RFSA: A Minimalistic Clustered Flash Array

***Abstract*—NAND flash is seeing increasing adoption in cloud and enterprise storage today because of its orders of magnitude lower latency ($\sim$100μs) and higher bandwidth (1-2GB/s) compared to hard disks. However, traditional methods of scaling-out storage using RAID arrays and Storage Area Networks (SAN) are ineffective for flash. Modern networking hardware (e.g. Ethernet, Fibre Channel) and software stacks add significant latency relative to the access time of flash and become bandwidth bound even with only a few flash devices. Flash performance is further degraded by the use of Flash Translation Layers (FTL) commonly found in flash drives today. The FTL performs flash management functions to hide flash characteristics, such as bad blocks, bit errors and erase-before-write, from the OS/applications. However, FTLs often operate suboptimally because they lack higher level information available to the OS/user. As a result, there has been a trend to shift flash management into higher level software stacks.**

In this paper, we propose a Reduced Flash Storage Array (RFSA) that (i) uses a controller-to-controller network to enable the scaling-out of multiple flash drives with very little overhead, and (ii) exposes a fast, error-free raw flash interface to the OS that enables higher level applications (e.g. file system, database) to efficiently manage multiple flash devices. We envision RFSA to be used within a rack cluster of servers to provide fast scalable flash storage. We show through benchmarks that RFSA can provide scalable capacity and bandwidth with negligible latency overhead, and it can exploit near theoretical max performance of the NAND chips.

I. Introduction

In addition to hard disks, NAND flash has risen to become a ubiquitous storage medium in recent years in data centers and enterprises. Flash offers a numerous benefits over traditional hard disks, including scalable bandwidth, significantly lower access latency and better power-performance [1]. However, because NAND flash has characteristics that are very different from hard disks, traditional storage architectures and system hardware software stacks are often inefficient for flash.

The deployment of flash in the data center faces two key challenges. The first problem is scalability. Because of the high performance characteristics of flash, new system bottlenecks have emerged that were previously unapparent with hard disks. For example, raw flash latency is typically around 100μs, or 100x lower than hard disks. Consequently, deploying flash in large clusters over networks (e.g. SAN or distributed file systems), using commodity hardware solutions such as Ethernet and Fibre Channel, can notably limit performance by adding latency. It has been shown that using iSCSI protocols over Ethernet adds almost 300μs of latency coming from the NIC and software network and block I/O stacks [2]. Network bandwidth is another bottleneck. A single flash drive, typically with bandwidth of 1-2GB/s, can easily saturate traditional SATA/SAS storage interface ports as well 10 Gbps Ethernet and Fibre Channel connections. Faster networking hardware, such Infiniband, is available, but comes at a prohibitive cost.

The second challenge is flash management. NAND flash is a lossy storage medium with limited program/erase cycles, bit errors, bad blocks and erase-before-write requirements. To hide these undesirable properties, manufacturers typically employ a Flash Translation Layer (FTL) within the storage device that runs management algorithms, to expose a traditional block device view to the operating system. However, FTL not only requires significant hardware resources (multi-core ARM controller with large DRAM [3]), it also negatively impacts performance. Ouyang et al. has measured that the FTL degrades flash bandwidth by as much as 49% [4]. Park and Shen [5] has shown that FTL accounts for over half of the I/O latency of the flash storage device. Furthermore, FTL in the storage device has little information from higher level applications. In a distributed storage environment, this translates to suboptimal flash management decisions being made by the FTL. There has been a push towards moving the FTL out of the storage device. EMC [6] and PureStorage [7] has built large flash arrays that globally manages multiple flash devices at the same time. REDO [8] has also shown that using a log-structured file system to manage flash can great reduce the I/O stack complexity while achieving higher performance. *However, realizing these ideas require a redesign of the storage device and flash controller that provides a raw interface to flash.*

In this paper, we introduce a scalable raw flash storage platform called RFSA to address the above issues. First, RFSA can be scaled in capacity and bandwidth, with negligible latency overhead, by connecting multiple RFSA flash boards directly to each other via inter-controller links. These links can run over relatively long distances such that each flash board may reside on a separate server. Second, the platform exposes an error-free raw flash interface to the operating system that allows the addressing of individual boards, channels, chips, blocks and pages of the device. RFSA does not use an FTL at the level of the storage device. Instead, it provides minimal support for error-free flash access and relies on higher level software to perform garbage collection, bad block management, address mapping and wear leveling. Such types of file system for flash already exists today and have shown notable performance improvements (e.g. F2FS [9], REDO [8]). This simplified I/O stack design allows RFSA to reach close to maximum theoretical performance of the NAND chips.

The key contributions of this paper are as follows:

1) We present the architecture and implementation of a RFSA storage device including a redesigned flash controller that exposes a raw flash interface. We show that by simplifying the FTL, we gain access to the full performance potential of the NAND flash chips.
2) We link together multiple RFSA boards and show that the design RFSA scales in both capacity and bandwidth at negligible latency overhead on a rack cluster of servers.

3) We show that RFSA can be used with a flash-aware file system, and we run a database benchmark to provide a glimpse of the performance gains from exposing a raw flash interface.

In the rest of the paper, we compare our design with related commercial and academic flash deployment solutions (Section II). Then we discuss RFSA's hardware architecture and the RFSA flash controller. (Section III). Next, we present the inter-controller logic that links multiple flash boards together in a scalable manner (Section IV). We show our prototype hardware platform (Section V). Finally, we present performance results for single and multiple boards (Section VI). Finally, we conclude with future works (Section VII).

