Beyond Block I/O: Implementing a Distributed Shared Log in Hardware

Michael Wei
UC San Diego

John D. Davis
Microsoft Research

Ted Wobber
Microsoft Research

Mahesh Balakrishnan
Microsoft Research

Dahlia Malkhi
Microsoft Research

Abstract

The basic block I/O interface used for interacting with storage devices hasn't changed much in 30 years. With the advent of very fast I/O devices based on solid-state memory, it becomes increasingly attractive to make many devices directly and concurrently available to many clients. However, when multiple clients share media at fine grain, retaining data consistency is problematic: SCSI, IDE, and their descendants don't offer much help. We propose an interface to networked storage that reduces an existing software implementation of a distributed shared log to hardware. Our system achieves both scalable throughput and strong consistency, while obtaining significant benefits in cost and power over the software implementation.

Categories and Subject Descriptors

B.1.1 [**Integrated Circuits**]: Types and Design Styles—*Gate arrays*; B.3.2 [**Memory Structures**]: Design Styles—*Mass storage*; D.4.2 [**Operating Systems**]: Storage Management; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*

1. INTRODUCTION

The advent of NAND flash has brought about a sea change in the storage industry. Since rotating media are accessed through a few slow mechanical disk heads, IOPS, especially random ones, are precious. However, in a flash array, each memory chip can potentially serve one or two orders of magnitude more random requests than an entire disk. Since there are many such chips in an array, I/O throughput is bounded only by the number of channels to the flash. Thus, the I/O rate of high-end SSDs often outpaces the ability of a general-purpose computer to issue requests. For example, Fusion-io claims to deliver 9 million IOPS [8] through custom APIs that bypass the operating system. In this environment, it is easy to see why vendors are moving toward appliances that offer specialized network fabrics to interconnect flash arrays as well as high-speed networks to connect multiple client computers to those arrays [20, 29].

SYSTOR '13, June 30 - July 02 2013, Haifa, Israel

A traditional storage server consists of a network-attached computer tied to a large storage array. In the past, array elements were rotating disks and the main problem has always been to make sure the disks could, in aggregate, keep up. Now, with lots of flash elements, each 100-1000 times faster than disk, the tables are turned. The best hope for keeping the flash busy is to increase the number of servers that a client can talk to, with many clients doing so concurrently.

Unfortunately, despite all the improvements flash has manifested in latency and I/O throughput, and despite the advent of fast networked access to such storage, client computers still access flash arrays through the logical-unit, block-read-write interface that hasn't changed all that much in three decades. Specifically, there has been little work on providing direct support for multi-client write access to shared storage and traditional modify-in-place semantics preclude the reconciliation of conflicting writes. Where shared (write) access to storage is required, an intermediary software service is often required to multiplex and coordinate request streams from multiple clients. For example, database and other transactional systems multiplex request streams in software long before they reach a disk. Similarly, while distributed file systems like FDS [18] and GPFS [22] might be able to stream data directly to block storage, shared access to metadata is typically mediated by a single software intermediary or via a lock server that provides temporary single-writer access at fine grain. Moreover, most distributed storage systems require that a server computer interpose between clients and media, at a minimum, translating between network and storage protocols.

The constraints cited in the previous two paragraphs form a conundrum. We need more servers to allow clients to drive the flash at speed, but doing so risks causing inconsistency. This conundrum is often solved by sharding of data with an independent mediator assigned to each shard [2]. However, this only moves the problem, since then operations involving multiple shards become difficult to coordinate.

This paper posits that an alternative storage device interface can facilitate shared and coordinated write-access amongst multiple client computers, and do so scalably without sac-

rificing consistency. We follow the well-worn example of reducing the core of a higher-level system to practice in hardware, specifically we support a distributed shared log. As previous work demonstrates [4], a shared log implemented in software can support multiple clients running applications that require both high thoughput and a total order on updates (such as databases or distributed key-value stores).

In this paper, we describe a hardware implementation that facilitates such a shared log. Our hardware implements the shared log protocols directly, without need for a coordinating server. Our device performs at least as well as previously published software prototypes. Specifically, it saturates a 1 Gb/s network link. However, the cost (both in dollars and power-budget) of our hardware implementation is only a fraction of cost of a server, saving an order of magnitude compared to the software implementation. Thus, we have a substantial advantage in scalability. Furthermore, we argue that special-purpose platforms such as ours, independent of form factor, present a viable hope for fully utilizing flash as network speeds increase to 10 Gb/s and beyond.

2. A DISTRIBUTED SHARED LOG

As described in the Introduction, we choose to expose a cluster of devices as a distributed shared log, thus circumventing the sharding pitfall. This approach follows our earlier work on the CORFU log [4]. It may be counter-intuitive that a single global shared log serves to circumvent the centralization bottleneck and to boost aggregate throughput. The trick in CORFU is to advance log-writers position by position extremely quickly using a centralized sequencer. Each reserved log position is filled directly and autonomously by a unique client, yielding utmost I/O parallelism. In order for this design to perform well, the sequencer is completely soft-state, and can orchestrate hundreds of thousands of client writes to the log per second. There remains the challenge of filling individual log positions consistently and reliably at high throughput.

The body of this paper is dedicated to the design of a storage hardware device capable of supporting CORFU writes, including the support for data replication. We refer to this device hereafter as a *shared log interface controller* or *SLICE*. Figure 1 depicts at high-level the translation of CORFU log offsets onto individual SLICE Virtual Addresses (SVA), which are mapped inside the SLICE onto SLICE Physical Addresses (SPA).

For the rest of this section, we briefly re-iterate the rationale behind the shared log approach; we refer the reader elsewhere [4] for a detailed discussion of the CORFU shared log, applications built with it, and its performance.

