SDF: Software-Defined Flash for Web-Scale Internet Storage Systems

Jian Ouyang Shiding Lin
Baidu, Inc.
{ouyangjian, linshiding}@baidu.com

Song Jiang *
Peking University and
Wayne State University
sjiang@wayne.edu

Zhenyu Hou Yong Wang
Yuanzheng Wang
Baidu, Inc.
{houzhenyu, wangyong03,
wangyuanzheng}@baidu.com

Abstract

In the last several years hundreds of thousands of SSDs have been deployed in the data centers of Baidu, China's largest Internet search company. Currently only 40% or less of the raw bandwidth of the flash memory in the SSDs is delivered by the storage system to the applications. Moreover, because of space over-provisioning in the SSD to accommodate non-sequential or random writes, and additionally, parity coding across flash channels, typically only 50-70% of the raw capacity of a commodity SSD can be used for user data. Given the large scale of Baidu's data center, making the most effective use of its SSDs is of great importance. Specifically, we seek to maximize both bandwidth and usable capacity.

To achieve this goal we propose *software-defined flash* (SDF), a hardware/software co-designed storage system to maximally exploit the performance characteristics of flash memory in the context of our workloads. SDF exposes individual flash channels to the host software and eliminates space over-provisioning. The host software, given direct access to the raw flash channels of the SSD, can effectively organize its data and schedule its data access to better realize the SSD's raw performance potential.

Currently more than 3000 SDFs have been deployed in Baidu's storage system that supports its web page and image repository services. Our measurements show that SDF can deliver approximately 95% of the raw flash bandwidth and provide 99% of the flash capacity for user data. SDF increases I/O bandwidth by 300% and reduces per-GB hardware cost by 50% on average compared with the commodity-SSD-based system used at Baidu.

Categories and Subject Descriptors B.3.2 [*Memory Structures*]: Design Styles - mass storage (e.g., magnetic, optical, RAID)

Keywords Solid-State Drive (SSD), Flash Memory, Data Center.

1. Introduction

To accommodate ever-increasing demand on I/O performance in Internet data centers, flash-memory-based solid-state drives (SSDs) have been widely deployed for their high throughput and low latency. Baidu was one of the first large-scale Internet companies to widely adopt SSDs in their storage infrastructures and has installed more than 300,000 SSDs in its production system over the last seven years to support I/O requests from various sources including indexing services, online/offline key-value storage, table storage, an advertisement system, mySQL databases, and a content delivery network. Today SSDs are widely used in data centers, delivering one order of magnitude greater throughput, and two orders of magnitude greater input/output operations per second (IOPS), than conventional hard disks. Given SSD's much higher acquisition cost per unit capacity, achieving its full performance and storage potential is of particular importance, but we have determined that both raw bandwidth and raw storage capacity of commodity SSDs, in a range of performance categories, are substantially underutilized.

In the investigation of bandwidth utilization we chose three commodity SSDs—Intel 320, Huawei Gen3, and Memblaze Q520—as representative low-end, mid-range, and high-end SSD products currently on the market. The raw bandwidth of an SSD is obtained by multiplying its channel count, number of flash planes in each channel, and each plane's bandwidth. The raw bandwidth of each plane is mainly determined by the flash feature size and was dis-

* This work was performed during his visiting professorship at Peking University.

ASPLOS '14, March 01 - 05 2014, Salt Lake City, UT, USA.

http://dx.doi.org/10.1145/2541940.2541959

SSD Type	Interface	NAND Type	Channels	Planes/ channel	NAND speed	Raw R/W Bandwidths (MB/s)	Measured R/W Bandwidths (MB/s)
Low-end	SATA 2.0	25nm MLC	10	4	ONFI 2.x	300/300	219/153
Mid-range	PCIe 1.1 x8	25nm MLC	44	4	40MHz async	1600/950	1200/460
High-end	PCIe 1.1 x8	34nm MLC	32	16	ONFI1.x async	1600/1500	1300/620

Table 1. Specifications and bandwidths of different types of SSDs. "NAND speed" specifies the NAND flash interface, which indicates the flash speed.

closed to us by the respective vendors. To obtain a device's peak read and write bandwidths we read or write data sequentially in erase-block units. Table 1 lists specifications and read/write bandwidths of the SSDs with 20% over-provisioning. As shown, the measured bandwidths range from 73% to 81% for read, and 41% to 51% for write, of the corresponding raw bandwidths. Interestingly, these ratios are relatively constant ranging from the low-end drive with the SATA 2.0 interface with maximum transfer rate of 300 MB/s, to the mid-range and high-end drives that cost 4-8 times more per unit capacity and use the PCIe 1.1 x8 interface with maximum transfer rate of 1.6GB/s. In practice the effective bandwidth received at the storage system serving real-world workloads can be much lower because I/O requests from upper-level software, such as file systems, databases, and key-value stores, are usually not optimized to generate SSD-friendly access patterns. The consequence is that the realized I/O bandwidth of Baidu's storage system is only 50% of the raw bandwidth of the flash hardware.

Another challenge in Baidu's web-scale storage system is the large amount of reserved flash space unavailable for storing user data. Space is reserved for two purposes: storing parity coding across channels for data loss protection, which accounts for approximately 10% of the flash space, and over-provisioning for accommodating random writes. In the conventional SSD architecture the amount of over-provisioned space can be critical to performance. Figure 1 shows random write throughput as a function of the over-provisioning ratio for the low-end SSD that has been widely deployed in Baidu's data centers. Though 50% over-provisioning seems to be excessive, 25% over-provisioning may be justified in this SSD architecture because it can improve throughput by 21% relative to 7% over-provisioning and by more than 400% relative to 0% over-provisioning. In our production system the over-provisioning is usually configured between 10% and 40% depending on the workload characteristics and performance requirements. For a workload with mixed random writes and sequential reads, raising the ratio from 22% to 30% can result in an increase of sustained throughput from less than 100MB/s to 400MB/s (more than 400%) because the read throughput can be greatly degraded by random writes triggering more frequent garbage collection when there is less over-provisioning.

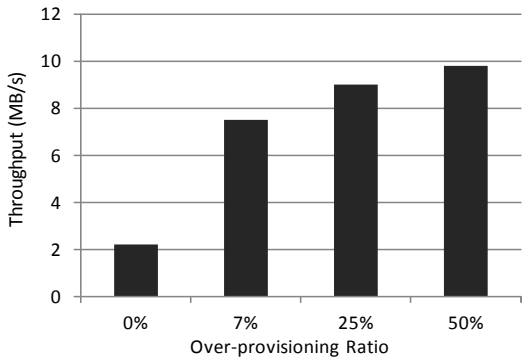

Figure 1. Throughput of the low-end SSD (Intel 320) with random writes of 4KB pages when various fractions of its raw capacity are allocated as over-provisioned space.

Significant under-utilization of resources in a large data center with hundreds of thousands of servers can have severe implications in terms of both initial and recurring costs. First, the hardware cost increases substantially because additional device acquisition is required to make up for unexploited performance and capacity potential. Second, the energy cost associated with additional hardware, such as servers and cooling equipment, increases. Third, physical space requirement increases. Finally, all of these contribute to increased maintenance costs.

In building a large-scale data center on a limited budget the common practice at major Internet companies such as Google, Facebook, and Baidu has been to use a large number of inexpensive commodity parts and servers that collectively meet the performance and storage requirements. With increasing performance requirements the system is expanded by adding more parts and servers and consequently pushing hardware, energy, and maintenance costs ever higher. A more economical strategy is to better exploit existing (and future) resources wherever possible. This is especially effective for devices that are relatively expensive and whose potential is significantly under-exploited, such as the SSD. However, current software, including system software, usually has been extensively optimized for managing the devices and extracting their maximum potentials as far as their standard interfaces allow. To unlock their full potential the devices need to be modified to provide an interface such that the software can more directly access the devices' in-

ternal operations, and interact with the devices in a manner more friendly to their performance characteristics. This customization has a cost barrier because new design and manufacturing are needed. However, if the device is deployed in large numbers (100,000+), the additional cost could be easily recouped. Accordingly, we should rethink the design and implementation of SSDs used in a data center with its particular workload characteristics as opposed to general-purpose use in, e.g., desktop and portable devices.