II. Related Work

Commercially there are a variety of flash solutions used in the data center. Individual SATA/PCIe-based SSDs [10] are frequently used to speed up database workloads or performance critical applications. These SSDs can be pooled together to be used as a caching layer for hot data in hard disks [11]. However, management of these SSDs are done by the individual devices, which often translates into suboptimal performance. Recognizing this, there has been an influx of all flash arrays on the market today such as the PureStorage FlashArray [7] and EMC XtremeIO [6]. These solutions package a large number of flash cards into a single machine that runs a global flash management software for hundreds of terabytes of flash, while adding useful features such as deduplication and encryption. While a packaged solution is attractive from in terms of convenience and features, it greatly sacrifices performance. Peak bandwidth from these flash arrays (many TB in size) is only 3-7GB/s with milliseconds of latency. This is orders of magnitude worse than the potential performance provided by the NAND flash chips. RFSA is similar in that it also aggregates multiple flash cards together, but our solution can reach near maximum NAND chip bandwidth.

To scale out even further, a Storage Area Network (SAN) is typically used. SAN is a software-based solution that uses network fabrics (e.g. Ethernet or Fibre Channel) to connect multiple storage nodes together. However, SAN adds significant latency due to the network and software stacks. This was studied in QuickSAN [2], where measured latencies were as high as 300μs - 3x higher than typical flash latency. Combining network with storage improves performance is a proven concept. For example, QuickSAN showed that the use of an an emulated PCM device with built-in Ethernet connections can help to reduce latency after bypassing software stacks. BlueDBM [12] used inter-FPGA links to achieve scalable bandwidth using several 16GB flash boards, although the hardware used was slow and relatively old. We adopt this idea in RFSA, using ultra-low latency inter-FPGA links to connect the devices, and in addition, we take into consideration management requirements of NAND flash.

Much research has gone into the design of the Flash Translation Layer (FTL), which is commonly used in flash devices today to hide undesirable flash characteristics and to provide interoperability with existing hardware/software interfaces [13]. However, running FTL requires significant hardware resources, including gigabytes of DRAM for NAND address mapping and multiple ARM cores for management algorithms. More importantly, the FTL is often suboptimal because it does not have access to higher level OS and application information. This can result in unnecessary data movement within the flash device leading to degraded bandwidth and higher access latency. In fact, modern SSDs only reaches 41% to 51% of the theoretical write bandwidth [4]. The functionalities of the FTL can also collide with the management functions of the OS file system. For example, common log-structured file systems has its own methods of address mapping and garbage collection schemes. However, the FTL adds another layer of translation and garbage collection. This management duplication significantly affects performance and flash life time due to unnecessary data movements. This problem is exacerbated in a distributed storage environment, each flash device is unaware of other devices, and operate their own FTL independently. Previous works such as SDF [4] and REDO [8] recognize this issue and they have shown greater efficiency using higher level software stacks to perform flash management. Similarly, as previous discussed, PureStorage and EMC provide commercial solutions that globally manage multiple flash devices. In RFSA, we adopt this thin-FTL concept to provide a fast, error-free access to raw flash that gives higher level software the freedom to better manage flash.

III. RFSA Device Architecture

Although the RFSA's controller does not run flash management routines, there are At the top level, RFSA is composed of a set of flash devices directly connected to each other via high speed links. Each RFSA device can be optionally attached to an server via standard storage interfaces (i.e. PCIe). The hardware architecture consists of a flash controller redesigned with a raw interface, a flash interface router with a tag renaming engine, a DMA transfer completion buffer, and an inter-controller networking module. Figure 1 shows a logical view of the architecture. On the software side, a driver provides raw access the physical addresses of the entire flash array, presented as a *global shared-memory* among all the servers. The hardware is responsible for routing the requests and data to different flash boards. The flash board is deployed on multiple machines in the same rack - hence we call this a clustered flash array.

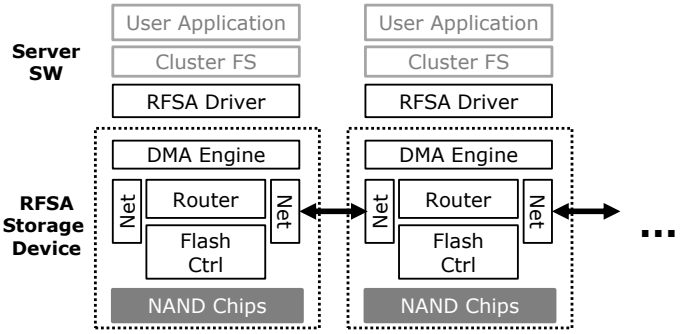


Fig. 1: RFSA architecture: RFSA is composed of a set of inter-controller connected flash devices, each of which can be optionally attached to a host server.

A. Physical Storage Device

The physical RFSA storage device follows a design that is similar to modern SSDs (Figure 2). Namely, we organize

multiple NAND flash chips (i.e. ways) into multiple individual buses (i.e. channels) to provide capacity and bandwidth scaling. Buses are controlled by an FPGA chip that implements all of the RFSA logical hardware blocks we presented earlier in Figure 1. However, the RFSA device also differs from traditional SSDs in that it also provides a high-speed serial network interface directly from the fabrics of the controller chip. These links are chip-to-chip and hence have very low latencies and high bandwidth. Implementation details of the device will be discussed in Section V.

