minFlash: A Minimalistic Clustered Flash Array

Ming Liu, Sang-Woo Jun, Sungjin Lee, Jamey Hicks and Arvind
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology, Cambridge, Massachusetts 02139
{ml, wjun, chamdoo, jamey, arvind}@csail.mit.edu

***Abstract*—NAND flash is seeing increasing adoption in the data center because of its orders of magnitude lower latency and higher bandwidth compared to hard disks. However, flash performance is often degraded by (i) inefficient storage I/O stack that hides flash characteristics under Flash Translation Layer (FTL), and (ii) long latency network protocols for distributed storage.**

In this paper, we propose a minimalistic clustered flash array (minFlash). First, minFlash exposes a simple, stable, error-free, shared-memory flash interface that enables the host to perform cross-layer flash management optimizations in file systems, databases and other user applications. Second, minFlash uses a controller-to-controller network to connect multiple flash drives with very little overhead. We envision minFlash to be used within a rack cluster of servers to provide fast scalable distributed flash storage. We show through benchmarks that minFlash can access both local and remote flash devices with negligible latency overhead, and it can expose near theoretical max performance of the NAND chips in a distributed setting.

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 significantly lower access latency, higher bandwidth, and better power-performance than hard disks [1]. However, traditional storage architectures and system hardware software stacks are often inefficient for NAND flash because it has very different characteristics than hard disks.

Figure 1(a) shows the current storage I/O stack for a modern SSD. The SSD presents a generic storage device interface to the host system (e.g. SATA AHCI or NVMHCI) with read/write operations to some device address. Within the SSD, vendors use a Flash Translation Layer (FTL) to hide all flash characteristics in order to maintain compatibility with existing I/O stacks. The FTL performs flash management tasks including wear-leveling, bad block management, address mapping and garbage collection. FTL operates in isolation; Higher level software such as file system, block device driver and user application cannot access or control the policies set by FTL. This causes inefficiencies in flash management. For example, modern file systems (e.g. EXT4) can incur random writes that would cause excessive garbage collection (GC) in the FTL. However, by choosing a log-structured file system and modifying its built-in GC algorithm for flash, we can reduce I/O traffic and remove GC from the FTL completely [2]. Another example is database systems, which often organizes data based on physical storage device characteristics and the schema of the tables. However, the FTL has another layer of data mapping policy that can result in unpredictable database performance. NoFTL [3] integrated FTL into a database to show significant speedup. Similarly, SDF [4] exposed parallel flash channels of a single board as separate devices to the OS. While SDF still runs FTL inside the flash device, it gives software control of scheduling and data layout of the channels to attain better performance. Other related work refactored the FTL for file systems (F2FS [5], REDO [2]) and atomic primitives [6]. These prior works suggest that higher level software can often manage flash better than a in-device FTL. We simply need a revised flash host interface to enable this.

Storage Area Networks (SAN) is a common approach used to consolidate storage in data centers. SAN uses commodity switched network fabrics (e.g. Ethernet, Fibre Channel) with SCSI protocols (e.g. FCP, iSCSI) to connect together multiple storage devices. The host is presented with a block device view over the SAN network. Figure 1(a) highlights the stacks for iSCSI over Ethernet in orange. Hosts run shared-disk file systems (e.g. GFS2 [7]) for consistent access to SAN storage. However, the SAN architecture can add 90μs to 300μs of latency due to the network and software stacks [8]. Since raw flash latency is around 100μs (100x lower than hard disks), this represents a 90%-300% overhead to access remote flash. An alternative to SAN is an all flash array [9] [10], which packages a large number of flash cards into a single storage server that runs a global flash management software. While convenient and scalable in capacity, they greatly sacrifice performance, operating at only 3-7GB/s with milliseconds of latency – a fraction of the performance potential of the raw flash chips. Today, large web companies (e.g. Google, Amazon) use local distributed storage for flash, where each compute server has direct-attached flash devices [11]. Software (e.g. HDFS [12]) manages the storage pool. While local access is fast, remote access over the network is slow. Prior work such as QuickSAN [8] and NASD [13] have combined network with storage to reduce overhead. As we will show, minFlash's architecture further improves on storage latency.