Baidu's *Software Defined Flash* (SDF) project is such an effort. By exposing the channels in commodity SSD hardware to software and requiring the write unit to be of the flash erase-block size, we can match the workload concurrency with the hardware parallelism provided by the multiple channels and minimize the interference in a flash channel serving requests from different threads, thereby exploiting individual flash's raw bandwidth. SDF requires the software to explicitly initiate block erasure operations, and requires the software erasure of a block before writing into it so that the drive does not need to reserve space for internal garbage collection. At the same time the system software can schedule the erasures during a flash's idle period to minimize their effect on the service quality of regular requests. In addition, we remove the parity coding across channels and, to minimize the possibility of data loss, rely on the system-level data replication that is already in place for data persistence. Together these make almost all raw flash space available for storing user data.

In addition to the concerns about the performance and cost of SSD, the SDF project is also motivated and enabled by a significant workload characteristic in data centers supporting Internet-wide services. Specifically, most write requests are sequential and are issued in a unit much larger than what conventional block devices assume (e.g., 4KB). At one extreme, write requests by some major services can be naturally sequential. For example, there are over 10,000 servers at Baidu for generating a large volume of index data that is not subject to later modification, and sequentially writing it to storage. At the other extreme, because serving small write requests can be unacceptably slow and subsequent reads can be expensive due to an unmanageably large body of metadata in a conventional file system, a common practice is to direct small-write traffic to a storage system managed as a log-structured merge (LSM) tree [16] such as Google's BigTable or Baidu's CCDB. In such systems writes are served in a unit as large as several megabytes. As a consequence, imposing a write unit of erase-block size in SDF does not demand significant software changes to accommodate the device's new interface.

In summary, in this paper we make three major contributions:

- We propose SDF (Software Defined Flash), an SSD device with a customized architecture and interface supported by an FPGA-based controller, to allow the software to realize its raw bandwidth and storage capacity. This represents an effort to customize hardware to specifically meet the requirements of the data center with respect to system performance, cost, and maintenance, and is distinct from today's common practice of relying only on the large number of commodity devices for system capability.
- To ensure that the raw bandwidth and space are fully delivered to the applications, we deploy SDF mainly to serve a major class of workloads involving large writes, including dumping sequential index files into storage and accessing storage for LSM-tree-style data management. We demonstrate that for these major applications the potential I/O performance can be attained with unsophisticated request scheduling strategies. Because of the high concurrency exhibited by the workloads the exposed multiple channels in SDF can be effectively exploited for much higher bandwidth. In this sense SDF is motivated by, and receives its name from, the software-shaped workloads.
- We have implemented the SDF controller using the Xilinx Spartan-6 FPGA in place of the commodity SSD flash translation layer (FTL), and using the Xilinx virtex-5 to implement its PCIe interface. SDF is installed in more than 3000 servers now running in Baidu's data center and supporting the web page and image repository services. Measurements on the production system show that the SDF can deliver up to 95% of the raw bandwidth while the commodity SSDs with a similar configuration can only deliver less than 50% on these real-world workloads, and that the bandwidth is comparable to that of the high-end PCIe-interfaced SSDs. In addition, SDF provides 99% of the raw flash capacity for user data, and its per-GB cost is about 50% less than that of high-end commodity SSDs used at Baidu.

The remainder of this paper is organized as follows. Section 2 describes the SDF design, including the hardware and interface designs. Section 3 presents and analyzes experimental results. Related work is presented in Section 4 and Section 5 concludes.

2. Design of SDF

Each SDF consists of a customized flash device wrapped with a layer of software providing an asymmetric block access interface (8 MB write/erase unit and 8 KB read unit). While the device provides multiple independent channels for parallel access, it is the software layer that makes this functionality accessible to applications and translates the hardware parallelism into significantly improved throughput delivered to the applications. Next we describe SDF's hardware structure including its FPGA-based controller design, design choices to reduce hardware cost, the design of the SDF interface, and the use of SDF in Baidu's storage system.

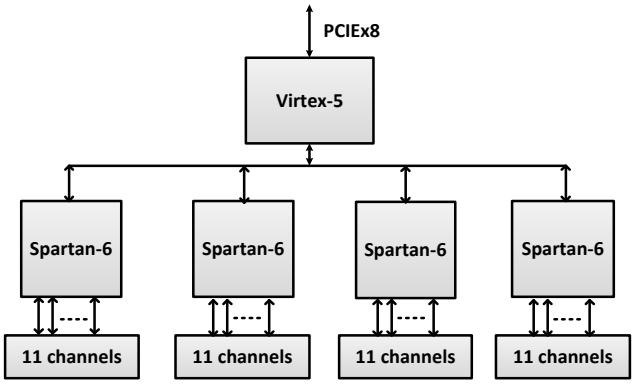

Figure 2. The hardware architecture of SDF.

Figure 3. Photograph of the SDF board.

2.1 SDF Hardware Structure

The SDF controller is composed of five FPGAs. A Xilinx Virtex-5 FPGA is used as a data path controller to implement PCIe DMA and chip-to-chip bridging as shown in Figure 2. It connects to four Xilinx Spartan-6 FPGAs. Each Spartan-6 FPGA implements an independent flash translation layer (FTL) for each of its 11 channels. Each of the 44 channels controls two Micron 25nm MLC flash chips—mainstream NAND chips currently used in a wide range of commodity SSDs. Each flash chip has two planes supporting parallel access, and each chip has 8 GB capacity giving the SDF device a raw capacity of 704GB.

To expose individual flash channels as independent devices to applications, each flash channel has a dedicated channel engine providing FTL functionality including block-level address mapping, dynamic wear leveling, and bad block management, as well as logic for the flash data path. The block mapping table and erase count table for dynamic wear leveling are stored in an on-chip SRAM exclusively accessed by the FPGA. A lookup into the block mapping table requires only one clock cycle. The SRAM memory consists of four SRAM banks that can be independently accessed. Because the operation on the erase-count table is a time-consuming search for the smallest count value, we distribute the table across the banks to allow a parallel search. Figure 3 is a photograph of an SDF hardware board.

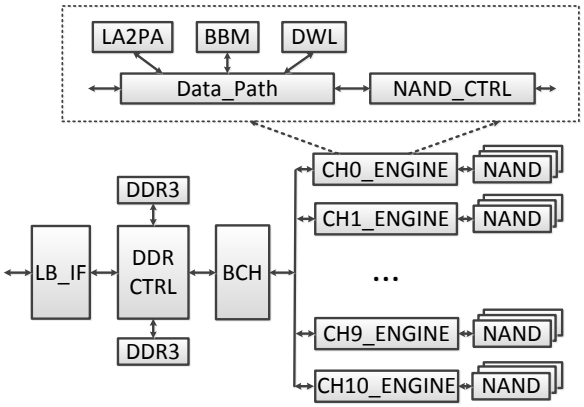

Figure 4. Logic diagram of a Spartan-6 FPGA for accessing a group of 11 channels in an SDF.

Figure 4 shows the logic diagram of the Spartan-6 FPGA that manages the data path between the NAND flash and the SDF interface. As shown, the local bus interface ("LB_IF") is a chip-to-chip bus connecting to the Virtex-5 FPGA chip-to-chip bridge. The bandwidth of the chip-to-chip bus is 16 times that available from each of the 11 channel engines. To fully utilize the chip-to-chip bandwidth all the channel engines share the same high-bandwidth data path, and data into and out of each channel is buffered in a DRAM to give more leeway in data transmission scheduling. Specifically, we have two 512 MB DDR3 16-bit DRAMs running at 533 MHz under the management of a DDR memory controller between the LB_IF and the channel engine. In each channel's dedicated controller ("CH_ENGINE") there are modules for translating logical addresses to physical addresses ("LA2PA"), bad block management ("BBM"), and dynamic wear leveling ("DWL"). To support applications demanding high IOPS, SDF adopts an interrupt reduction technique to reduce CPU load and thereby improve IOPS. SDF merges interrupts for channel engines in the same Spartan-6 and then merges interrupts from all of the Spartan-6s in the Virtex-5. With this merging, the rate of interrupts is only 1/5 to 1/4 of the maximum IOPS.