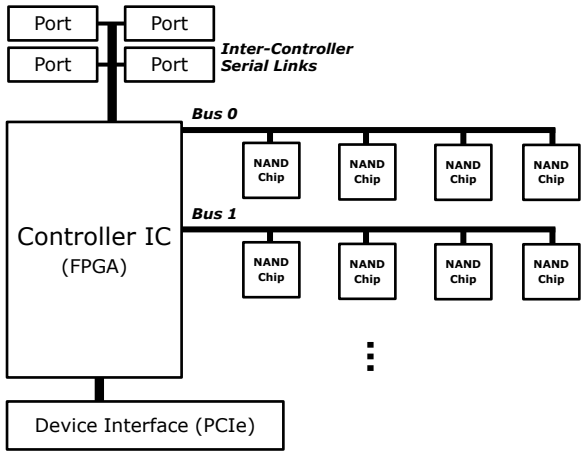


Fig. 2: RFSA physical storage device: NAND chips organized into buses with serial link coming from the controller chip.

To fully exploit the performance of this flash device, we first need a driver interface which allows users to issue hundreds of requests to the device for parallel out-of-order execution. Secondly, we need a flash controller design that maximizes bus utilization and minimizes access latency overhead. We discuss our software driver interface below, followed by the flash controller design.

B. Software Driver Interface

The RFSA device driver has two key characteristics. First, to leverage flash device parallelism, it uses a tagging scheme that labels requests and responses such that aggressive out-of-order execution of requests is possible on the flash controller. Second, unlike traditional SSDs, the device driver exposes the internal organization of the flash array. As we have previously discussed this opens up opportunities for higher level applications to more efficiently manage the flash device.

To use the API, the user should create a multi-entry page buffer, where each entry is a flash page buffer (typically 8KB) and is associated with a unique tag. This buffer will be used as a completion buffer for page reads, and as a write buffer to temporarily hold write data until it can be transferred. We provide only page aligned accesses to the flash array for higher bandwidth since it is the minimum unit of transfer of the flash chips. To issue commands to the flash array, the user first obtains a free tag, and then calls a send command function, passing along (1) the tag, (2) the operation (read page, write page, erase block), (3) the target physical address (board, bus, chip, block, page) and (4) a pointer to the page buffer associated with the tag. When the operation completes, an interrupt is raised with the tag of the completed command, and a user-defined callback function will execute. Then the command is considered done, and the tag as well as page buffer space can be recycled.

This device interface gives the user complete control over the management of the flash array. For example, file systems such as REDO [8] can choose optimal segment sizes and mappings that matches the organization of the flash array to maximize bandwidth. Databases storage engines may explicitly manage its data chunks to decide when to perform garbage collection. We note that this interface also provides a global shared-memory view of all connected flash devices. Any server may access any of the boards by specifying the board ID as part of the address. Concurrency control is left up to higher level software such as a a database or a clustered file system (e.g. GFS, NFS).

C. A Simplified Flash Controller

Unlike modern SSDs that use multi-core ARM CPUs and gigabytes of DRAM, RFSA's flash controller aims to use much less hardware area to achieve near theoretical max NAND performance. This is accomplished by providing only basic flash management routines at the device level, and relying on higher level applications to more efficiently manage flash. In the controller, we chose to implement basic FTL tasks that are naturally more amenable to hardware, including bit error correction (ECC), bus/chip scheduling, and NAND I/O protocol communication (i.e. ONFI). The controller architecture is shown in Figure 3. We discuss some of its details below.

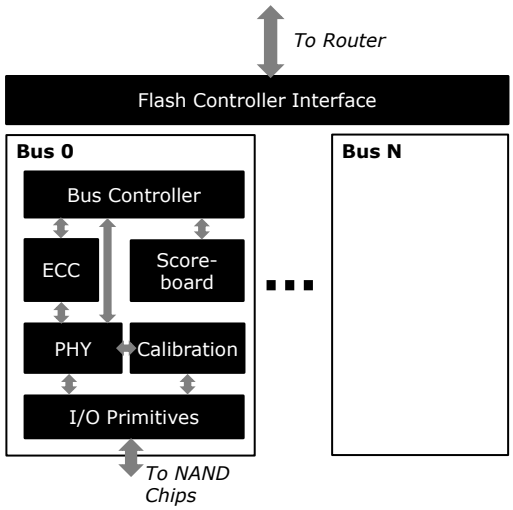


Fig. 3: RFSA flash controller: Each bus has a separate controller that schedules commands, corrects bit errors and provides I/O communication with NAND chips. The controller has a tagged command/data interface.

1) Bus Controller and Scoreboard: Since flash buses are independent from each other, we create a separate bus controller for each. It is the the role of the bus controller to maximize bus utilization and hide flash latency. Since a single flash operation takes hundreds of microseconds to several milliseconds to complete, their execution can be overlapped. For example, while one chip is performing a read, control of the bus can be given to another chip to stream in data

to be written. The bus controller schedules these operations to maximize bandwidth. Incoming requests are distributed to each bus controller, which are subsequently inserted into the scoreboard and distributed to individual chip request FIFOs to be executed in order (Figure 4). For each chip, we keep a busy timer counter and a stage register. The current scheduler works in a priority round robin fashion. It rotates to picks the first request that has the highest priority among all the chips and and enqueues it for execution. It then sets the busy timer to a predefined *estimate* of when the stage would be complete (NAND latencies are variable). Finally it updates the stage register to indicate which stage of the request it is currently on. It then moves to the next chip to schedule the next request. When the busy timer expires, the scheduler queues a poll request to check if the chip has finished its job. If it has, the next stage is scheduled, otherwise the busy timer is set once again to further wait a predefined amount of time. When the request is complete, it is removed from the chip queue. We set the scheduler to prioritize small short command/address bursts on the bus over long data transfers to keep chips as busy as possible.