We have built a scalable flash storage platform called minFlash that reduces both I/O stack and network overheads to fully expose flash performance. minFlash connects multiple custom flash devices using an *inter-controller network* and exposes an *error-free, distributed shared memory, native* flash host interface. Using this flash interface, we expose flash characteristics to allow cross-layer optimization and reshuffling of the I/O stack while gaining close to theoretical peak performance of the flash chips. We keep basic functionalities in the controller, namely ECC and bus/chip scheduling, to provide low-overhead and error free access to flash. Higher

978-3-9815370-6-2/DATE16/

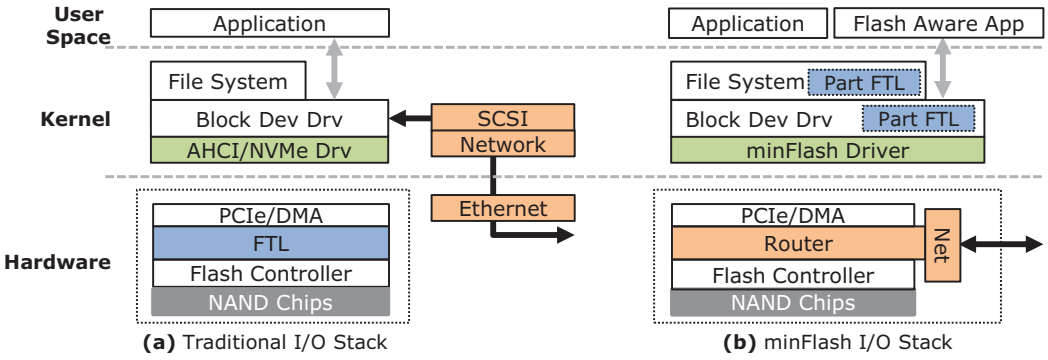


Fig. 1: A comparison of traditional network and storage stacks with minFlash's design.

level FTL management algorithms, such as bad block management, address mapping and garbage collection are left to the OS/application software. Using inter-controller links, minFlash scales in capacity and bandwidth with negligible latency. minFlash device controllers handles routing of commands and data directly without interference from the host, therefore entirely bypassing traditional network and I/O stacks. We show that *minFlash has the low-latency characteristics of a direct-attached storage as well as the scalability of a distributed storage architecture.*

The key contributions of this paper are as follows:

1) A minFlash architecture and its hardware implementation including a new host interface to access the array.
2) A scalable storage system where multiple minFlash boards are connected using a rack-level inter-controller network which has negligible latency.
3) Demonstration of the performance gains from reshuffling the I/O stack using a flash-aware file system on minFlash.

In the rest of the paper, we introduce minFlash's architecture (Section II). Then, we present the inter-controller logic to scale up minFlash (Section III). We show our prototype hardware platform (Section IV) with measured performance results (Section V). Finally, we conclude with future works (Section VI).

II. minFlash Architecture

Figure 1(b) shows minFlash's storage I/O stacks and network stacks. In minFlash, all of flash management functionalities are built into host software stacks (highlighted blue). The FTL is not just *moved* to the host, it is also *simplified and reduced* by eliminating redundant operations and applying optimizations using rich system level information. For example, one property of flash is that reads/writes occur in 8KB-16KB pages, but erases can only be done in blocks of many pages. Moreover, a page cannot be overwritten – it must be erased first. The FTL handles this using a mapping table of logical to physical address within the SSD. In minFlash, we move this mapping table to the file system, or even higher to a flash-aware user application (e.g. database). Similarly, we can combine a log-structured file system's garbage collection (GC) algorithm with a traditional FTL's GC algorithm to eliminate an extra layer. Wear-leveling and bad block management are built into the lower level block device driver so they can be shared by higher layers in the software.

To enable these flash management optimizations, minFlash's device interface provides a raw, error-free and shared-memory view of the storage cluster. Internal layout of the flash array is made accessible by addressing the flash device IDs, channels (buses), ways (chips), blocks and pages. Only low level bit-errors and chip/bus scheduling is hidden to be handled by the minFlash hardware and driver. Upper layer software decides how to map data and manage flash.

In addition, minFlash eliminates most of the network software overheads by moving the storage network to the flash device. While we can connect multiple flash devices using commodity networking hardware such as Ethernet or Fibre Channel, their latencies add at least 90% overhead to flash. Instead, we chose to use high-bandwidth, low-latency inter-controller chip-to-chip serial links. These links operate over shorter distances but are sufficient for a rack server. Each device uses a thin network router next to the flash controller to direct flash request/responses. This provides very low-latency access to remote storage. Each host accesses minFlash as a distributed shared memory, where any host may access any device by simply specifying the device ID as part of the address.