In the SDF, 59% of the logic area of each Spartan-6 is used to implement its functionalities, of which 42% (25% of total) is used for BCH codec for individual flash chips. The Huawei Gen3 is a conventional SSD whose structure is the same as that of SDF as described in Figure 2 and so can be considered the predecessor of SDF. In comparison, the Huawei Gen3 uses 70% of the logic area of the Spartan-6. By removing logic for garbage collection, inter-chip parity coding, and static wear leveling, SDF reduces the use of logic space on the Spartan-6 FPGAs by 16%. The remaining FPGA logic capacity in our design can be used for "computing in storage" as demonstrated elsewhere [17].

2.2 Reducing Hardware Cost

We have expended significant effort to reduce the hardware cost of the SDF device by including features and functionalities only when they are proven truly necessary in the environment for which they were designed. This is in contrast with the design of commodity SSD intended for serving various workloads covering a large spectrum of characteristics in terms of request operation (read or write), request size, and request locality. In particular, because of flash's out-of-place data update, in the design of commodity SSD substantial attention has been paid to the optimization of logic for effectively serving small writes, which are associated with expensive garbage collection operations and carry a significant performance implication. In favor of reduced cost, streamlined design, and much improved performance over versatility, we omit a number of features implemented by conventional SSDs from the SDF design. This simplification also enabled a complete production-quality design in only seven months by two engineers. These simplifications and their motivation are detailed following.

The capacity provided by the SDFs is small relative to the total storage requirement, so the SDFs are mainly used as cache for hard disks. For this reason, data that is rarely accessed is not expected to reside in the cache for a long period of time, so the SDF does not conduct static wear leveling wherein blocks that have experienced relatively few writes have their data migrated elsewhere within the device to free them for future writes. This not only simplifies the hardware design but also reduces the performance variation caused by the associated sporadic data movement.

The SDF hardware does not include a DRAM cache. Conventional SSDs usually include a large DRAM cache for reducing access latency. However, with DRAM cache data consistency can be compromised by a power failure so a battery or capacitor must be employed to prevent data loss. The cache and the battery (or capacitor) add to the hardware cost. In Baidu's storage infrastructure, recently accessed data have been cached in the host memory, and the access patterns experienced by a storage device are not expected to have such strong temporal locality that reused data is likely to be retained in an SSD's internal DRAM cache. Therefore SDF does not provide such a cache at all to save cost. Write requests are acknowledged only after the data is actually stored on the flash so all writes in SDF are effectively synchronous operations.

SSDs consist of multiple NAND flash chips, and conventional SSDs provide data protection by using RAID 5 parity coding across the flash chips in addition to powerful BCH ECC protection within individual flash chips for fault detection and correction with attendant cost, complexity, and reduced flash space available for user data. However, in our large-scale Internet service infrastructure, data reliability is provided by data replication across multiple racks. Therefore SDF excludes the parity-based data protection and relies on BCH ECC and software-managed data replication. Our experience has shown that even without the parity-based protection, failures reported by SDF are rare in our system: during the six months since over 2000 704GB SDFs were deployed in Baidu's storage system there has been only one data error that could not be corrected by BCH ECC on the flash chips and had to be reported to the software for data recovery. Besides the benefits of smaller controller logic and removed performance penalty, elimination of the parity code also saves substantial space. For example, assuming that every 10 flash channels are protected by one flash channel storing parity in a conventional SSD, SDF increases the usable flash capacity by 10% relative to conventional practice. In Baidu's previous SSD-based storage system, over-provisioning was usually about 25% to maintain sustained throughput for the regular workloads. However, when the SSD was almost full and served very intensive writes, it was sometimes necessary to raise over-provisioning to 40%. By removing the over-provisioned space and other hardware costs, SDF achieves 20% to 50% cost reduction per unit capacity, mainly as a function of the amount of over-provisioning in systems used for comparison. In Baidu's CCDB storage systems, where most SDFs are currently employed to support performance-critical services such as online construction of inverted indices for hot webpages, the cost reduction is around 50% after eliminating the need of having 40% over-provisioning space in its SSD devices.

2.3 SDF Interface Design

A major concern in the effort to tap the raw performance of SSD is the cost of write amplification and its associated cost for garbage collection operations. In flash data cannot be over-written in place. Instead, logically over-written data is first invalidated and the space later reclaimed by garbage collection. Then the data needs to be erased before the space can be re-used for writing new data. There is an asymmetry in the flash's read/write/erase operations, where read and write are in the unit of page size, usually 8 KB, and erase is in the unit of block size, usually 2 MB, i.e., 256 times larger. To reclaim a block, the FTL must copy all of the valid data in a block to an erased block before the erase operation can be carried out, resulting in amplified write operations. It is well known that amplified writes, though necessary with the asymmetry between write and erase units, can significantly degrade the device's performance and demand a substantial portion of the device's raw space to accommodate out-of-place writes to achieve acceptable performance, as well as reduce the life of the flash by the amplification factor. This effect is more strongly exhibited when a workload contains a large number of small random writes. To alleviate these negative effects on SSD's performance and longevity we make three optimizations in the SDF interface, as follows.

First, we expose the asymmetry in the read/write/erase operations to the software. Current SSDs provide a symmetric interface, where both read and write use the same unit

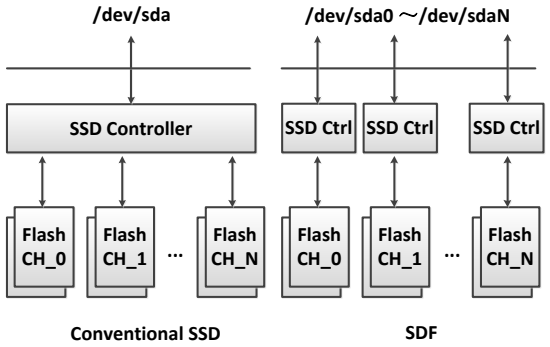

Figure 5. Comparison of (a) conventional SSD's structure (b) and SDF's structure.

size, which is one or a few flash pages (usually 4 KB). We keep the small read unit but greatly increase the write unit size to be a multiple of the flash erase block size and require write addresses to be block-aligned. Thus write amplification is eliminated because no flash block can contain both valid and invalided data pages at the time of garbage collection.

Second, we expose the device's internal parallelism to the workloads. As mentioned, each of the 44 channels of an SDF has its own controller or FTL engine. As shown in Figure 5, unlike a conventional SSD where there is only one controller for the entire device, and the device is software mounted as a single logical device (such as */dev/sda*—partitioning a conventional SSD is only notional), SDF presents each channel to the applications as an independent device, such as */dev/sda0* through */dev/sda43*. This exposure of parallelism is an innovation in maintaining high service concurrency to exploit the device's raw bandwidth without losing the benefit of writing in the erase-block unit in individual flash chips for minimal garbage collection overhead. In a conventional SSD service concurrency is maintained by striping data across the channels so that one request can be served by multiple channels. However, using this approach to achieving high bandwidth is in conflict with the goal of eliminating write amplification, which requires writes into individual flash chips to be in the unit of erase block units. There are two flash chips, or four planes, in a channel, and each channel needs an 8 MB write in service of one request to exploit the in-channel parallelism. To have high service concurrency in the whole device, a request would have to be excessively long (tens of MB). This requirement on workloads would be too demanding and unlikely to be met in a data center environment. By exposing SDF's 44 channels individually to the software we can leave the task of exploiting the parallelism on the channels to the software, which can employ multiple threads to concurrently issue requests to different channels. To exploit the parallelism (four flash planes) within one channel, we set SDF's write unit at 8 MB and data are striped over the flash chips in a channel with a stripe size of 2 MB, the per-chip erase unit size.