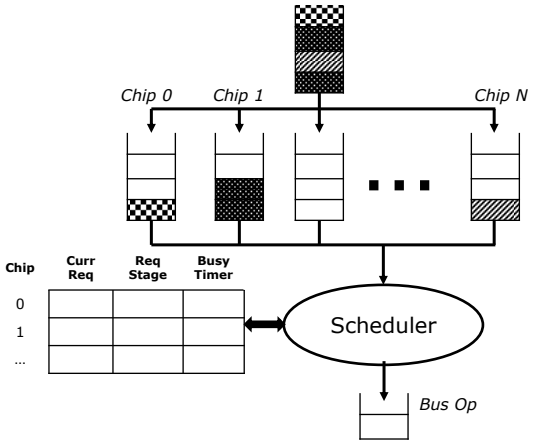


Fig. 4: Bus scheduling and scoreboard: Incoming requests are distributed into separate chip queues. Scheduler enqueues each stage of the request onto the bus using a priority round-robin scheme.

2) Error Correction: Bit errors are common in NAND flash chips, and error rate is influenced by many factors such as the number of P/E cycles, the value of the data and environment variables. Exposing bit errors to the software is burdensome and unnecessary, hence we implement the ECC portion of the FTL inside our flash controller to ensure that our software interfaces are bit-error free. Commercial SSDs today use multiple levels of DSP techniques to correct errors. BCH codes or LDPC codes are commonly used [14]. For simplicity, we use Reed-Solomon codes since it consumes less hardware resources. In practice, we found that the $RS(255, 243)$ configuration with 12B parity (4.7%) meets the minimum ECC requirements of the MLC chips we are using. We concede that a disadvantage of using Reed-Solomon is its variable decoder latency that could cause stalls on the bus. On a production system, we would use a more advanced ECC algorithm, such as LDPC.

3) Controller Hardware Interface: The flash controller provides a tagged data/command interface in hardware. Similar to the software interface, commands issued to the controller are uniquely tagged. Read and write data transfers occurs in tagged bursts (128-bit in our design) corresponding to the command tag. This scheme is required because data from all buses are aggressively packed and sent out at the interface for maximum bandwidth. As a result, data from different buses may be interleaved, and data from the same bus may appear out of order with respect to the commands based on when it was scheduled. We use a set of completion buffers to transfer the data over to host server.

D. DMA Engine and Completion Buffer

We use two modes of data transfer between the server and flash boards: (i) RPC-style memory-mapped I/O interface, and (ii) DMA over PCIe to/from the server memory. Generally, commands and acknowledgments are transferred using the low-latency RPC interface, and page data is transferred via high bandwidth DMA.

DMA should be performed on large chunks of data (e.g. 128 B) for high bandwidth. We distribute out-of-order data bursts from the controller to a per-tag completion buffer to queue up sufficient bursts for a DMA transfer. To ensure that our hardware design meets FPGA timing, we divide up the completion buffer into smaller memory chunks, where each chunk is associated with a fixed portion of the tag space. The DMA engine then performs the data transfer to the address offset in host memory given by the software read request. An acknowledgment is sent to the software to trigger an interrupt upon completion of a full page transfer.

For writes, the controller receives a request with a pointer to the location of write data in host memory. However, because there is limited buffer space in the flash controller, data transfer to the controller cannot begin until the command has been scheduled and the controller has reserved space for the data. Upon receiving this request, the DMA read engine relays the entire page to the controller.

IV. Multi-Device Linking

RFSA can be scaled in both capacity and internal bandwidth by connecting multiple flash boards together via inter-controller links. This is done in a transparent way to the software, which simply sees an addition address dimension (i.e. board ID). Since all of the routing is done by the flash device, flash boards may be linked with or without attaching to a server. In the former case, RFSA appears as a shared-memory storage device to each server, similar to a SAN. In the latter case, RFSA appears as a RAID array of flash devices to a single server.

A. Shared Controller Management

To provide shared access to flash, we introduce a flash interface router (Figure 1) that splits/merges remote and local data/commands onto both the flash controller and the DMA engine. To ensure fair resource sharing, we use rotating priority arbiters for all datapaths.

Since multiple servers may issue commands to the same controller, there could be tag collisions if each server is only aware of their own set of unique tags. Using global unique tags

can resolve the problem, but this requires cooperation from the software and refactoring of tag space when flash devices are added or removed. Instead, we apply a layer of indirection by renaming *host* tags into *controller* tags. Upon receiving a command, the router obtains a free controller tag from the free queue and stores the original host tag with the source server ID of the command in a look-up table. Responses from the controller that are labeled with controller tags will index into the table to find the original host tag and command source, which are repackaged with the response to be routed back to its source. This look-up datapath is fully pipelined.

B. Controller-to-Controller Network

Since RFSA is aimed at rack level deployment where the servers are separated over relatively short distances, we assume a lossless network between the flash devices. We use a linear array network topology which runs vertically up and down the rack (average $2n/3$ hops). This simplifies routing of packets and allows us to connect the devices using short, equidistant cables. We leverage direct chip-to-chip multigigabit transceivers (MGT) to provide massive bandwidth and extremely low latency. We use a packet switched router on top of MGT with an arbiter that supports independent virtual channels and token-based end-to-end flow control on each virtual channel. We instantiate a virtual channel for each datapath of the flash interface (read data, write data, command, ack etc.) to connect multiple controllers together.

