reilly, 3rd ed., 2005.

[25] "IOZone Filesystem Benchmark." http://www.iozone.org.

[26] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, disk*. Morgan Kaufmann, 2008.

[27] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems," in *35th International Symposium on Computer Architecture (ISCA 2008), June 21-25, 2008, Beijing, China*, pp. 63–74, 2008.

[28] Intel, "Intel Architecture Instruction Set Extensions Programming Reference, 319433-023." https://software.intel.com/sites/default/files/managed/07/b7/319433-023.pdf, August 2015.

[29] T. J. Ham, B. Chelepalli, N. Xue, and B. Lee, "Disintegrated control for energy-efficient and heterogeneous memory systems," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pp. 424–435, Feb 2013.

[30] Intel, "Intel QuickData Technology Software Guide for Linux." http://www.intel.com/content/dam/doc/white-paper/quickdata-technology-software-guide-for-linux-paper.pdf, May 2008.