Third, we expose the erase operation as a new command to the device that is invoked by the software. The erase operation is much more expensive than read or write: for example, erasing a 2 MB block takes approximately 3 ms. When an erase operation is in process in a channel the servicing of regular requests to the channel can be significantly delayed. Erase operations scheduled by an SSD's internal logic at times unknown to the applications can cause unpredictable service quality variation, which is especially problematic for performance-critical workloads. Furthermore, if the erase operation were not exposed, SDF would still need to over-provision space to accommodate out-of-place overwriting of data. To be ready to quickly serve bursts of writes, SDF would have to provision a high percentage of raw space for this purpose. By providing applications the erase command, SDF does not need to reserve any space for out-of-place writes and garbage collection. Instead, SDF exposes all of its raw space to the software and moves the responsibility for garbage collection to the software. Either programs or an I/O scheduler are required to conduct an erase operation on a block before it can write data into it. With the software in control of the garbage collection operation it can schedule these operations in the background, or at lower priority than the servicing of high-priority requests.

2.4 SDF in Baidu's Storage System

Baidu's data centers store hundreds of petabytes of data with a daily data processing volume reaching dozens of petabytes. With an ever-increasing performance requirement on the storage system, hard disks, and even conventional SSDs, are becoming inadequate. SDF represents Baidu's latest effort to increase performance. There are different types of data in Baidu's storage systems, including web pages and their indices, images, data in cloud storage, and various system log data. For operational convenience and efficiency, applications prefer to present and access different data to the storage system in a variety of formats including database tables, files in a file system, or simple key-value pairs. Accordingly, the storage system includes three data management subsystems, namely *Table*, *FS*, and *KV*, to serve the aforementioned data formats. Internally all three are treated as key-value pairs. In the *Table* system, the key is the index of a table row, and the value is the remaining fields of the row. In the *FS* system, the path name of a file is the key and the data or a segment of data of the file is the value.

Each of the subsystems is hosted on a cluster of storage servers. According to their keys, requests from clients are hashed into different hash buckets called *slices* in our system. A slice uses Baidu's CCDB system to manage its KV pairs using a log-structured merge (LSM) tree [16]. Similarly to Google's BigTable [5], CCDB uses a container for receiving KV items arriving in write requests. The container has a maximum capacity of 8 MB. Data that are being accu-

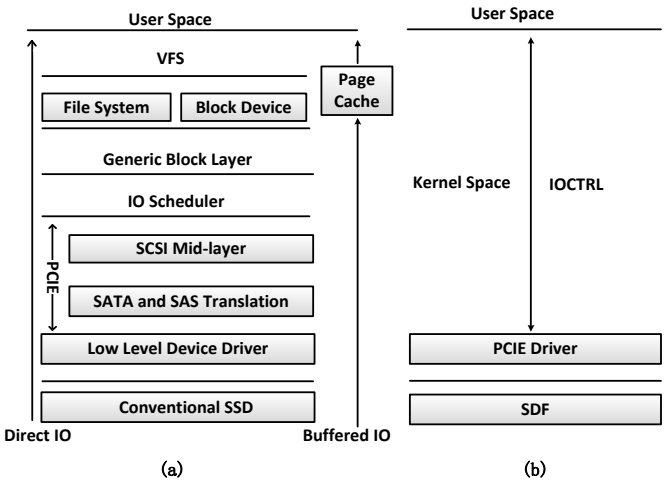

Figure 6. Comparison of (a) conventional SSD's I/O stack and (b) SDF's I/O stack supporting its block interfaces.

mulated in the in-memory container are immediately saved in a log in an SSD or a hard disk to prevent data loss. When a container is full, a *patch* is formed, and the patch is written into the SDF device on the server. Patches on the storage experience multiple merge-sorts, or multiple reads and writes before they are placed in the final large log. All write requests in CCDB are to store patches to the SDF and match the 8 MB SDF write unit. By managing data in the LSM tree, all metadata of the KV pairs can be accommodated in DRAM memory, and a read request from clients can be served with only one SDF read.

Underneath the slice abstraction is a unified user-space block layer for processing and passing I/O requests from the slices to the SDF device. Currently this layer dictates a fixed 8 MB write size and requires that each write request arrive with a 128-bit ID. At this time only the low 64 bits of an ID is used to uniquely identify a write or the block of data involved in a write. A client requests the 64-bit part of an ID, to serve as a key, from a server that maintains a counter for generating unique IDs. The block layer uniformly hashes the ID to determine a channel to serve the write. Specifically, blocks with consecutive IDs are written to the 44 channels in a round-robin fashion. This arrangement abstracts the tasks of managing storage space, such as determining which channels to write, what physical spaces have been erased and are ready for receiving new data, and what spaces are not. In future work we will introduce more intelligent scheduling policies to balance the load across the channels should a skewed workload occur, and coordinate timings for the SDF to serve different types of requests so that on-demand reads take priority over writes and erasures.

For Google's BigTable, a distributed file system (GFS [9]) is responsible for its I/O requests as well as data replication for reliability. For Baidu's CCDB, data is replicated over slices. Since slices reside on individual server nodes, a question arises of whether we should retain the local file system between the slices and the SDF. A general-purpose operating system such as Linux, as well as the file system part of it, performs functionalities such as access authorization, name parsing, address space protection between different users, and data caching and prefetching. On our servers operations in this software layer impose as much as 12 μs additional time spent on a single I/O request. This overhead can be substantial relative to the high-speed flash data access. Currently SDF is used exclusively by the CCDB data management software on a local server, so most of the file system's functionalities are unnecessary. For efficiency, we open the device as a raw device file and use the IOCTRL interface to largely bypass the kernel and directly access its low-level PCIE driver. Some basic file system functionalities, such as block allocation and logical-to-physical address translation, are implemented in the block layer. A comparison of the I/O stacks of SDF and a conventional SSD is depicted in Figure 6. The resulting latency of SDF's software stack is only about 2-4 μs, which is mostly used for handling message signaled interrupts (MSIs) from the PCIe interface.

CPU	Intel E5620x2 2.4 GHz
Memory	32 GB
OS	Linux 2.6.32 kernel
NIC	Intel 82599

Table 2. Client and server node configuration.

3. Performance Evaluation

In this section we evaluate SDF's performance in Baidu's storage infrastructure. Currently there are more than 3000 SDFs deployed in the production system. The SDFs have been in use for over six months and a large amount of performance data has been collected from the monitoring and profiling system. We first describe the experimental setup, then the evaluation results for both microbenchmarks and Baidu's production workloads.

3.1 Experimental Setup

For the experiments we set up a number of client nodes to send KV read and write requests to a server node. The client and server nodes have the same configuration as described in Table 2, and are in the same cluster. The server is connected to a switch by two 10 Gbps NICs, and the clients each use one 10 Gbps NIC. The SDF is installed on the server, and its specification is given in Table 3. The specification of the Huawei Gen3, the predecessor of SDF, is the same. In the evaluation we use the Huawei Gen3 and the SATA-based Intel 320 for comparison. We will focus our comparison primarily on the Huawei Gen3 to examine the performance implications of SDF's architectural changes. In the experiment we leave 25% of the raw capacity of the Huawei Gen3 as over-provisioned space for internal use such as garbage col-

Host Interface	PCIe 1.1x8
Channel Count	44
Flash Chips per Channel	2
Planes per Flash Chip	2
Channel Interface	Asynchronous 40 MHz
Flash Capacity per Channel	16 GB
NAND Type	25 nm MLC
Page Size	8 KB
Block Size	2 MB

Table 3. Device configuration for both the Baidu SDF and the Huawei Gen3.

lection. In the Huawei Gen3 data is striped over its 44 channels with a striping unit size of 8 KB. The Intel 320 uses 25 nm MLC NAND flash and the SATA 2.0 interface. The Intel 320 has 10 channels, and 20 GB (12.5%) of its raw capacity of 160 GB is reserved for internal use including garbage collection.