C. Scalability

We remark that the hardware design of RFSA allows it to scale up with little increase in hardware resources. Address width will become wider with more boards, but this does not translate to additional hardware structures. This theoretically allows us to chain hundreds of flash boards together for capacity scalability. In terms of performance, the latency of MGT links are so low (we measured 0.5μs/hop), that even a hundred hops still translates to less latency than flash access time. This easily surpasses modern SAN-based storage architectures. Bandwidth will be affected with many boards as the network can become congested and I/O interfaces to the servers (e.g. PCIe) can become rate limiting. However, we argue that the *aggregated* bandwidth of the entire flash array scales, since all of the bandwidth of all flash devices is still made available to the user. This is unlike some commercial flash array appliances such as PureStorage, where only a fraction of the total potential bandwidth of the flash chips is exposed via Fibre Channel or Ethernet ports. We can further increase cross-sectional bandwidth of the network by choosing a different topology, such as a star or a mesh, provided that the physical links can remain lossless.

V. RFSA Platform Hardware

We implement RFSA using a Xilinx VC707 FPGA development board [15], which has a Virtex-7 FPGA, x8 PCIe Gen 2.0, and 8x10Gbps MGT serial links. This is the primary carrier FPGA board. We designed a custom flash board to attach to the VC707 via the FPGA Mezzanine Card interface (FMC) [16]. Up to two flash boards may be attached to a carrier board, but we use a single board for experiments in this paper.

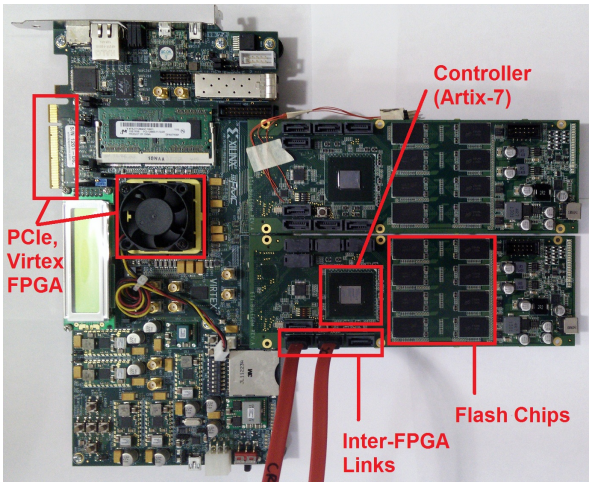


Fig. 5: RFSA prototype platform hardware

Parallel flash channels can provide linear bandwidth scaling, but is limited by the number of I/O pins, hardware resources on the controller (i.e. FPGA) and PCB routing. Chips on the same bus also provide parallelism by hiding access latency, but this is only effective if the bus bandwidth has not saturated. Based on these constraints, we chose an 8-channel, 8-way design using 256Gbits Micron MLC NAND chips for a total of 512GB of flash storage per board. Thus the maximum aggregated bus bandwidth is 1.6 GB/s. Typical latencies for the chip are 75μs for reads and 1300μs for writes.

We integrate a smaller Xilinx Artix-7 FPGA on the flash board that serves as the flash controller. This controller connects to the primary VC707 board via 4x 6.6Gbps MGT GTP connections over the FMC port. In addition, the flash board pins out 4x 10Gbps high speed serial connections (MGT GTX) from the VC707 board in the form of SATA ports. We use cross-over SATA cables as our inter-FPGA links to connect multiple boards together on a rack. Figure 5 is a photo of the flash board.

We used Ubuntu 12.04 with Linux 3.13.0 kernel for our software implementation and benchmarks. We used the Connectal [17] software-hardware codesign library to provide RPC-style request/responses and DMA over PCIe. Connectal allows us to expose our raw flash interface in both the kernel and userspace. We used a modified version of the Reed-Solomon ECC decoder from Agarwal et al. [18] in our controller.

VI. Evaluation

In this section, we measure and evaluate (1) single device performance of RFSA, (2) multi-device, multi-host performance and (3) application compatibility and performance. We show that our RFSA device can reach peak bandwidth of 1.2 GB/s (75% of theoretical bus bandwidth) with a mere 15μs of I/O latency overhead. We show that chaining multiple devices together in a distributed fashion adds trivial amount of latency, while effectively increasing the aggregated bandwidth. We demonstrate that RFSA is compatible with and improves the performance existing software layers by running a flash-aware file system and a database benchmark on top of RFSA.

A. Resource Usage

A breakdown of the resource usage is shown in Table I. Note that only major components are shown. On the Artix-7 FPGA, the flash controller uses about half of the resources for all 8 channels: 23% registers, 56% LUTs and 50% BRAM, and 46% of I/O pins. On the Virtex-7 FPGA, PCIe with DMA engine and the inter-FPGA network consumes 14% registers, 25% LUTS and 20% BRAM. Overall, the design is relatively small. A large amount BRAM is required for network buffering and flash burst buffering. Most LUTs and registers are used for pipelining the design and muxing/demuxing datapath. ECC uses the most area on the flash controller in terms of logic.