A shared log is a powerful and versatile primitive for ensuring strong consistency in the presence of failures and asynchrony. It can play many roles in a distributed system: a consensus engine for consistent replication, providing functionality identical to consensus protocols such as Paxos [15] (geographically speaking, Corfu and Paxos are neighboring Greek islands); a transaction arbitrator [10, 23, 25] for isolation and atomicity; an execution history for replica creation, consistent snapshots, and geo-distribution [14]; and even a primary log-structured data store that leverages fast ap-

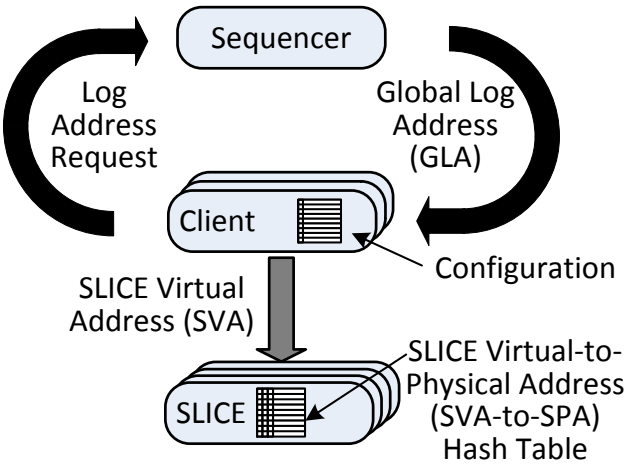

Figure 1: CORFU high-level architecture

pends on underlying media as in [21, 24]. We expect CORFU to enable a new class of high throughput distributed, transactional applications suitable for datacenter or cloud infrastructure. The key vision here is to persist everything onto the global log, and maintain metadata in-memory for fast access and manipulation. Indeed, we already have several positive experiences with systems built atop CORFU, including a coordination service which exposes an API compatible with ZooKeeper [11], the Hyder database [5], a general-purpose transactional key-value store and a state-machine replication library [4], and a log-structured virtual drive.

A shared log is also a suitable abstraction to implement atop a cluster of flash storage units. As has been argued before (e.g., see [2]), flash memory is best utilized in a log-structured manner due to its intrinsic properties. CORFU takes this approach one step further by treating an entire flash cluster as a single distributed, shared log, where client machines append to the tail of a single log and read from its body concurrently. Each log entry is projected onto a fixed set of flash pages and data is made resilient by replication over the page-set. The cluster as a whole is balanced for parallel I/O and even wear by projecting different entries onto distinct page sets, rotating across the cluster. In this design, CORFU completely adopts the vision behind networked, log-structured storage systems like Zebra [9], which balance update load across a cluster in a workload-oblivious manner. The difference is that the CORFU log is global and is shared by all clients. Furthermore, the storage servers enforce properties discussed in the next section that are required for a coordinated and shared media.

3. THE SLICE API

Guiding our storage device design is the CORFU vision that we can avoid any centralized meta-service on the I/O path from clients to storage, yet support strong consistency. There are three key effects on the design of a SLICE unit in this regard:

1. Since multiple clients may attempt to access the same physical storage page, and barring the use of a meta-

seal(epoch). This command marks all the addresses belonging to the epoch as read-only and marks the current epoch as *sealed* at the target SLICE.

read(epoch, sva). This command reads a page of data at the given virtual page address. On success, it returns data and the epoch when the data was written. Otherwise, it returns an error value. Reads and writes operate on 4KB pages. This size was picked to match the page size of commonly available NAND flash parts, but there is nothing fundamental about this choice.

write(epoch, sva, data). This command writes a single page of data to the virtual address. If the epoch is current (not sealed), then the command writes the data to the address and returns a positive acknowledgement. Otherwise, it returns an error value. A SLICE cannot count on writes arriving from the network in log order.

Only one write is allowed to each address. If a client attempts to write a page that has already been written, an error value is returned.

check(epoch). This command returns the maximal written position as well as the maximal extent of contiguously written entries.

trim(epoch, trimtype, sva). This command marks written pages that can be erased during garbage collection. There are two variants of the virtual address argument referred to as *prefix* and *offset* trim. Prefix trim operates on the entire prefix of the address space before the argument address. Offset trim operates on a single address alone.

Figure 2: SLICE API

service to manage access capabilities, SLICE exposes a *write-once* address space.

2. In order to enforce write-once semantics on pages that have been written and reclaimed for cleanup, the address space of every SLICE grows infinitely, limited only by the device's lifetime.

3. To manage configuration changes to the cluster, each SLICE is marked with a configuration *epoch*. In this way, a SLICE can deny service from clients which are not aware of the current epoch's configuration.

Figure 2 provides a complete description of the SLICE API, followed by a full account of its rationale. As originally envisioned in the CORFU publication [4], this API is simple and concise enough to be implemented in hardware. We do not designate a specific network protocol underlying this API, because it can be implemented atop any protocol with reasonable integrity guarantees. Also, we note that although the focus of this paper is using NAND flash as the underlying storage media, this API would apply to other storage media (i.e., HDD, Phase Change Memory, etc.), with the most likely difference being in caching and data access granularity size.

A variety of considerations, which are intertwined with the CORFU protocol design, contribute to the above API, and we discuss them below.

Handling Reads/Writes. CORFU is implemented by using a client-side library to project virtual log-entries onto page addresses on individual SLICEs. Clients access those addresses directly so as to perform reads and writes.

The crux of allowing multiple clients to write concurrently to the same device lies in how writes are handled. As mentioned in the previous section, CORFU utilizes a soft-state sequencer to prevent clients from attempting to write the same log position. However, a client that wins access rights to a page might hang indefinitely, leaving a **hole** in the log at a position it had intended to write. If a hole remains unfilled, then clients that count on reading the entire log must block, believing that the hole is the end of the log. CORFU deals with this by permitting log readers to "fill holes" with recognizable junk as needed. This introduces a potential race between writers and hole-fillers. In order to address this potential race, we count on the SLICE address space to be write-once, so either the writer or the hole-filler will succeed, but not both.

Handling Replication. More generally, the SLICE write once semantics allows CORFU to provide fault tolerance through consistent data replication in a purely client-driven manner. Briefly, clients replicate writes to multiple servers using a variant of Chain Replication [28]. A client copies data from replica to replica using the fixed order of the chain. We can recover from a failed write by continuing any partially filled chain in the same manner, copying the prefix to complete the chain.

Handling Trims. Because physical flash isn't infinite, applications must occasionally *trim* the log. As mentioned above, a SLICE provides two flavors of trim. Prefix trim can be used to produce a compact log with few trimmed positions between the log head and tail. An application implementing a strict snapshot-and-log persistent data structure might make use of such a log. Offset trims, alternatively, can provide a sparse log with valid data distant from the active log head. This can be useful, for example, when constructing a log-structured file system or block store. When applications dictate a compact log, the garbage collector has little to do other than erase storage. However, a sparse log is more difficult to handle. As in SSD garbage collection, we must find blocks with the least number of valid pages, relocate those valid pages, and erase the blocks so that they can be rewritten, while balancing this with block wear-leveling.