3.2 Experiments with Microbenchmarks

SDF is designed to deliver I/O service with high throughput and low latency to applications with high concurrency. To reveal the storage devices' performance, in the evaluation we first generate synthetic workloads on the server. For SDF we send requests to the SDF as a raw device, bypassing the kernel I/O stack. For the Intel and Huawei SSDs the requests are sent to the respective devices via the Linux I/O stack.

In the experiments we first run microbenchmarks that issue random read requests of different sizes (8 KB, 16 KB, and 64 KB) and 8 MB write requests to each of the three devices. For SDF we use 44 threads—one for each channel—to exploit its hardware parallelism. For the other two devices only one thread is used because they expose only one channel, and the thread issues asynchronous requests. For SDF all requests are synchronously issued and the benchmarks issue requests as rapidly as possible to keep all channels busy. The results are reported in Table 4. Because SDF does not allow writes smaller than 8 MB we use only 8 MB writes in the test.

To put SDF's performance in perspective we need to know the throughout limits imposed by the PCIe interface and the flash itself. The maximum PCIe throughputs when it is used for data read and write are 1.61 GB/s and 1.40 GB/s, respectively, and SDF's aggregate flash raw read/write bandwidths are 1.67 GB/s and 1.01 GB/s, respectively. From the table we can see that SDF enables throughput close to the architectural limits: for read its 8 MB-request throughout of 1.59 GB/s is 99% of the bandwidth limit imposed by the PCIe interface, and for write, 0.96 GB/s throughout is 94% of the flash's raw bandwidth. Even for reads with requests as small as 8 KB the throughout is 76% or more of the PCIe bandwidth limit. The expensive erasure operation can reach a 40 GB/s throughput in SDF (not shown in the table).

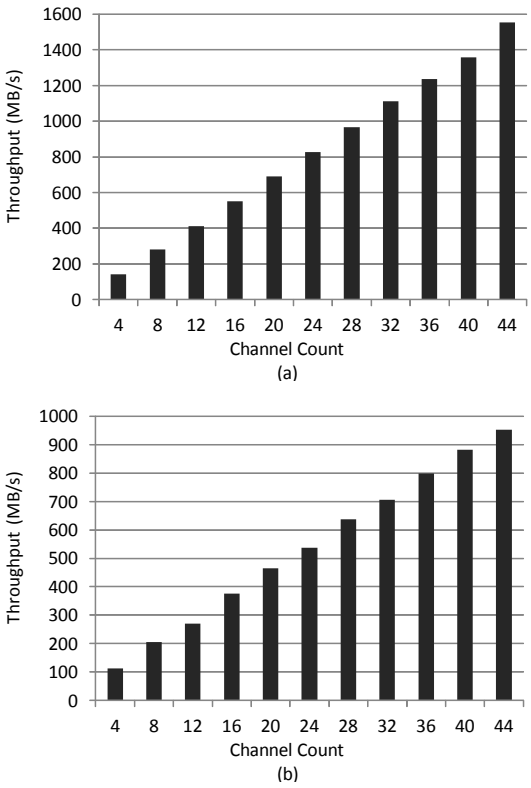

Figure 7. Throughputs of SDF with sequential reads (a) and writes (b) when different numbers of channels are used.

Though the Huawei Gen3 has the same number of channels, type and count of flash chips, and FPGA controller hardware, its throughput is consistently lower than that of SDF. In particular, with a request size of 64 KB or smaller its throughput is substantially lower. In SDF, as long as a request is not larger than 8 MB it is serviced by one channel. In contrast, in the Huawei Gen3 the logical address space is striped over 44 channels with an 8 KB striping unit. A large request is split into multiple sub-requests, which are serviced in different channels. Accordingly, requested data has to be split (for write) or merged (for read) in the request service, and each channel serves a larger number of smaller requests. This adds to the overhead and reduces throughput.

We also experimented with the SDF when only limited numbers of channels are concurrently accessed, still with each channel receiving requests from a dedicated thread. Figures 7(a) and 7(b) show the throughput of the SDF with sequential 8 MB read and write requests when different numbers of channels are employed. As shown, the throughput increases almost linearly with the number of channels before the PCIe's bandwidth limit or the flash's raw bandwidth is reached, demonstrating that the SDF architecture scales well.

Because performance predictability can be important in a data center, especially for on-line service, we evaluate latency variation for write requests on the Huawei Gen3 and

Operations	8 KB Read	16 KB Read	64 KB Read	8 MB Read	8 MB Write
Baidu SDF (GB/s)	1.23	1.42	1.51	1.59	0.96
Huawei Gen3 (GB/s)	0.92	1.02	1.15	1.20	0.67
Intel 320 (GB/s)	0.17	0.20	0.22	0.22	0.13

Table 4. Throughputs of the Baidu SDF, Huawei Gen3, and Intel 320 devices with different request sizes for read and write operations. Because the SDF's write unit is 8 MB, only write throughputs for 8 MB are reported.

.

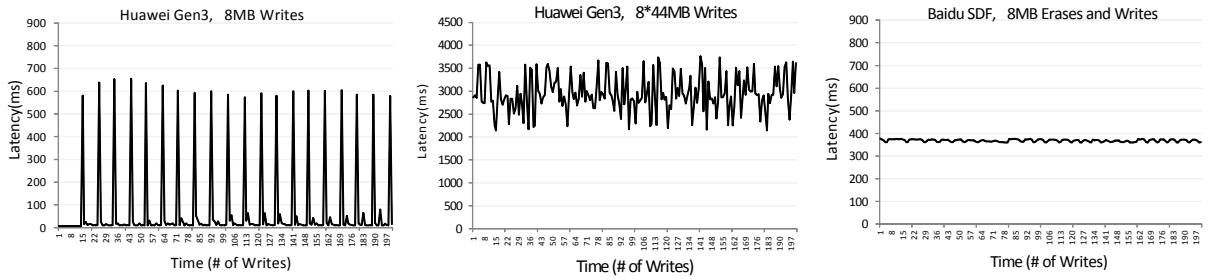

Figure 8. Latencies of write requests on the Huawei Gen3 and the Baidu SDF when writes with an aggregate total of 28 GB data are in progress. The Huawei Gen3 serves write requests of either 8 MB or 352 MB. The Baidu SDF serves write requests of 8 MB simultaneously on its 44 channels. By design these devices were almost full at the beginning of the experiments.

the Baidu SDF. Note that for SDF an erase operation is explicitly performed immediately before a write, so we include the erase time in its write latency as reported in Figure 8. We first send 8 MB write requests to the Huawei Gen3 and the SDF. As shown in Figure 8 the latency of the Huawei Gen3 varies widely—between 7 ms and 650 ms—with an average of 73 ms. In contrast, SDF's latency is 383 ms with little variation. The Huawei Gen3 has a 1GB on-board DRAM buffer and a hit in the buffer produces very low latencies. However, this buffer has been removed in SDF. By performing erase operations explicitly before individual writes rather than conducting bursty garbage collection, SDF provides consistent write performance. This makes SDF a competitive device for application demanding high performance predictability. If we increase the size of requests to the Huawei Gen3 to 352 MB so that each of its 44 channels accesses 8 MB data, the latency variance is reduced to 25% of average request latency (2.94 seconds). However, the requirement of such large requests is impractical for most applications.

3.3 Experiments on the Production System

Currently the deployed SDFs are supporting web page services on the production system. The crawler collects web pages from the Internet and stores them in the web page repository on a storage system. Web pages in the repository will undergo various processing steps including extraction of their abstracts, analysis of the link relationship among pages, and construction of a reverse index to feed into index servers. As showed in Figure 9, the web page repository is the central component of the search engine. We use the CCDB's Table system to process the web page repository.