Module Name	# Inst	LUTs	Registers	BRAM
Bus Controller	8	7131	4870	21
→ *ECC Decoder*	2	1790	1233	2
→ *Scoreboard*	1	1149	780	0
→ *PHY*	1	1635	607	0
→ *ECC Encoder*	2	565	222	0
FMC SERDES	1	3061	3463	13
Artix-7 Total		**75225 (56%)**	**62801 (23%)**	**181 (50%)**
PCIe and DMA Engine	1	56302	62277	116
Serial Network	1	20448	25676	94
Virtex-7 Total		**76750 (25%)**	**87953 (14%)**	**210 (20%)**

TABLE I: Artix-7 and Virtex-7 Resource Breakdown

B. Single Local RFSA Board Performance

1) Page Access Latency: We define read page access latency as the time it takes to receive the data for an entire 8KB page after issuing a single read page request (queue depth of 1). Write access latency is similarly defined as the time between issuing the write request, transferring data and receiving an acknowledgment from the controller that the operation is complete. The latency breakdowns are shown in Table II.

	Read Latency (μs)	Write Latency (μs)
PHY Commands	1	1
NAND	69	418 (variable)
ECC	4	0.1
Data Transfer	43	43
PCIe/Software	11	14
Total	128	476 (variable)

TABLE II: Local Read and write access latencies

The NAND read latency of 69μs is within the expected range based on the Micron datasheet (<75μs). With 8K pages (8600B including ECC) transferring at 8-bit wide 100MHz DDR bus speed, this page transfer process incur 43μs of latency. We note that these latencies are intrinsic properties of the NAND chip that are not related to the design of RFSA. The Reed-Solomon ECC decoder latency varies with the number of bit errors. On average, we observe 2.18 bit errors per page, which translates into only 4μs of latency. This is expected to increase as the chips age. Latency of carrier to flash board communication over FMC and command scheduling/issue latencies are both insignificant as there are no significant stalls in the processing pipeline. PCIe flash request/responses, DMA and driver software incurs an additional 11μs of latency. This is in the range of industry storage interface standards such as SCSI/SAS (6μs) and NVMe (2.8μs) [19]. In total, the read access latency to a single RFSA device is 128μs, with a mere 11% overhead from the RFSA controller.

The overhead introduced for writes is similarly low. However, the total write latency must be taken with a grain of salt since NAND programming latency is highly variable (up to 2.1ms for our MLC chips). For example, programming multiple bits that share the same MLC NAND cell requires significantly more time than programming a single bit into the cell. Disregarding these intrinsic NAND latencies, we observe that by exposing a raw flash interface, without additional software FTL layers at the device level, we can access flash storage at close to native flash latency. In comparison, our measurements show that simply running the FTL incurs up to 30μs of additional latency. This is not including further inefficiencies in the management algorithm.

2) Bandwidth vs. Transfer Size: For this benchmark, we partition the address space such that data is striped across channels and chips to leverage full parallelism of the flash device. This is the typical way for modern FTLs to map its address space. We measure the bandwidth from issuing requests for various transfer sizes, starting from single 8KB pages to 16MB chunks. We measure both sequential and random accesses. The former setting has all pages of the chunk in a linear address space (i.e. perfectly striped), and the latter has chunks that contain pages that are spread out randomly in the address space. Results are shown in Figure 6. We note that random writes are not measured because of the erase-before-write property of NAND flash that prevents us from randomly writing data. Typically the FTL performs garbage collection or address remapping to handle random writes. In RFSA, we leave these functionalities to higher level software.

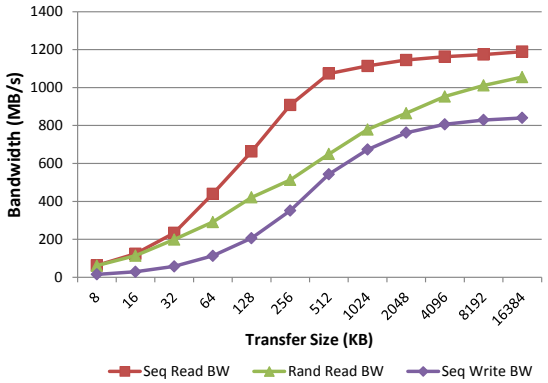


Fig. 6: Bandwidth vs. transfer size of the RFSA device

For all curves, bandwidth grows quickly as more channels are used at the beginning of the graph. The growth slows when we move towards chip parallelism as the bus becomes busier and eventually saturates. Random access performance is slightly worse due to address collisions that would reduce parallelism. This is, of course, orders of magnitude faster than hard disks. Peak sequential performance is a respectable 1.2GB/s, which is 75% of the maximum bus bandwidth of the device. We note that 5% of the overhead inherently arises from transferring ECC parity bits. Additional overhead comes from commands, addresses and chip status polling on the bus, as well as some imperfections in scheduling. We conclude that overall our raw interface design is very efficient, and is able to fully expose the raw bandwidth of the flash array to the software driver.