Handling Reconfiguration. When storage elements fail or capacity is added, the system must undergo a reconfiguration to add new SLICEs or remove failed ones. Clients are alerted to changes in the global configuration when a SLICE indicates that a client's working epoch has been sealed. If such a reconfiguration takes place, all the existing SLICEs must be sealed so as to deny further mutations within the old epoch. This protocol ensures inter-client consistency by prohibiting operations by clients that do not know of the new epoch.

Why an Infinite Address Space. Putting together write-once semantics and trimming implies that we cannot recycle SLICE addresses and use a trimmed address to refer to a reclaimed pages; this might lead to a violation of the write-once semantics. Hence, a SLICE exports an infinite space of virtual addresses (called SVA, see Figure 1), which is mapped onto its finite space of physical addresses (referred to as SPA in Figure 1). An alternative design would be possible for prefix trims: handle trims via reconfiguration, thus generating new epochs to circumvent the need to overwrite reclaimed addresses. However, a compact log requires the application to work hard to move old data to the front of the log. Moreover, sparse logs are useful for a number of workloads. We therefore decided to support sparse logs in the most natural way, allowing local SLICEs to perform local wear-leveling and garbage collection as do individual SSDs, but also giving good performance for applications that manage logs in a compact fashion.

4. PROTOTYPE IMPLEMENTATION

We implemented our SLICE prototype on an FPGA using the Beehive many-core architecture [17]. In the following section, we describe the data structures our prototype uses to satisfy the requirements of the SLICE API. Then we outline the hardware itself and the control flow used in the processing of shared-log requests. We conclude the section with a look at a few specific design details.

4.1 Implementing the API

In order to support an infinite address space, the storage device must provide a persistent mapping from a (potentially sparse) 64-bit virtual address (SVA) onto a physical address (SPA). SSDs often use such a structure, although the map's domain is usually limited to the nominal disk size of the SSD, and the granularity of the mapping function is often coarser than a single page. There is considerable overlap between what is described here and the functionality of the Flash Translation Layer (FTL) firmware found in an SSD. Thus, there is good reason to think about merging these components. We discusses this possibility in Section 4.6. In practice, an SVA need only be large enough to support the maximal number of writes for a given device. Since NAND flash supports a limited number of erase cycles, we can base our data structures on the notion that the size of an SVA is roughly bounded by the number of flash pages times the maximal erase cycle count.

Our current implementation uses a traditional hash table to implement a map that resolves to flash pages of size 4 KB. This data structure occupies 4 MB of memory per GB of target flash. We have designed, but not implemented, a significantly more compact structure using Cuckoo Hashing as described in Section 4.4.

The referent of the page map contains per-page state (e.g.: unwritten, written, trimmed) as well as an SPA if the page is in the written state or awaiting reclamation. We keep three pointers with regards to the overall SVA space on each SLICE: a head pointer to denote the maximum written entry; and a minimum unwritten pointer (below which there are no holes); and a pointer below which all trimmed pages have been reclaimed. An additional pointer indicating the minimum written position can also be used to restrict the set of logical addresses under consideration during prefix trim. These pointers need not be maintained persistently since they can be recovered from the mapping table. All trimmed positions that both lie below the minimum unwritten pointer and have been reclaimed can be eliminated from the map.

We optimize the hole-filling operation by using a special value of flash page pointer to denote the junk pattern that is used to fill a hole in the log. Thus, hole-filling can be accomplished by manipulating the mapping table: set the physical page pointer to the junk value and mark the page as trimmed. In addition to the mapping structure, the SLICE implementation must track the set of sealed epochs and maintain a free list of flash pages for new writes. The former must be stored persistently, but the latter can be reconstructed from the mapping table. For best performance, the ordering of the free list should take into account specific peculiarities of the media, such as locality or the need to perform sequential writes within flash blocks.

Should it become necessary to efficiently enumerate very sparse logs, we could introduce a data structure to track ranges of reclaimed addresses and an API method to access it (e.g, FindNextWritten). However, the applications we have built so far only walk through the compact portion of logs, so we have not yet found the need to take such measures. Some of the newer API functions that are part of the software implementation of the CORFU server have not yet been fully implemented in hardware. For example, the minimum unwritten pointer was not part of our original hardware design. These features can and will be reintegrated straightforwardly into the current design.

4.2 Hardware Design

Our prototype hardware design is presented in Figure 3. Each SLICE comprises an FPGA with a gigabit Ethernet link, a SATA-based SSD, and 2 GB of DDR2 memory. The design is flexible and scalable: hundreds of SLICEs may be used to support a single shared log given sufficient network capacity. We have engineered our SLICE unit to be inexpensive and low-power while delivering sufficient performance to saturate its Ethernet link. While our prototype unit is built with an FPGA, we envision that a production device would be built with a low-cost ASIC and a NAND flash array instead of a SSD, offering a better performance, lower price, and lower power than the platform that we are currently using.

Inside the FPGA, we use a variant of the Beehive. Beehive is a many-core architecture implemented in a single FPGA. A single Beehive instance can comprise up to 32 conventional RISC cores connected by a fast token ring. Network interfaces, a memory controller, and other devices, such as disk controllers, are implemented as nodes on the ring. Control messages for memory access traverse the ring as do data writes to memory. Data is returned from reads via a dedicated pipelined bus. There are additional data paths to enable DMA between high-speed devices and memory.

We configure various Beehive cores to take on specific roles, as shown in Figure 4. Whereas the memory controller, Ethernet core, and System core are common to all Beehive de-

	% Idle	Instructions
System	15.5%	2,544
PacketProc	0.7%	4,116
Read	8.0%	612
Write	8.9%	634
Metadata	3.95%	712
SATA	5.3%	1,110
Comm	N/A	516

Table 1: SLICE per core idle time and number of assembly instructions. (Comm core code written in assembly, all other core code written in C.)

signs, we use the following special-purpose cores to construct a SLICE.

- A packet processing core handles the parsing of network requests and the formatting of responses.
- A metadata core manages the mapping table.
- A SATA core coordinates I/O to the SSD.
- A read and a write core interact with the metadata core, and initiate and monitor the completion of SATA requests.

The Beehive architecture enables us to handle requests in parallel stages while running the FPGA at a low frequency (100 MHz), thus reducing device power. Note that new functionality can be easily added to the SLICE design. Additional cores running specialized hardware can enhance the performance of timing-critical tasks. For example, our current design uses a specialized hardware accelerator to speed up packet processing. At the same time, latency-insensitive operations can be coded in a familiar programming language (C), significantly reducing complexity.