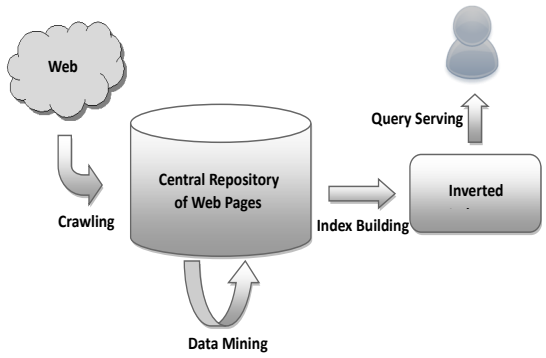

Figure 9. Building and indexing in a Web page repository.

In the system each storage server hosts one or multiple slices. A slice provides storage and access service in a LSM-tree fashion for KV pairs whose keys fall into a specific range. In the experiments each slice is always loaded with requests from a single client, each client continuously sends synchronous read/write KV requests to one slice by setting its request keys within the slice's designated key range. A client sends a request immediately after it receives the response to its previous request. To improve network and server efficiency requests can be batched, that is, one request may contain multiple read/write sub-requests, each reading or writing one KV pair. The number of sub-requests contained in a request is called the request's *batch size*.

3.3.1 Random Reads

We first let the clients issue 512 KB random read requests to one or a number of slices. That is, each requested value is of 512 KB and the key is randomly and uniformly selected

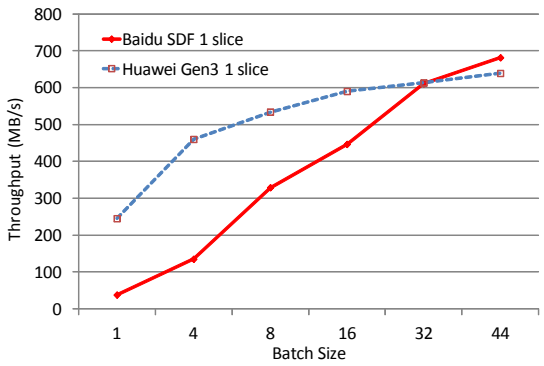

Figure 10. Throughputs of one storage node with Baidu SDF and Huawei Gen3 with one slice with varying batch size. Random 512 KB read requests are used in this experiment.

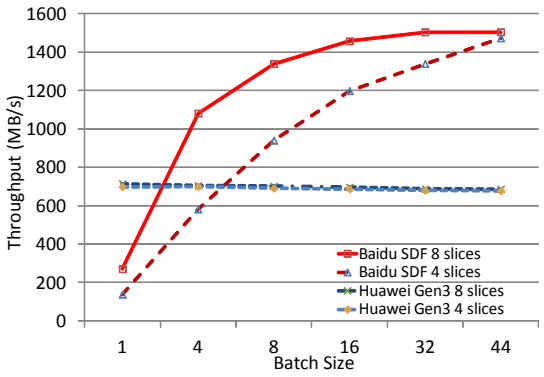

Figure 11. Throughputs of one storage node with the Baidu SDF and the Huawei Gen3 when running four or eight slices with various batch sizes. Random 512 KB read requests were used in the experiment.

from a slice's key range. Figure 10 shows the throughputs with different batch sizes when only one slice is used with the Baidu SDF and the Huawei Gen3. As shown, SDF's throughput lags behind the Huawei Gen3 until the batch size reaches 32. SDF's performance advantage with high channel concurrency cannot be realized with synchronous requests and only one slice. For example, when the batch size is one, requests are served at one channel at a time in SDF, delivering only 38 MB/s throughput. With increasing batch size, different sub-requests are more likely to be served at different channels in parallel and the throughput steadily increases. However, for the Huawei Gen3 throughput with a batch size of only one is 245 MB/s, much greater than its counterpart for SDF. The FTL of the Huawei Gen3 maps contiguous logical addresses across the channels in a round-robin manner with a striping unit of only 8 KB. The requested 512 KB data are well distributed over all 44 channels and service of even one request can be fully parallelized. In contrast, in SDF with only one slice the service parallelism is limited by requests' batch sizes because each request is served by only one channel.

Another interesting observation from Figure 10 is that the Huawei Gen3 still has higher throughput than SDF when the batch size is substantially large (up to 32). SDF exploits the parallelism among sub-requests in a synchronous request, while the Huawei Gen3 exploits parallelism within one sub-request, so the Huawei Gen3 can service one sub-request faster than SDF. Once it is done with a sub-request, it can send the data back to the client at the same time that it is serving the next sub-request. In this scenario the SDF does not have the efficiency achieved by the Huawei Gen3 with its pipelined data access because SDF completes the service of all sub-requests in a request at almost the same time.

If multiple slices are used in a server the concurrency of requests from different slices would increase SDF's throughput by keeping more channels busy at the same time. Figure 11 shows the throughput for four and eight slices. As shown, when the batch size is one, for SDF only a few of its 44 channels are in parallel service (at most 4 or 8 channels when 4 or 8 slices are used, respectively). In contrast, with 8 KB striping these numbers of slices are sufficient to keep all of the channels in the Huawei Gen3 fully occupied. Accordingly, it's no surprise that the Huawei Gen3 has greater throughput than SDF. However, with increasing batch size SDF's throughput correspondingly increases. For example, with a batch size of 4, the 8-slice throughput increases by almost four times, from 270 MB/s to 1081 MB/s. When the batch size is sufficiently large the increase in throughput of SDF seems to flatten as all channels become busy. Interestingly, the increase does not stop when the product of slice count and batch size reaches the channel count (44). With round-robin address mapping the random requests cannot be evenly distributed over the channels when the request count is only slightly larger than the channel count. That is, more concurrent requests are needed to have a better balanced load. In future work a load-balance-aware scheduler will be employed to help SDF reach its peak throughput (approximately 1.5 GB/s) with a smaller number of concurrent requests. Though with a small number of concurrent requests the Huawei Gen3 can achieve high throughput (approximately 700 MB/s), further increasing the concurrency does not further increase the throughput. To the contrary, the throughput actually decreases slightly with higher concurrency. With 512 KB requests and an 8 KB data striping unit on 44 channels, each channel is accessed for only 8 KB or 16 KB data in the service of one request, leading to very high access concurrency. Accordingly, the scheduling overhead may increase and the service time of unsynchronized requests at different channels may increase some requests' service time. In addition, the curves for the 4-slice and 8-slice throughputs of the Huawei Gen3 are nearly coincident, indicating that the increased concurrency cannot be taken advantage of.

To see how request size affects throughput, in Figure 12 we plot the throughput with different request sizes (32 KB, 128 KB, and 512 KB) and different slice counts (1, 4, and

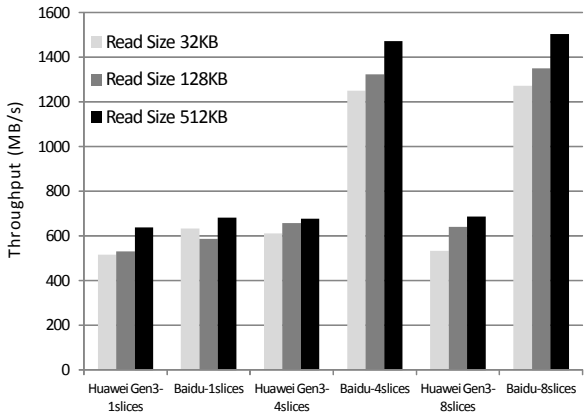

Figure 12. Throughputs of one storage node with the Baidu SDF and the Huawei Gen3 when running one, four, or eight slices with batch size of 44. Random read requests of 32 KB, 128 KB, and 512 KB were used in this experiment.