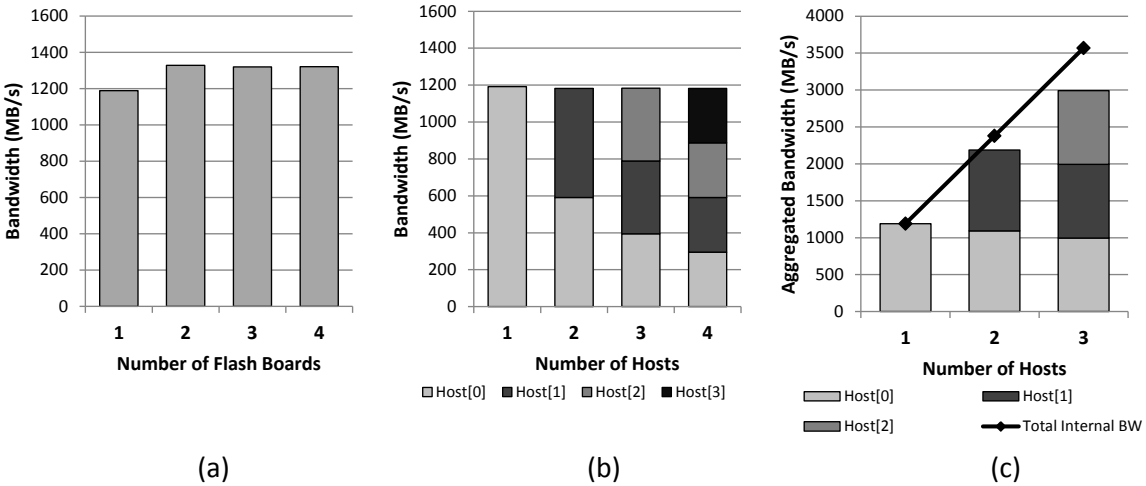


Fig. 7: Multi-device bandwidth. (a) single host, multiple flash boards. (b) multiple hosts, single flash board. (c) multiple hosts, multiple flash boards.

C. Multi-Device Performance

We chain together 4 RFSA flash boards attached to 4 separate host servers to measure multi-device performance.

1) Access Latency: Figure 8 show the flash page read access latency over multiple hops of RFSA devices. Because of direct chip-to-chip links, the inter-controller network latency is virtually non-existent. In fact, latency variations (shown by error bars) in software and NAND chips far exceed measurable network latency. Our hardware counters indicate that each hop is a trivial 0.5μs. Because accessing local PCIe attached flash and remote flash boards are equally fast, RFSA's global shared-memory interface appears as local storage, even though it is physically distributed among multiple machines. This is a vast improvement over SAN-based storage architectures.

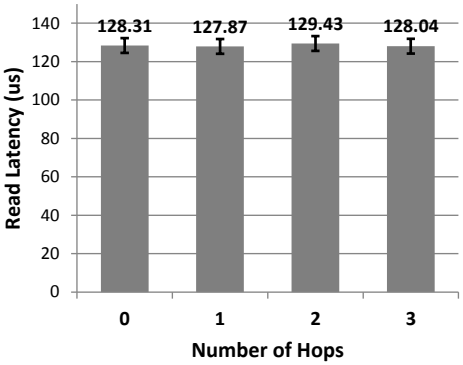


Fig. 8: Multi-hop page access latency

2) Bandwidth: We measure RFSA's bandwidth under the following scenarios: (1) single host accessing multiple connected RFSA devices (Figure 7a), (2) multiple hosts accessing the same device (Figure 7b), and (3) multiple hosts accessing multiple devices (Figure 7c). All accesses are random reads of 16MB chunks.

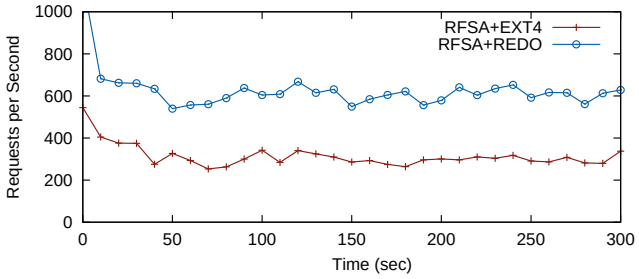
The first scenario is similar to a RAID-0 arrangement. We get some speed-up (from 1.2 GB/s to 1.32 GB/s) by accessing multiple boards in parallel, but ultimately we are bottlenecked by our x8 PCIe Gen 1.0 implementation. We are in the process of upgrading to Gen 2.0, which would double the interface bandwidth. However, because the total aggregated bandwidth from multiple RFSA flash boards is extremely high, we do not expect any single server use up all of the bandwidth. RFSA is more powerful than RAID arrays in that it makes the aggregated bandwidth available to *multiple* compute servers while maintaining the low latencies of direct attached storage.

The second scenario examines the behavior of RFSA when there is resource contention for the same flash device. The graph shows that the RFSA controller and the network routers can very fairly distribute the bandwidth to each host, while maintaining peak overall bandwidth. This is important when hosts are performing parallel computations.

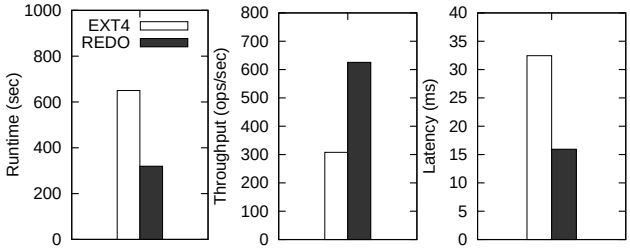
The last graph shows the aggregated bandwidth scalability of the global shared-memory flash store, with multiple servers randomly accessing the entire address space. The line in the graph shows the total maximum internal bandwidth provided by the flash devices (a simple multiple of the bandwidth of a single device). The bars in the graph are the achieved aggregated throughput from the hosts' perspective. We reach 92% of the maximum potential scaling with 2 hosts and 84% with 3 hosts for a total of 3 GB/s. We expect RFSA to continue scaling well for up to a cluster of 10-20 servers on a single rack, given our high bandwidth 80Gbps serial network.