Table 1 shows the percentage of time the various cores are idle under maximal load and number of assembly instructions per core in the SLICE design. The Comm core has a slightly different architecture than the rest of the cores (it runs all its code from ROM and cannot execute/read/write DRAM except using DMA), thus we did not measure its idle time. If we need more or differently allocated compute resources, we can use different configurations of cores. In an earlier alternative design, we used two Packet Processing Cores running the same code base: one processed even packets and the other processed odd packets. The earlier design used more FPGA resources than the current design, but both designs can run the Ethernet at wire speed. We could also just as easily add a second Comm, PacketProc, Read or Write core, should the workload require it.

4.3 An Example Write Request

Requests to a SLICE arrive over the network. As an exemplar, we describe, below, the nine steps required to service a write request as it moves around the ring, as shown in Figure 4.

1. The SLICE receives a packet at the Communication (Comm) core.

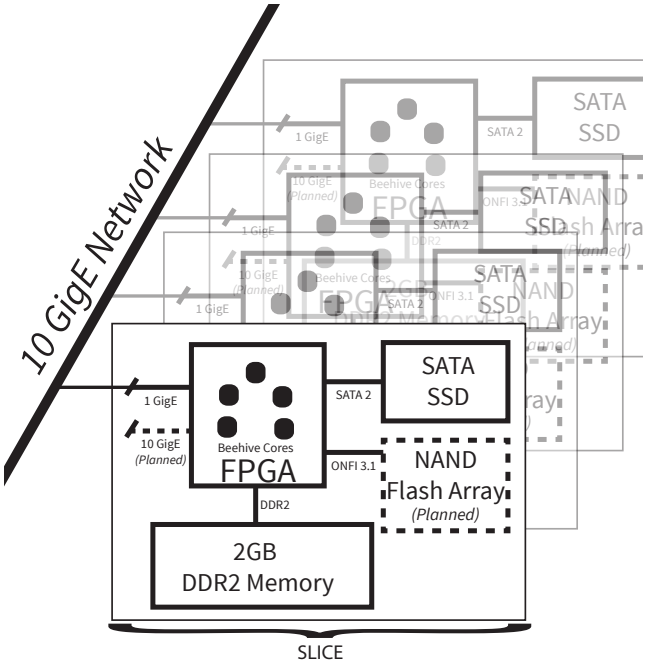

Figure 3: SLICE prototype hardware system design. Each SLICE contains a FPGA which receives requests over gigabit Ethernet and serves them using an SSD. Future designs may implement 10 gigabit links and direct access to a NAND flash array.

2. When a packet is received (we support jumbo packets up to 9,000 bytes) it is placed into a specific location in DRAM in a circular buffer using DMA and a page token is created.
3. The Comm Core forwards the page token to the Packet Processing Core to process the packet header information.
4. The Packet Processing Core forwards the page token to the Write Core. The Packet Processing Core also starts to construct a reply message for the client while the packet request is satisfied.
5. The Metadata Core receives the page token to check the metadata to make sure the SVA has not been written or the epoch has been sealed and picks a SPA off the free-list. The SPA and the memory address for the data are forwarded to the SATA Core.
6. The SATA Core reads a page from the data buffer located at the memory address and stores the page on the SSD at the specified SPA.
7. The SATA Cores informs the Write Core that the request is complete by sending the page token back, starting the reverse journey back to the Comm Core.
8. The page token is sent back to the Packet Processing Core to complete the reply packet.
9. The Comm Core sends the reply packet back to the client when the page token returns with a pointer to the reply packet.

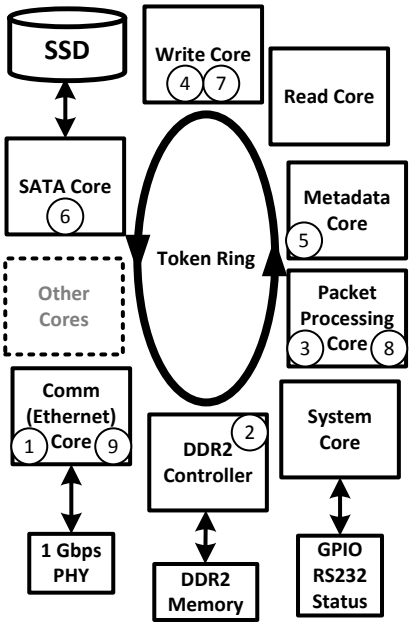

Figure 4: SLICE hardware architecture. Inside the FPGA, Beehive cores and a DDR memory controller are connected via a token ring. Specialized cores allow the system to interact with peripheral hardware such as the network and storage.

4.4 Address Mapping Using Cuckoo Hashing

We now present our design for using Cuckoo Hashing to efficiently map an SVA to an SPA. Cuckoo Hashing minimizes collisions in the table and provides better worst-case bounds than other methods, like linear scan [19]. Under Cuckoo Hashing, two (or more) mapping functions are applied to each inserted key, and such a key can appear at any of the resultant addresses. If, during insertion, all candidate addresses are occupied, the occupant of the first such address is evicted, and a recursive insert is invoked to place it in a different location. The original insertion is placed in the vacated spot. On average, 1.5 index look-ups are required for successful lookups in such a table. Table lookups for entries not in the table always require two lookups, one for each mapping function.

In order to save space in each hash table entry, we store only a fraction of the bits of each SVA. The remainder of the bits can be recovered by using hash functions that are also permutations. Such permutations can be reversed, for example during a lookup, to reconstruct the missing bits so as to determine whether the target matches. The end result of hashing an SVA can then be represented by the mapping function F which is the concatenation F_1 and F_2, computed as described below. The lower order bits of F are used to index into the mapping hash table and the remainder of F is stored in the table entry for disambiguation, along with a bit indicating which mapping function was used. This ensures that for any given table entry, we can recover all of F from an entry's position and contents, and thus we can derive X and Y, and finally the original SVA.

An example of the forward and reverse process for the mapping function is provided below:

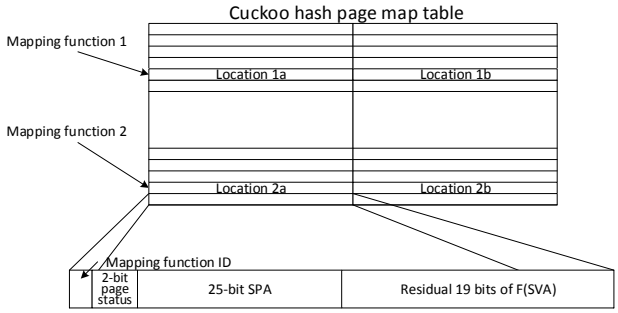

Figure 5: Cuckoo hash page map and table entry.