A. Host Interface and Flash Controller

The minFlash device driver uses physical flash addresses to expose organization of the entire distributed flash array. Remote devices are accessed using an extra address dimension (i.e. device ID). The inter-controller network will transparently route the request and responses. Requests and responses are tagged so that aggressive out-of-order execution of flash commands is possible by the flash controller to reach full bandwidth. The use of the interface is restricted by flash properties (e.g. erase-before-write). This is shown below:

- **ReadPage(tag, device, bus, chip, block, page)**: Reads an 8KB flash page on any device.
- **WritePage(tag, device, bus, chip, block, page)**: Writes an 8KB flash page on any device. Pages must be erased before written, and writes must be sequential within a block.
- **EraseBlock(tag, device, bus, chip, block)**: Erases a 256 page block on any device.

- **Ack(tag, status)**: Acknowledges completion of a request of a certain tag. Status indicates OK, bad blocks on erase or uncorrectable error on reads.

To use the API, we create a multi-entry page buffer in host DRAM, where each entry is a flash page buffer (typically 8KB) and is associated with a unique tag. This is a completion buffer for page reads, and temporary transfer buffer for page writes. The user then obtains a free tag and sends that to the device along with the operation (read, write, erase) and physical flash address. Data is then transferred via DMA to/from the buffer. Upon completion, an interrupt is raised and a callback handler recycles the tag/buffer. As we will show, the overhead of this interface is very low.

On the hardware side, the minFlash device adopts a simplified flash controller that implements a basic set of NAND control tasks including bit error correction (ECC), bus/chip scheduling, and NAND I/O protocol communication (Figure 2). Incoming flash requests are distributed by bus address, and the bus controller uses a scoreboard to schedule parallel operations onto the flash chips for maximum bandwidth. The scheduler works in a priority round robin fashion, rotating to select the first request that has the highest priority among all the chips and and enqueues it for execution. We prioritize short command/address bursts on the bus over long data transfers, and older requests over newer ones. For ECC, we use $RS(255, 243)$ Reed-Solomon codes, which has variable decoding latency in contrast to BCH or LPDC codes used in modern SSDs [14]. However, it was chosen for its simplicity.

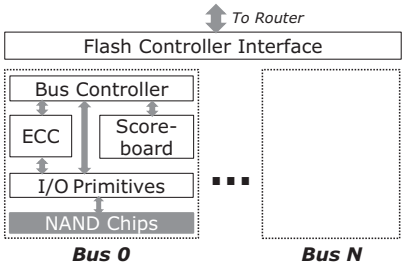


Fig. 2: minFlash flash controller

III. Multi-Device Linking

minFlash can be scaled up in both capacity and bandwidth by connecting multiple devices together via inter-controller links in a peer-to-peer fashion. This is transparent to software, which simply sees an additional address dimension (i.e. device ID). Since routing is localized to the minFlash flash device, minFlash boards may be linked with or without attaching to a host server. In the former case, minFlash appears as a distributed shared-memory storage device to each host. In the latter case, minFlash appears as a RAID array of flash devices to a single host.

A. Shared Controller Management

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

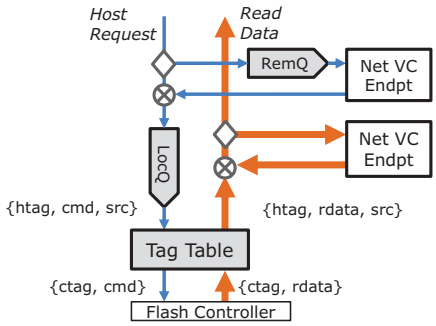


Fig. 3: Flash interface router with tag renaming for read datapath

Since each host issues requests to minFlash using their own set of tags, there could be a collision of tags if multiple hosts issue requests to the same controller with the same tag. Tags must be unique for the responses to be matched with the request. We resolve this by a layer of indirection, renaming host tags (*htag*) into controller tags (*ctag*). Upon receiving a request, the router obtains a free controller tag from the free queue and stores the original host tag with the source server ID of the request 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 request source, which are repackaged with the response to be routed back to its source. The read datapath of the router is shown in Figure 3.

Host request queue depths are equal to the number of host tags, which is equivalent to the number of requests that must be in flight to keep the PCIe bandwidth saturated. Similarly, controller request queue depth is matched with the number of requests to use full bandwidth of the flash device.