D. Application Performance

Finally, we run a flash-aware file system called REDO [8] on top of RFSA to illustrate its compatibility with OS and user applications as well as to show the advantage of exposing a raw flash interface. REDO is a log-structured file system that contains built-in flash management functionalities. By removing redundancies that arise from separately using FTL and traditional file systems, REDO can achieve higher performance using less hardware resources. To validate this on our RFSA platform, we run the popular Yahoo Cloud Serving Benchmark (YCSB) [20] with MySQL+InnoDB at default settings. YCSB was configured to perform 200,000 updates to 750,000 records in the database. We compare two I/O stack configurations: (1)



(a) Requests per second over time



(b) Runtime (c) Throughput (d) Latency

Fig. 9: YCSB benchmark results comparing (1) RFSA + REDO and (2) RFSA + page FTL + EXT4

RFSA+REDO file system and (2) RFSA + host page-level FTL + EXT4 file system. The latter configuration emulates a traditional SSD I/O architecture. Measurements are shown in Figure 9.

We see that RFSA+REDO doubles the performance of RFSA+FTL+EXT4 in both throughput and latency. This is gain primarily comes from reduced number of I/O operations that REDO performs for the same workload. By merging file system and FTL functions, REDO can cut down on redundant and unnecessary I/Os in garbage collection, while maximizing the parallelism of the flash device. REDO is one of many examples of OS-level and user-level software that can take advantage of the raw flash interface provided by RFSA.

VII. Conclusion and Future Work

We have presented RFSA, a clustered flash array that provides scalable and distributed raw flash storage to rack servers. We proposed a simplified flash controller for RFSA that exposes a low-overhead error-free interface to flash. We have shown that such an interface can be effectively combined with a flash-aware file system to double the performance of some database workloads. RFSA scales by using chip-to-chip serial links to directly connect flash controllers to each other. Latency overhead of this network is negligible and aggregated bandwidth of the system scales close to linearly.

On the hardware side, we are examining new network topologies that would increase cross-sectional bandwidth to allow the system to scale out to greater number of devices and even beyond a rack. On the software side, we are looking into supporting distributed flash management at the file system level, potentially using the serial network to pass metadata information between nodes.

References

[1] P. Desnoyers, "Empirical evaluation of nand flash memory performance," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 1, pp. 50–54, Mar. 2010.

[2] A. M. Caulfield and S. Swanson, "Quicksan: a storage area network for fast, distributed, solid state disks," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 464–474.

[3] "Samsung SSD 840 EVO Datasheet Rev. 1.1," http://www.samsung.com/global/business/semiconductor/minisite/SSD/downloads/document/Samsung_SSD_840_EVO_Data_Sheet_rev_1_1.pdf, 2013, accessed: Apr. 2015.

[4] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang, "Sdf: Software-defined flash for web-scale internet storage systems," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 471–484.

[5] S. Park and K. Shen, "Fios: a fair, efficient flash i/o scheduler." in *FAST*, 2012, p. 13.

[6] "EMC XtremeIO," http://www.xtremio.com/, accessed: Apr. 2015.

[7] "PureStorage FlashArray," http://www.purestorage.com/flash-array, accessed: Aug. 2014.

[8] S. Lee, J. Kim, and A. Mithal, "Refactored design of i/o architecture for flash storage," *Computer Architecture Letters*, vol. PP, no. 99, pp. 1–1, 2014.

[9] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2fs: A new file system for flash storage," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA: USENIX Association, 2015, pp. 273–286. [Online]. Available: https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee

[10] "FusionIO," http://www.fusionio.com, accessed: Aug. 2014.

[11] "NetApp Flash Pool," http://www.netapp.com/us/products/platform-os/flashpool.aspx, accessed: Apr. 2015.

[12] S.-W. Jun, M. Liu, K. E. Fleming, and Arvind, "Scalable multi-access flash store for big data analytics," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA '14. New York, NY, USA: ACM, 2014, pp. 55–64.

[13] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "System software for flash memory: A survey," in *Proceedings of the 2006 International Conference on Embedded and Ubiquitous Computing*, ser. EUC'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 394–404.

[14] E. Sharon, I. Alrod, and A. Klein, "ECC/DSP System Architecture for Enabling Reliability Scaling in Sub-20nm NAND," http://www.bswd.com/FMS13/FMS13-Klein-Alrod.pdf, August 2013, accessed: Aug. 2014.

[15] "Xilinx Virtex-7 FPGA VC707 Evaluation Kit," http://www.xilinx.com/products/boards-and-kits/EK-V7-VC707-G.htm, accessed: Aug. 2014.

[16] "FPGA Mezzanine Card Standard," http://www.xilinx.com/support/documentation/white_papers/wp315.pdf, August 2009, accessed: Aug. 2014.

[17] M. King, J. Hicks, and J. Ankcorn, "Software-driven hardware development," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: ACM, 2015, pp. 13–22.

[18] A. Agarwal, M. C. Ng, and Arvind, "A comparative evaluation of high-level hardware synthesis using reed-solomon decoder," *Embedded Systems Letters, IEEE*, vol. 2, no. 3, pp. 72–76, Sept 2010.

[19] D. Cobb and A. Huffman, "NVM Express and the PCI Express SSD Revolution," http://www.nvmexpress.org/wp-content/uploads/2013/04/IDF-2012-NVM-Express-and-the-PCI-Express-SSD-Revolution.pdf, May 2012.

[20] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154.