- Split the SVA bitwise into two values X and Y of equal size.
- Given two hash functions, H_1 and H_2, compute functions F_1 and F_2 on X and Y as follows:
  - $F_1 = H_1(Y) \bigotimes X$
  - $F_2 = H_2(F_1) \bigotimes Y$
- X and Y and hence the original SVA can be recovered from F_1 and F_2 as follows.
  - $H_2(F_1) \bigotimes F_2 = H_2(F_1) \bigotimes (H_2(F_1) \bigotimes Y) = Y$
  - $F_1 \bigotimes H_1(Y) = (H_1(Y) \bigotimes X) \bigotimes H_1(Y) = X$

A mapping function can be built using hash functions H_1 and H_2 from a collection of 256-entry hash-value arrays. By splitting the argument into byte-sized values N_i, we compute H_1 by treating each N_i as an index into the i^{th} array, and XORing the results. H_2 is computed similarly using $255 - N_i$. This reduces the number or arrays by a factor of two, reducing on-chip storage requirements.

To express these data structures more concretely, consider a 128 GB SLICE (32 million 4KB pages). If NAND flash can be erased 100,000 times, this equates to a device endurance of 3×10^{12} page writes, or 42 bits. We use Cuckoo Hashing with two mapping functions. However, in order to achieve better memory cache locality, each index bucket contains two entries, creating four possible spots for each new SVA insertion. We provision a Cuckoo hash table that is 20% larger than the number of physical page entries. For 128 GB of flash, we use a table with 2^{26} 6 byte entries, which consumes roughly 240 MB of memory, or about 1.88 MB/GB.

Figure 5 shows the page mapping table with the two mapping functions and the required metadata page entry of 48 bits: 25 bits of physical page address, 19 upper bits of F, 1 bit for flash block status, 1 bit for mapping function ID, and 2 bits for the page status (e.g., written, trimmed, or unwritten).

We evaluated a software implementation of the Cuckoo Hashing page mapping scheme and compared it with Chain Hashing. To do so, we ran sequences of insertion / lookup pairs using a varying number of keys on hash tables of both types,

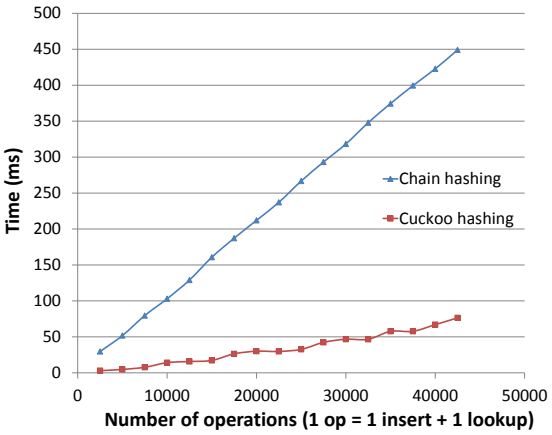

Figure 6: Time required to insert and lookup keys in a Chain hash versus a Cuckoo hash page map.

and then compared the elapsed times. Figure 6 shows the difference in performance, about 10X, when using the two page mapping schemes. We used a 64,000 entry table for both tests. These tests employed a dense key-space with relatively few hash collisions. The advantage of Cuckoo Hashing should increase with the likelihood of collisions.

4.5 Persistence

The stability of SLICE storage depends on the persistence of its mapping table. Building a persistent mapping table for a CORFU software implementation is problematic. Writing separate metadata for every data write is not plausible. The remaining possibilities either involve batching metadata updates, which risks losing state on power failure, or writing metadata and data in the same chunk, which reduces the space available for data. Fortunately, when custom hardware is in play, a further option becomes available. Using super-capacitors or batteries, we can ensure that the hardware will always operate long enough to flush the mapping table. Our optimized mapping table takes only a few seconds to flush to flash, so this is an attractive option for metadata persistence (and many SSDs use the same technique). We have specified the hardware needed for this capability, but not yet implemented it. Ultimately, solid-state storage with fine write granularity, such as PCM, would provide the best alternative for storing such metadata and modifying it in real time.

4.6 SLICE and the Flash Translation Layer

Our SLICE prototype uses an existing SSD rather than raw flash. Using an SSD, each SPA referenced in our mapping table is a logical SSD page address. This was an expedient for prototyping, and it eliminates a raft of potential problems. For instance, we don't need to worry about out-of-order writes, since these are possible on an SSD but problematic on raw flash. Furthermore, we don't need to worry about bad block detection and management or error correction. But the most significant problem that using an SSD eliminates is the need to handle garbage collection and wear-leveling. With an SSD, allocating a flash page during a write operation is as simple as popping the head of the free list. Similarly, reclaiming a page requires adding it to the free list and (optionally) issuing a SATA TRIM command to the drive. Wear-leveling is performed by the SSD.

The downside of using an SSD is that it duplicates Flash Translation Layer (FTL) functionality. Specifically, our mapping table requires a extra address translation in addition to that done by the SSD. Since SSDs are fundamentally log-structured, and since we are in practice writing a log, which is significantly simpler than a random-access disk, one might hope that this would result in a less complex FTL. A further downside is that we lose control over the FTL, which might have been useful to facilitate system-wide garbage collection. For example, if there are many SLICEs in a system, it is possible to use the configuration mechanism in CORFU to direct writes away from some units and allow garbage collection and wear-leveling to operate in the absence of write activity. In addition, if we had access to raw flash, our system would be able to store mapping-table metadata in the spare space associated with each flash page and possibly leverage this, ensuring persistence without special hardware, in the manner of Birrell et al. [6].

Fortunately, it seems likely that writing a log over an SSD will in many cases produce optimal behavior. An application that maintains a compact log works actively to move older, but still relevant data from the oldest to the newest part of the log. Doing this allows such applications to trim entire prefixes of the log. This sort of log management is appropriate for applications that maintain fast changing and (relatively) small datasets, such as ZooKeeper [11]. With this sort of workload, appends to the log march linearly across the address-spaces of all the SLICEs, and prefix trims at the head of the log proceed at the same pace. This should produce optimal wear and capacity balancing across an entire cluster. Assuming that our firmware allocates SSD logical pages in a sequential fashion, the regular use of prefix trim should help avoid fragmentation at the SSD block level which is a major contributor to write amplification [1].