B. Controller-to-Controller Network

We envision minFlash to be used for rack level flash deployment where the servers are separated over relatively short distances. Therefore, a lossless network is assumed between the flash devices. Currently, 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 cables. The physical network is provided by controller chip-to-chip multigigabit transceivers (MGT). We use a deterministic 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[15]. We instantiate a virtual channel for each datapath of the flash interface (read data, write data, request, ack etc.) to connect multiple controllers together.

C. Scalability

We remark that the hardware design of minFlash scales with little increase in hardware resources. Only address widths increases with the number of devices. Latency of MGT links is extremely low at 0.5μs/hop, which is 100x better than Ethernet. Even a hundred network hops is still less than flash

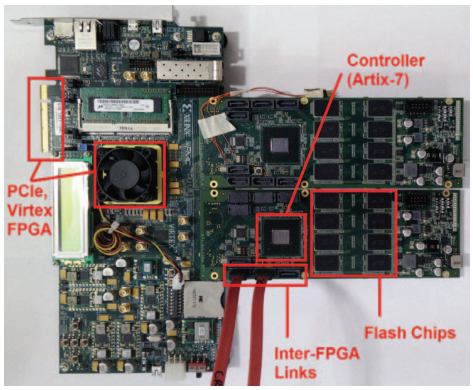


Fig. 4: minFlash prototype platform hardware

access time. Furthermore, by locally attaching each minFlash devices to a distributed set of host servers, all of the flash bandwidth is exposed. Linking more devices increases the total internal bandwidth of the flash storage as well as the storage capacity. Multiple compute servers can consume this bandwidth in parallel, each up to the peak rate of their storage interface (i.e. PCIe).

IV. MINFLASH PLATFORM HARDWARE

We implement minFlash using a Xilinx VC707 FPGA development board [16], which has a Virtex-7 FPGA, x8 PCIe Gen 2.0, and 4x10Gbps MGT serial links. We designed a custom flash board to attach to the VC707 via the FPGA Mezzanine Card interface (FMC) [17]. The flash board is a 512GB, 8-channel, 8-way design using 256Gbits Micron MLC NAND chips. The maximum aggregated bus bandwidth of the board is 1.6 GB/s. Typical latencies for the chip are 75μs for reads and 1300μs for writes. We use a smaller Xilinx Artix-7 FPGA on the board to implement the low-level flash controller. 4x10Gbps MGT serial links are pinned out as SATA ports on each device. Cross-over SATA cables are used as the inter-controller links to connect multiple boards together on a rack. Figure 4 is a photo of the hardware. This hardware is also being used in BlueDBM [18].

We used Ubuntu 12.04 with Linux 3.13.0 kernel for our software implementation and benchmarks. We used the Connectal [19] 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. [20] in our controller.

V. EVALUATION

We measure and evaluate (1) local device performance, (2) multi-device, multi-host performance and (3) user application performance. In summary, a local minFlash device can reach peak bandwidth of 1.2 GB/s (75% of theoretical bus bandwidth) with a mere 15μs of I/O latency overhead. Chaining multiple devices together in a distributed fashion adds trivial amount of latency, while effectively increasing the aggregated bandwidth. We run a database benchmark with a flash-aware file system on minFlash to demonstrate that reducing the storage I/O stack can provide superior overall performance.

A. Local Device Performance

1) Page Access Latency: Page access latency (Table I) is measured from the host as the time it takes for an 8KB read/write request to complete, including data transfer.

	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 I: Local Read and write access latencies

NAND intrinsic read latency was measured to be 69μs with an additional 43μs of transfer latency on the bus. The Reed-Solomon ECC decoder latency varies with the number of bit errors, and currently accounts for only 4μs. This is expected to increase as the flash chips age. PCIe flash request/responses, DMA and host driver incur an additional 11μs of latency. In total, the read access latency to local minFlash device is 128μs, with a mere 15μs (12%) overhead from the minFlash controller and driver. For writes the overhead 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). Overall, we observe that our raw flash interface provides accesses at close to native flash latency.