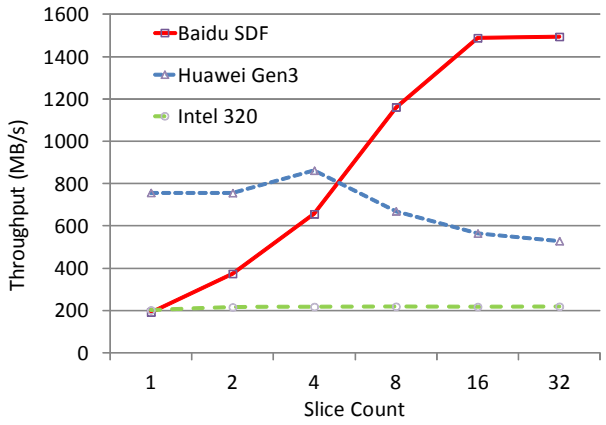

Figure 13. Throughputs of one storage node with sequential reads on the Baidu SDF, the Huawei Gen3, and the Intel 320 with various numbers of slices.

8) when the batch size is fixed at 44. These request sizes are representative for web pages, thumbnails, and images, respectively, in the system's workloads. As shown in the figure, as long as requests are served in parallel at different channels in SDF, both small and large requests can lead to high I/O throughput, though large requests have moderately greater throughput ("Baidu-4slices" and "Baidu-8slices" in the figure). Only when the concurrency in the workload is very limited is SDF's throughput as low as that of the Huawei Gen3 ("Baidu-1slice"). By exposing all 44 channels to the software, with highly concurrent workloads we expect the software to be able to fully exploit the exposed parallelism.

3.3.2 Sequential Reads

At Baidu, an important I/O-intensive operation is to build inverted index tables from web page tables. As shown in Figure 9, the inverted index servers send a request for building

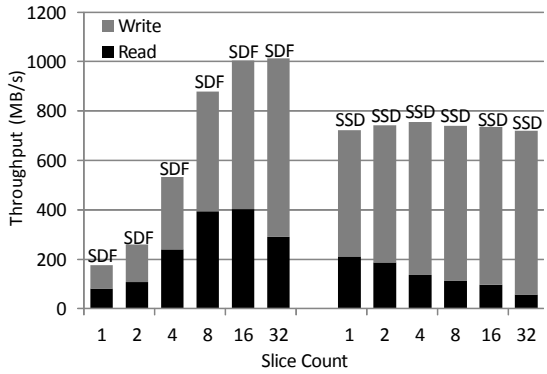

Figure 14. Throughputs of one storage node with the Baidu SDF and the Huawei Gen3 SSD running various numbers of slices with write requests whose sizes are primarily in the range between 100 KB and 1 MB. The requests are not batched (equivalently, the batch size is 1) in the experiment.

the inverted index table to the 'Central Repository of Web Pages' servers, where the web page data is stored on SDFs. On each of the web page servers one or multiple slices can be started. As previously stated, each slice is responsible for storing and accessing keys in a specific range. Because keys for different tables are created in different ranges, each table is managed by a different slice. According to their needs the inverted index servers may ask for a varying number of inverted index tables to be built at a given time. Accordingly, the number of busy slices can vary. The I/O operation in an index-building slice is to scan, or sequentially read, all the KV pairs in the designated key range. To limit the I/O demand of a slice, each slice uses six threads issuing synchronous requests to the storage in the production system. Figure 13 shows the throughput when different numbers of slices are requested in the service. The throughput of SDF scales well with the increase in slice count up to 16 slices where SDF's peak throughput is reached. In contrast, the throughput of the Huawei Gen3 does not scale at all. Even worse, it can become lower when a large number of slices are requested. In this experiment we included the Intel 320 which shows almost constant throughput. This experiment clearly demonstrates the advantage of exposing internal channels to the software.

3.3.3 Writes

In CCDB, Baidu's KV store system, all writes are accumulated into 8 MB patches, which are equivalent to SSTable in Google's BigTable, before being written into storage. Because write requests from the clients cause compaction operations, which entail reading patches from storage, merge-sorting the patches, and writing them back to the storage, a storage device services I/O requests originating from both the clients' write requests and CCDB's internal read and write requests, all with 8 MB request size.

In the experiment each client continuously sends synchronous write requests to a slice. Figure 14 shows the read and write throughput of the Baidu SDF and the Huawei Gen3. Because only the compaction operation incurs reads, the read throughput shown in Figure 14 represents the intensity of the internal data cleanup operation. At the beginning of the experiment, all patches on the storage have been sorted. Therefore, the volume of new written data is correlated to the demand on the compaction operation. Requests from the clients take priority over compaction-incurred requests.

As shown, SDF's throughput continues to increase until the slice count reaches 16 where the throughput peaks at around 1 GB/s. When the slice count is small, the limited concurrency leads to SDF's low throughout in serving client write requests. Accordingly, the compaction intensity, indicated by the read throughput, is also low. However, SDF can take good advantage of high concurrency where its peak throughput reaches a value between the device's peak write throughput (approximately 0.95 GB/s) and its peak read throughput (approximately 1.50 GB/s). Considering the presence of erasure operations, SDF achieves excellent peak throughput. When the slice count increases from 16 to 32 the percentage of the throughput attributable to reads is reduced because the increased demand from the clients' write requests pushes the compaction load lower. In contrast, the Huawei Gen3 exploits channel parallelism for individual 8 MB-patch accesses so it can achieve much higher throughout when the slice count is small, but its throughput does not increase with increasing slice count. Furthermore, because of its limited throughput, the Huawei Gen3 compaction throughput decreases with increasing slice count. With 32 slices the throughput is less than 15% of the aggregate throughput, indicating that most of new written data is left unsorted. This would significantly compromise performance of subsequent read requests from clients.

4. Related Work

We organize our discussion of related work as addressing the reduction of the overhead of random writes, the support of high I/O concurrency, and the reduction of access latency.

4.1 Reducing the Overhead of Random Writes

Though SSD's random-write throughput is higher than that of hard disks, it is much less than its sequential-write throughput [15]. Even for modern high-end SSDs the throughput of random write is only about $1/10$ to $1/7$ of sequential writes [15]. Further, random writes can cause internal fragmentation and lead to performance degradation by an order of magnitude and can significantly reduce lifetime of the device because write amplification is much greater than for sequential write [15]. To address these issues researchers have tried to transform random writes to sequential writes at either the file system level or at the FTL level.

At the file system level, Min proposed a log-structured file system (SFS) to transform random writes in the file system into sequential writes at the SSD [15]. Other log-based file systems, such as JFFS2 [19], YAFFS2 [4], and UBIFS [3], are designed by considering NAND flash characteristics and are used widely, especially in mobile and embedded domains. The log-based file system can eliminate random writes to the flash memory. By requiring a large write unit, SDF essentially forces the software to batch write data as a log. On top of SDF, at Baidu a user-space distributed log-based storage system is implemented. It knows the data types and how actively particular data is accessed, so it can implement a more intelligent block cleaning scheme compared to a log-based file system.

At the FTL level some researchers have used a log buffer at the FTL level to reduce the overhead of random writes. Kawaguchi et al. proposed a flash memory driver that uses a log-based FTL and provides a block interface to the flash that sequentially writes data to the flash memory in a way similar to log-structured file system [11]. Kim et al. proposed BAST, a hybrid FTL scheme [12]. It uses page-level mapping for random writes and block-level mapping for sequential writes [13]. The FAST FTL improves on BAST with a flexible address mapping in the log buffer to improve space utilization. Some FTLs reduce the overhead caused by random writes by considering data access characteristics. LAST reduces the garbage collection overhead by exploiting data locality [14]. CAFTL reduces write traffic to the flash memory by eliminating duplicate writes and redundant data [7]. Both the file-system-level and the FTL-level efforts cannot completely eliminate the garbage collection overhead caused by random writes. SDF eliminates random writes by setting the write size to a multiple of the size of NAND physical erase block. Thus SDF's FTL does not need to conduct garbage collection, and write amplification ratio is kept at one.