In other applications, for example a CORFU virtual disk, it can be too expensive to move all old data to the head of the log. Because offset trim operates at single page granularity, we can support applications that require data to remains at static log positions. In this case, the flash array must make the usual tradeoffs between leaving data in place and balancing wear by moving data, regardless of whether we use an SSD or raw flash. The FTL in an SSD manages these tradeoffs all the time, but if we implemented the FTL, we would then get the option to do it within a SLICE, in a more distributed fashion, or both.

5. EVALUATION

We evaluate three different CORFU server instances (two software and one hardware, respectively): traditional Xeon-based server-attached storage, low-power Atom-based server-attached storage, and our SLICE prototype. The Xeon server is a dual socket Dell 2950 with E5345 cores running at 2.33 GHz. The Atom server is a Zotac dual-core N330 running at 1.6 GHz. The SLICE prototype platform varies for the different tests described below. Both the Xeon and Atom-based servers run Windows Server 2008 R2 and a software CORFU server as a user-level process.

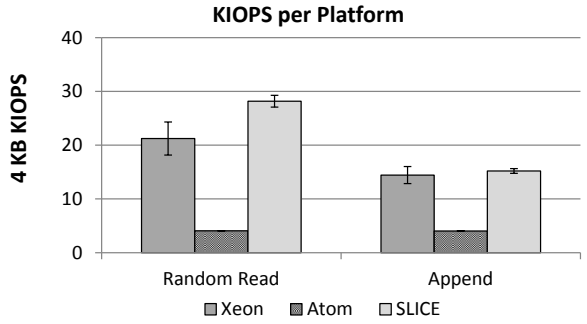

Figure 7: SLICE read and append throughput.

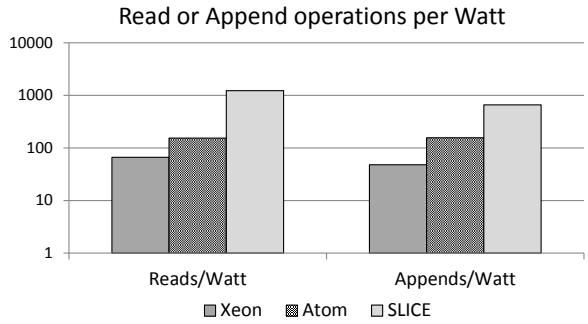

Figure 8: Requests serviced per Watt.

5.1 Single Instance Test

Our evaluation first focuses on a single SLICE instance, as we seek to understand the performance, power and cost (in terms of die area) benefits of our custom design compared to more traditional network storage approaches. We do not compare against more expensive SAN or NAS solutions. Our SLICE hardware platform for this test is the Xilinx XUPV5 development board [30]. All single instance platforms employ an Intel X25-M SSD [13] with write-caching enabled. This SSD is specified to achieve 35K random read and 17K sequential write operations per second (4 KB). Network connectivity is provided by a 10 Gb/s router with 1 Gb/s downlinks. Requests are generated by clients, running as user-level processes on Xeon-class Windows machines. All three platforms handle the same number of I/O requests: reads generated by a two test clients issuing up to 64 simultaneous requests. We report average request completion bandwidth in Figure 7. A UDP-based network stack is used for these tests with jumbo Ethernet frames.

Given that we use 1 Gb/s links, and accounting for communications overheads, we can at best hope to be able to handle around 29,000 operations per second. The SLICE platform achieves around 28K random read IOPS while the Xeon server achieves around 22K for the same test. The Atom system is CPU-bound, spending over 80% of its time in kernel mode handling interrupts. The 8-core Xeon-based system never exceeds 50% CPU utilization and also spends the bulk of its time servicing interrupts. The SLICE does not perform appreciably better when SSD operations are replaced with in-memory copies. It is worth noting that the end-to-end latency of a Xeon-server read operation is several times longer than that on the SLICE.

As shown by Figure 7, both the SLICE and Xeon servers perform close to the SSD-manufacturer's specified sustained sequential bandwidth during the append test. Because we are constructing a log-structured store, we hope to get sequential write performance. Even though requests can be reordered on the network, the SLICE implementation has the ability to place incoming write requests in sequential order on the SSD (since the SLICE controls the mapping to physical addresses). The functionality is currently lacking in the Windows version, and thus the Xeon append performance is slightly worse. The Atom append performance is CPU-bound, as it is for reads. Our measurements show that the Xeon is handling nearly 30,000 interrupts per second, while the Atom handles about half that rate. As our results demonstrate, the low-power Atom is clearly not capable of handling the number of interrupts required to saturate the network channel in this configuration. While the Xeon implementation could possibly be further tuned, the read path is already extensively optimized and we are still experiencing latencies in the critical path. We built a well balanced SLICE architecture that fully utilizes the cores shown in Figure 4. As Table 1 shows, at maximum IOPs, the cores are almost fully utilized. The benefit of the SLICE design is that if we needed higher performance, we could add more cores, e.g., use multiple Comm and/or PacketProc cores. On the other hand, we believe it to be very difficult to extract optimal performance from traditional operating systems running on multi-core platforms under high interrupt load [31].

The results of these performance tests suffer high variance, particularly for append. This is exacerbated by variable SSD performance and by the peculiarities of interrupt thread affinity on multi-core machines (both client and server).

Power Consumption. Eliminating the server from the network-attached storage equation reduces power and improves system efficiency. In Figure 8 the log-based y-axis records the number of requests that can be serviced per Watt for the three systems in the single server test. The Xeon consumes about 300 W during our tests, whereas the Atom platform consumes at 26 W and the SLICE consumes 23 W. (We do not count the power consumption of the sequencer under the assumption that it is amortized over the cluster.) The Xeon and SLICE systems are not CPU-bound, and as expected, the appends require more power because of the additional steps required to perform an append, resulting in fewer appends per Watt. In contrast, the Atom platform is CPU-bound, capping the number of requests that can be serviced and limiting the number of requests per Watt to be the same for both reads and appends. Notably, the Atom provides more requests per Watt than the more powerful Windows machine. However, the SLICE provides 4-18 times better performance in this metric than both software implementations.

5.2 FPGA Utilization and ASIC Area

We use the Xilinx LX110T Virtex-5 FPGA to realize the SLICE prototype. Although we used the Virtex-5, this design could be realized on a much cheaper Spartan-6. Table 2

SLICE	LUTs	BRAM
V5-LX110T	36%	24%

Table 2: Small FPGA utilization for the SLICE prototype.

SLICE	ARM A9 (power)	ARM A9 (perf)
0.618 mm^2	2.30 mm^2	3.35 mm^2

Table 3: 40 nm area estimates for the SLICE, power optimized ARM Cortex-A9 and performance optimized ARM Cortex-A9 [3], respectively.

provides LUT and BRAM usage which includes the memory controller, caches for the cores, and necessary buffering for the pages.