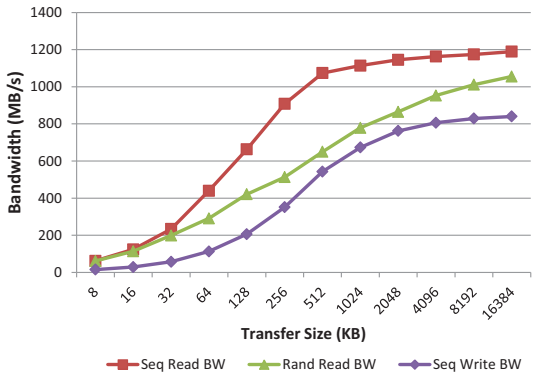


Fig. 5: Bandwidth vs. transfer size of the minFlash device

2) Bandwidth vs. Transfer Size: To measure bandwidth, address space is partitioned such that data is striped across channels and chips for maximum parallelism. Requests are issued for transfer sizes from single 8KB pages to 16MB chunks for both sequential and random accesses. Results are shown in Figure 5. 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. Higher level software needs to perform garbage collection and address remapping to handle random writes.

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

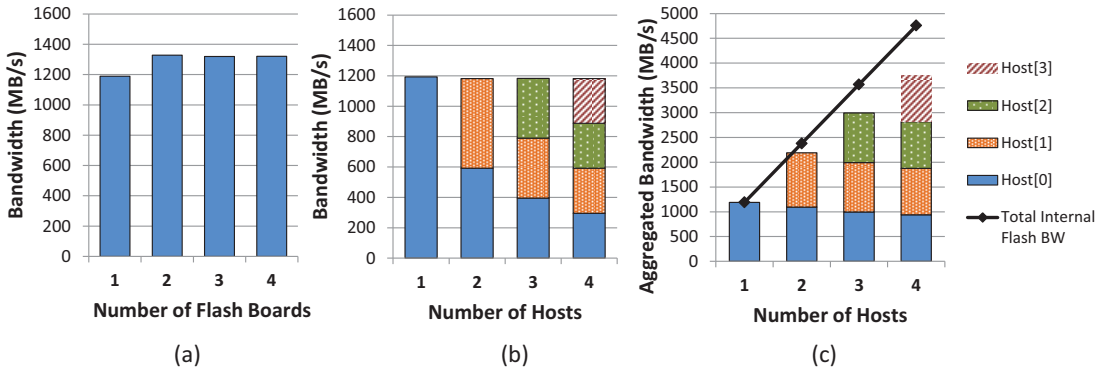


Fig. 6: Multi-device bandwidth. (a) single host, multiple flash boards. (b) multiple hosts, single flash board. (c) multiple hosts, multiple flash boards.

is slightly worse due to address collisions that would reduce parallelism. 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 our interface and controller are very efficient, and is able to fully expose the raw bandwidth of the flash array to the host.

B. Multi-Device Performance

Multi-device performance is measured by chaining together 4 minFlash devices in a linear array attached to 4 separate host servers.

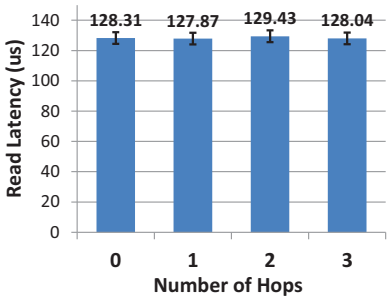


Fig. 7: Multi-hop page access latency

1) Access Latency: Figure 7 show the flash page read access latency over multiple hops of minFlash 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 devices are equally fast, minFlash's global shared-memory interface appears as fast as a local storage, even though it is physically distributed among multiple machines. In comparison, accessing storage over traditional networks incur at least 90μs of additional latency.

2) Bandwidth: We measure minFlash's bandwidth under the following scenarios: (1) single host accessing multiple connected minFlash devices (Figure 6a), (2) multiple hosts accessing the same device (Figure 6b), and (3) multiple hosts accessing multiple devices (Figure 6c). All accesses are random reads of 16MB chunks.

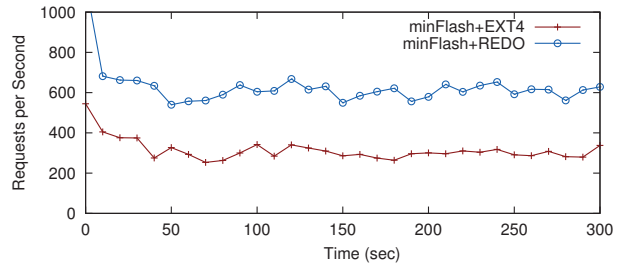
The first scenario is similar to a RAID-0 arrangement. We observe some speed-up (from 1.2 GB/s to 1.32 GB/s) by accessing multiple boards in parallel, but ultimately we are bottlenecked by PCIe (currently x8 Gen 1.0). We are in the process of upgrading our hardware IP to Gen 2.0, which would double the interface bandwidth. In general, because the total aggregated bandwidth from multiple minFlash flash boards is extremely high, a single server's storage interface cannot consume all of the bandwidth. minFlash 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 minFlash when there is resource contention for the same flash device. The graph shows that the minFlash 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 on the same data.