4.2 Supporting High I/O Concurrency

SSDs may be deployed in an environment with a large number of applications or threads simultaneously issuing requests to them, that is, with workloads of high concurrency, and it is necessary to exploit parallelism at multiple layers of the I/O stack to achieve high throughput. Here there are three layers that are most relevant. The first is the Linux I/O subsystem, which includes VFS, the file system, the generic block layer, the scheduler layer, the SCSI mid-layer, and the SATA/SAS translation layer. The second is the device driver, which includes command queues, interrupt handlers, and a hardware register interface. The third is in the SSD itself, including the host interface, on-device DRAM, internal data paths, and multiple NAND channels. Many previous works have investigated parallelism at each of these three layers. Seppanen et al. suggest exploiting the parallelism of the Linux I/O subsystem with multiple AIO requests or by using multiple threads issuing synchronous I/O

requests [18]. To take advantage of this parallelism applications need to use AIO or be multi-threaded, necessitating modification of legacy programs. SDF uses multiple threads with synchronous I/O requests to exploit Linux I/O subsystem parallelism because synchronous I/O is easier to implement and is compatible with many legacy applications.

Modern drivers support multiple deep command queues such as NCQ [18] for SATA II, and NVM Express [2] which is designed to address the challenges of system scalability. NVM Express provides optimized command issuance mechanisms and multiple deep queues. Using multiple queues can support concurrent operations and help to improve scalability. Moreover, when the queues are operated on multiple CPU cores, interrupt handling can be more efficient. Following the same principle, SDF creates a queue for each channel. Hahn et al. consider issues with the request scheduling when SSD channels are exposed [10]. We leave studies on intelligent request scheduling on SDF as future work.

Chen et al. investigated the performance implications of SSD's internal parallelism [6]. As we have demonstrated, the aggregate channel-level bandwidth of a commodity SSD can be greater than that made available at the host interface by a significant factor, implying that the internal parallelism in an SSD is far from fully exploited. By exposing flash channels to applications, SDF can effectively increase available channel-level parallelism, and applications can manage data layout among channels or among planes in a channel.

4.3 Reducing Access latency

Foong et al. break down the latency overhead for a request at the granularity of the layers of the I/O stack [8]. As reported, the Linux I/O subsystem takes about 9100 CPU cycles to issue a request to the device and about 21900 CPU cycles to process I/O completion. The total time spent in the Linux I/O subsystem for one I/O request is about 12.9 μs on a 2.4GHz mainstream server processor. In contrast, reading a NAND page from the cell to the host interface takes only about 75 μs for 25nm MLC flash [1], so the software I/O stack is responsible for about 17% of the total request service time for a page read request, which we regards as substantial. Our solution is to simply bypass the Linux I/O subsystem and traditional driver layer, using instead an ultra-light-weight user-space interface and thin driver. The SDF driver provides a register interface and a message signaled interrupt handler. Thus the software layer of SDF is able to significantly reduce the software overhead compared with conventional systems.

5. Conclusions and Future Work

In the paper we present SDF, a software-defined SSD, for web-scale Internet storage infrastructures. SDF is a high-performance and low-cost SSD architecture with high utilization of raw bandwidth and storage space and with predictable performance. It has successfully replaced traditional SSDs at Baidu's Internet storage infrastructure with 3000 SDFs deployed and more scheduled. SDF's success derives from its hardware-software co-design. In SDF, the hardware exposes the internal channels to the applications through customized FTL controllers. Dovetailing with the hardware design, the software layer enforces large-granularity writes and provides primitive and light-weight functionalities through kernel bypass. Our experimental measurements show that SDF can deliver about 95% of the raw flash bandwidth and provide 99% of the flash capacity for user data. SDF increases the I/O bandwidth by 3 times and reduces per-GB hardware cost by 50% on average compared with Baidu's commodity-SSD-based system.

The success of the SDF at Baidu is highly dependent on two key workload characteristics that we believe are also prevalent in most data centers supporting Internet services. One is large sequential writes. Because write requests have been mostly served in log-structured storage systems, such as Google's LevelDB and Facebook's Haystack, SDF's performance advantage on write can be realized without re-writing or re-structuring existing software. The other is high concurrency for exploiting SDF's exposed channels. Because Internet service is mostly multi-threaded and transaction-based for supporting millions of users, high concurrency in the I/O workload is expected and is the reality in most cases.

The Baidu SDF research is ongoing on multiple fronts. First, as previously described we will implement a load-balance-aware scheduler to help SDF reach its peak throughput with a smaller number of concurrent requests. Also, to further pursue our hardware/software co-design approach, we intend to further investigate the possibility of "moving compute to the storage," that is, integrating specialized computational functionality into the storage devices themselves to minimize I/O traffic; an initial success has already been reported [17]. Finally, recognizing that flash memory has finite write endurance, we believe that that it would be both possible and useful to incorporate, and expose, a data reliability model for flash memory in our infrastructure.

Acknowledgments

The SDF project has benefited greatly from constructive input from many engineers and interns at Baidu, in particular Guangjun Xie, Wei Qi, Hao Tang, and Bo Peng. We would like to thank Kei Davis for constructive input on the final composition of this paper. We are also thankful to Jason Cong for his support and suggestions on the paper.

References

[1] "Micron 25nm MLC Product Datasheet." *http://www.micron.com*.

[2] "NVMe: Non-volatile Memory Express." *http://www.nvmexpress.org/*.

[3] "UBIFS: Unsorted Block Image File System." *http://www.linux-mtd.infradead.org/doc/ubifs.html/*.

[4] "YFFS: Yet Another Flash File System." *http://www.yaffs.net/.*

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. "Bigtable: A Distributed Storage System for Structured Data." In *Seventh Symposium on Operating System Design and Implementation*, 2006.

[6] F. Chen, R. Lee, and X. Zhang. "Essential Roles of Exploiting Internal Parallelism of Flash Memory based Solid State Drives in High-speed Data Processing." In *IEEE 17th International Symposium on High Performance Computer Architecture*, 2011.

[7] F. Chen, T. Luo, and X. Zhang, "CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives." In *9th USENIX Conference on File and Storage Technologies*, 2011.

[8] A. Foong, B. Veal, and F. Hady. "Towards SSD-ready Enterprise Platforms." In *1st International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, 2010.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. "The Google File System" In *19th ACM Symposium on Operating Systems Principles*, 2003.

[10] S. S. Hahn, S. Lee, and J. Kim. "SOS: Software-based Out-of-order Scheduling for High-performance NAND Flash-based SSDs." In *IEEE 29th Symposium on Mass Storage Systems and Technologies*, 2013.

[11] A. Kawaguchi, S. Nishioka, and H. Motoda. "A Flash-memory based File System." In *Winter USENIX Technical Conference*, 1995.

[12] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho. "A Space-efficient Flash Translation Layer for Compact-flash Systems." In *IEEE Transactions on Consumer Electronics*, 2002.

[13] S. W. Lee, D. J. Park, T. S. Chung, D. H. Lee, S. Park, and H. J. Song. "A Log Buffer-based Flash Translation Layer Using Fully-associative Sector Translation." In *Trans. on Embedded Computing Systems*, 2007.

[14] S. Lee, D. Shin, Y. J. Kim, and J. Kim. "LAST: Locality-aware Sector Translation for NAND Flash Memory-based Storage Systems." In *SIGOPS Operating Systems Review*, 2008.

[15] C. Min, K. Kim, H. Cho, S. Lee, and Y. Eom. "SFS: Random Write Considered Harmful in Solid State Drives," In *10th USENIX Conference on File and Storage Technologies*, 2012.

[16] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. "The Log-structured Merge-tree (LSM-tree)." In *Acta Informatica* 33(4):351-385, 1996.

[17] J. Ouyang, S. Lin, Z. Hou, P. Wang, Y. Wang, and G. Sun. "Active SSD Design for Energy-efficiency Improvement of Web-scale Data Analysis." In *International Symposium on Low Power Electronics and Design*, 2013.

[18] E. Seppanen, M. T. O'Keefe, and D. J. Lilja. "High Performance Solid State Storage under Linux." In *IEEE 26th Symposium on Mass Storage Systems and Technologies*, 2010.

[19] D. Woodhouse. "JFFS: The Journaling Flash File System." In *Ottowa Linux symposium*, 2012.