If the SLICE were a high-volume part, an ASIC could be a possible target, providing lower chip costs, higher performance, and lower power. We used the FreePDK technology library for the 45 nm silicon process [26] to estimate the area our design would occupy on an ASIC. We pulled out the memory structures from the SLICE design and used the Synopsys Design Compiler to provide area estimates. The memory structure area for the caches, FIFOs and other memory structures are estimated using SRAM bit cell estimates that include standard overheads for these structures. Finally, we scaled down the device area to a 40 nm process to match the ARM published area estimates for an A9 ARM core [3], commonly used in NAS or other embedded systems as a reference point. The low area overhead demonstrates the low effort and cost required to implement the SLICE as an ASIC. As shown in Table 3, the SLICE, occupying less than 1 mm^2, is approximately 18 to 27% the size of a performance or power optimized A9 core, respectively. In fact, our area estimate is pessimistic because it includes the area of a memory controller, which is part of the system-on-chip and not included in the ARM Cortex-A9 core area [3].

5.3 Scale Test

The final part of our evaluation focuses on scaling in a distributed context. We use a CORFU server configuration with no replication in which 16 clients generate requests for up to eight SLICEs. The SLICEs are implemented on the BEE3 [7], a multi-FPGA development platform. The SLICEs from each BEE3 are connected to different top of rack switch with 1 Gb/s downlinks and one 10 Gb/s uplink. The clients run on the same Xeon-class machines used in the single-server experiment, and are connected to a switch similar to that used for the SLICEs. The BEE3 and client switches are connected through a single 10 Gb/s switch.

We compare the BEE3 results to those obtained from a set of software CORFU servers running on a setup similar to the clients, again on an isolated switch. We report BEE3 numbers using a UDP protocol stack with jumbo frames, while the software server stack employs TCP, which currently gives better performance than our UDP server implementation. Although our system can scale to more than 8 SLICEs, even with our current configuration we are nearing the point where the 10 Gb/s switch is a bottleneck.

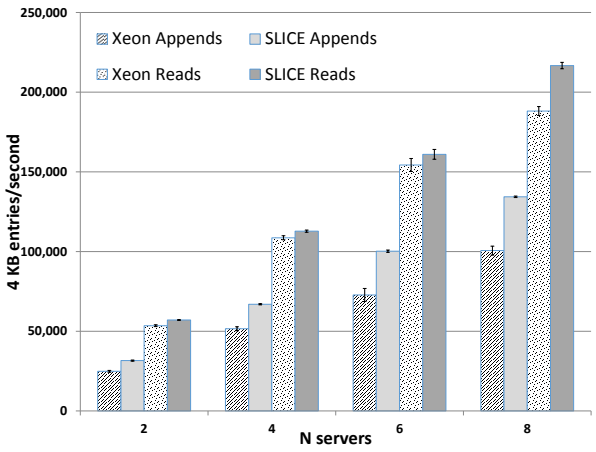

Figure 9: SLICE append and read scaling.

All of the SLICE instances and software servers include a single Intel 320 SSD at 120 GB capacity. This drive is newer than, but quite similar in random-read performance to the X25-M used in the single instance test. The Intel specification [12] states that it can achieve 38K random 4KB read IOPS, 14K random 4KB write IOPS, and 130 MB/s of sequential writes.

Figure 9 shows aggregate system throughput for appends and reads using a constant number of clients and a varying number of servers. Read tests are bounded by the 1 Gb/s links for both platforms. For two to eight servers reads scale linearly. As can be seen in the figure, the SLICE marginally outperforms the Xeon platform for reads. In the eight server case, we are able to obtain network goodput equaling 89% of the available bandwidth to the each SLICE, while only about 77% using the Xeon. The remaining bandwidth is partially occupied by network packet overheads. It is to be expected that the software server bandwidth would be somewhat lower due to the absence of jumbo Ethernet frames.

In the append results, we would expect performance to be bounded by the SSDs for the Xeon configuration and by the network for the SLICE. The Xeon servers consistently get near 14K IOPS per server, which is the maximum provided by the SSD for random writes. As expected, the software server can only achieve the random write rate because appends can and do appear out of order, and they are not reordered on the way to disk. We expect this limitation to be more important here as compared to the single instance test given the greater number of clients. On the other hand, the SLICE gets over 16K IOPS because it always writes sequentially. Although this is less than the full sequential speed of the SSD, it is not clear that relatively small sequential and asynchronous write requests can actually run the disk at full sequential speed.

In order to understand the limits of scaling on appends for greater numbers of servers, we must investigate the CORFU log sequencer. We can add SLICEs to increase append bandwidth, assuming an adequate network, but performance is ultimately limited by the sequencer. We have demonstrated a fast sequencer in user space on a standard multi-processor

and that can issue 570K tokens per second on pre-existing TCP connections with no batching. This sequencer uses the Windows Registered I/O Networking Extensions [16] to avoid buffer pinning and kernel wakeup overheads. It's important to note that a NIC fully capable of distributing incoming network load across multiple processors is required because IP-stack traversals would limit performance if executed serially. Even two-fold batching of sequence numbers would result in service well over 1 million per second, thus requiring more than 64 SLICEs for full append utilization even if network bottlenecks could be avoided. The challenge will be to find distributed applications that require so much throughput.

5.4 Discussion & Future Work

Our performance results are not surprising in the context of the previous work by Suzuki et al. [27], which demonstrates how to use an FPGA architecture to extend a PCIe bus over Ethernet so as to access a fast SSD. Unlike [27], our work is focused on a shared, distributed flash storage solution that gives performance and consistency. We also remove the intermediary server to reduce power consumption and use a consumer-grade SSD. In fact, our numbers are only somewhat lower than those of Suzuki even though they used a faster network and a much more capable and expensive SSD. Moreover, we were not entirely sure that we could saturate even a 1 Gb/s network using the very slow 100 MHz Beehive soft cores. As it turned out, our concerns were unwarranted.

Our SLICE implementation uses polling rather than asynchronous interrupts and thus avoids the interrupt bottlenecks experienced by general-purpose servers [31]. Since all our cores perform a dedicated function, polling cycles are never wasted cycles. Furthermore, we can easily substitute hardware logic in performance critical code paths, and have done so to optimize various part of our design such as an Ethernet offload engine and SATA DMA engine. Moreover, our device's low power-consumption helps realize the true energy-savings that are possible with network-attached solid-state storage.