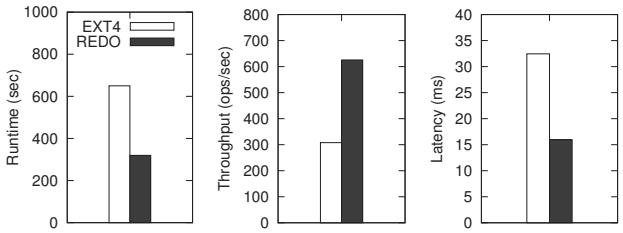
The last graph shows the aggregated bandwidth scalability of the global shared-memory flash store, with multiple servers *randomly* accessing the entire address space. The line in the graph shows the total maximum internal bandwidth provided by the flash devices (a simple multiple of the bandwidth of 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 79% with 4 hosts for a total of 3.8 GB/s. For even more hosts, we do expect the serial network bandwidth to eventually become a bottleneck. However, with help from the application to optimize for some access locality, minFlash can expect to reach close to peak internal flash bandwidth. Overall, minFlash wins in latency against SAN and distributed file systems, in bandwidth efficiency against flash arrays, and in scalability against locally attached flash.

C. Application Performance

Finally, we run a flash-aware file system called REDO [2] on top of minFlash to demonstrate its compatibility with host



(a) Requests per second over time



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

Fig. 8: YCSB benchmark results comparing (1) minFlash + REDO and (2) minFlash + page FTL + EXT4

software and the benefits of exposing flash characteristics. 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. We ran Yahoo Cloud Serving Benchmark (YCSB) [21] with MySQL+InnoDB at default settings, performing 200,000 updates to 750,000 records. We compare two I/O stack configurations: (1) minFlash+REDO file system and (2) minFlash + host page-level FTL + EXT4 file system. The latter configuration emulates a traditional SSD I/O architecture. Measurements are shown in Figure 8.

We see that minFlash+REDO doubles the performance of minFlash+FTL+EXT4 in both throughput and latency. This gain primarily stems 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 minFlash.

VI. Conclusion and Future Work

We have presented minFlash, a clustered flash storage platform that (i) exposes a shared-memory native flash interface to enable the optimization of the flash I/O stack and (ii) uses an inter-controller network within the storage device to bypass network overheads when scaling up. Latency overhead added by the platform is merely 15μs for accessing *both local and remote storage* since network latency is negligible. Aggregated bandwidth of the system also scales well even when there are simultaneous random requests from multiple hosts.

For future work, 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.

VII. Acknowledgements

This work was funded by Quanta (Agmt. Dtd. 04/01/05) and Samsung (Res. Agmt. Eff. 01/01/12). We thank Xilinx for their generous donation of VC707 FPGA boards and design expertise.

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] 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.

[3] S. Hardock, I. Petrov, R. Gottstein, and A. Buchmann, "Noftl: Database systems on ftl-less flash storage," *Proc. VLDB Endow.*, vol. 6, no. 12, pp. 1278–1281, Aug. 2013.

[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] 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.

[6] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. Panda, "Beyond block i/o: Rethinking traditional storage primitives," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, Feb 2011, pp. 301–311.

[7] "GFS2," https://sourceware.org/cluster/gfs.

[8] 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.

[9] "PureStorage FlashArray," http://www.purestorage.com/flash-array.

[10] "EMC XtremeIO," http://www.xtremio.com/.

[11] S. Miniman, "Server SAN Market Definition," http://wikibon.org/wiki/v/Server_SAN_Market_Definition, August 2014.

[12] "Hadoop Distributed File System," http://hadoop.apache.org/docs/stable/hdfs_user_guide.html.

[13] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka, "A cost-effective, high-bandwidth storage architecture," *SIGPLAN Not.*, vol. 33, no. 11, pp. 92–103, Oct. 1998.

[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.

[15] S.-W. Jun, M. Liu, S. Xu, and Arvind, "A transport-layer network for distributed fpga platforms," in *Proceedings of the 25th International Conference on Field-Programmable Logic and Applications*, ser. FPL '15, London, UK, UK, 2015.

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

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

[18] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind, "Bluedbm: An appliance for big data analytics," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 1–13.

[19] 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.

[20] 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.

[21] 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.