There are several topics for future work. Clearly, it would be preferable to address raw flash rather than an SSD. We believe there are efficiencies to be gained by integrating our SLICE architecture and an FTL. For example, there currently are two logical-to-physical mappings in our prototype, where only one would be optimal.

Wear-leveling and garbage collection is a simple matter if the SLICE log is always contiguous and moving forward. However, management of flash is more complicated if data becomes static, creating gaps in log order. In this case, there may be advantages in constructing a distributed FTL to perform wear-leveling holistically across an entire cluster.

Finally, in our SLICE design, we did not address the question of persistence for metadata, for example mapping tables. We assume that battery or super-capacitors can be used for power while we flush metadata to stable storage in the case of catastrophic failure. We view this as a requirement for a production system and drop-in hardware solutions exist.

6. CONCLUSIONS

In this paper, we have demonstrated a hardware implementation of an API that allows direct, networked-client access to a distributed shared log. This API requires no intermediary server to arbitrate conflicts between concurrent writes. With a 1 Gb/s NIC, our FPGA implementation is bounded by the network for reads. Its performance equals or surpasses that of a general-purpose server implementing the same API. Yet, the cost in parts and power of our FPGA solution is much less than the comparable generic server box. We cannot claim that our implementation is optimal, but its storage throughput is well-balanced with its network interface capacity, and that was the ultimate goal.

The question that remains to be answered is what will happen at 10 Gb/s and beyond. There is considerable evidence that SSDs will continue to scale up in IOPS and bandwidth. However, even our small experiments at 1 Gb/s suggest that general-purpose computers and operating systems find it difficult to keep up when packet transmissions take single digits of microseconds and below. In this domain, it seems likely that intermediary software running as a user-level process will become a bottleneck if forced to handle every request. Even running a simple storage server such as the CORFU software implementation at speed becomes challenging (as we have learned from work-in-progress building a fast sequencer). Moreover, since processor cores are no longer getting much faster, the only path to better performance on a general-purpose architecture is to utilize more cores. However, coordinating multiple cores with the network and storage stacks such that everything is fully utilized is not easy.

Our work suggests that one viable way forward is for SSDs to be run in parallel, much as cores are on a multi-processor. With many SSDs in play, it makes sense to interconnect them with a network fabric to increase connectivity between clients and servers. And finally, as latencies decrease, storage platforms that are more specialized to the task at hand will likely win out, using techniques such as polling rather than interrupts, and with implementations close to, or in, hardware.

7. ACKNOWLEDGMENTS

We would like to thank Vijayan Prabhakaran for his contributions to the CORFU project as well as to this paper. Udi Wieder shared with us his expertise on Cuckoo Hashing. Paul Barham helped us understand the performance of the system. And we especially thank Chuck Thacker and his team for building the Beehive system.

8. REFERENCES

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX ATC*, 2008.

[2] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *SOSP*, 2009.

[3] ARM Ltd. Performance tab: ARM Cortex-A9 performance power & area. `http://www.arm.com/products/processors/cortex-a/cortex-a9.php`, 2012.

[4] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. Davis. CORFU: A shared log design for flash clusters. In *NSDI*, 2012.

[5] P. Bernstein, C. Reid, and S. Das. Hyder: A transactional record manager for shared flash. In *CIDR*, 2011.

[6] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. *Operating Systems Review*, 41(2):88–93, 2007.

[7] J. Davis, C. P. Thacker, and C. Chang. BEE3: Revitalizing computer architecture research. In *MSR-TR-2009-45*.

[8] Fusion-io, Inc. Fusion-io achieves more than nine million iops from a single iodrive2. `http://www.fusionio.com/press-releases`, 2013.

[9] J. Hartman and J. Ousterhout. The Zebra striped network file system. *ACM TOCS*, 13(3):274–310, 1995.

[10] R. Haskin, Y. Malachi, and G. Chan. Recovery management in QuickSilver. *ACM TOCS*, 6(1):82–108, 1988.

[11] P. Hunt, M. Konar, F. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX ATC*, 2010.

[12] Intel Corporation. Intel Solid-State Drive 320 Series. `http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-320-specification.pdf`, 2013.

[13] Intel Corporation. Intel X18-M/X25-M SATA Solid-State Drive - 34 nm Product Line. `http://download.intel.com/design/flash/nand/mainstream/Specification322296.pdf`, 2013.

[14] M. Ji, A. Veitch, J. Wilkes, et al. Seneca: Remote mirroring done write. In *USENIX ATC*, 2003.

[15] L. Lamport. The part-time parliament. *ACM TOCS*, 16:133–169, 1998.

[16] Microsoft Corporation. Registered Input/Output (RIO) API Extensions. `http://technet.microsoft.com/en-us/library/hh997032.aspx`, 2013.

[17] Microsoft Research. Beehive distribution for licensees. `http://research.microsoft.com/en-us/um/people/birrell/beehive/`.

[18] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat datacenter storage. In *OSDI*, 2012.

[19] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 41:122–144, 2004.

[20] Pure Storage, Inc. Pure Storage FlashArray FA-300 Series. `http://www.purestorage.com/pdf/Pure_Storage_FlashArray_Datasheet.pdf`, 2013.

[21] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM TOCS*, 10(1), 1992.

[22] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST*, 2002.

[23] F. Schmuck and J. Wylie. Experience with transactions in QuickSilver. *Operating Systems Review*, 25(5), 1991.

[24] M. Seltzer, K. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: A performance comparison. In *USENIX ATC*, 1995.

[25] A. Spector, R. Pausch, and G. Bruell. Camelot: A flexible, distributed transaction processing system. In *Compcon Spring'88*.

[26] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal. FreePDK: An open-source variation-aware design kit. In *IEEE International Conference on Microelectronic Systems Education*, 2007.

[27] J. Suzuki, T. Baba, Y. Hidaka, J. Higuchi, N. Kami, S. Uchida, M. Takahashi, T. Sugawara, and T. Yoshikawa. Adaptive memory system over Ethernet. In *HotStorage*, 2010.

[28] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, 2004.

[29] Violin Memory, Inc. 6000 Series Flash Memory Arrays. `http://www.violin-memory.com/products/6000-flash-memory-array/`, 2013.

[30] Xilinx. XUPV5-LX110T User Manual. `http://www.xilinx.com/univ/xupv5-lx110T-manual.htm`.

[31] J. Yang, D. B. Minturn, and F. Hady. When poll is better than interrupt. In *FAST*, 2012.