Enabling Cost-effective Data Processing with Smart SSD

Yangwook Kang
Computer Science
UC Santa Cruz
ywkang@cs.ucsc.edu

Yang-suk Kee
System Architecture Lab
Samsung Semiconductor Inc.
yangseok.ki@samsung.com

Ethan L. Miller
Computer Science
UC Santa Cruz
elm@cs.ucsc.edu

Chanik Park
Memory Strategic Planning
Samsung Electronics
ci.park@samsung.com

***Abstract*—This paper explores the benefits and limitations of in-storage processing on current Solid-State Disk (SSD) architectures. While disk-based in-storage processing has not been widely adopted, due to the characteristics of hard disks, modern SSDs provide high performance on concurrent random writes, and have powerful processors, memory, and multiple I/O channels to flash memory, enabling in-storage processing with almost no hardware changes. In addition, offloading I/O tasks allows a host system to fully utilize devices' internal parallelism without knowing the details of their hardware configurations.**

To leverage the enhanced data processing capabilities of modern SSDs, we introduce the Smart SSD model, which pairs in-device processing with a powerful host system capable of handling data-oriented tasks without modifying operating system code. By isolating the data traffic within the device, this model promises low energy consumption, high parallelism, low host memory footprint and better performance. To demonstrate these capabilities, we constructed a prototype implementing this model on a real SATA-based SSD. Our system uses an object-based protocol for low-level communication with the host, and extends the Hadoop MapReduce framework to support a Smart SSD. Our experiments show that total energy consumption is reduced by 50% due to the low-power processing inside a Smart SSD. Moreover, a system with a Smart SSD can outperform host-side processing by a factor of two or three by efficiently utilizing internal parallelism when applications have light trafic to the device DRAM under the current architecture.

I. Introduction

Rapid developments in non-volatile memory (NVRAM) technology have challenged the assumption that I/O devices in storage systems are slow. In contrast to hard drives, where the performance is limited by the movement speed of mechanical parts, the performance of NVRAM devices can be improved simply by adopting more powerful processors and improving the internal bandwidth to the underlying storage media. More specifically, modern high-performance Solid State Disks (SSDs) have multiple powerful processors, large battery-backed DRAMs, and 8–16 (or more) independent I/O channels to provide high performance.

Despite the increasing hardware capability of SSDs, however, the performance at the application end is still limited due to legacy hardware and software designed for hard drives. Legacy storage subsystems typically throttle the number of pending I/O requests to accommodate disks that do not support concurrency; however, SSDs require a large number of concurrent I/O requests to maximize their performance. Equally important, locking and interrupt mechanisms can introduce more overhead than processing an I/O request [21], and block interfaces such as SAS and SATA are neither fast nor rich enough to leverage the potential of SSDs. The NVMe (Non-Volatile Memory Extension) standard for PCI Express [7] is designed to exploit the native performance of devices, but it is not yet robust, and requires applications to understand the characteristics of the underlying media to optimize their performance.

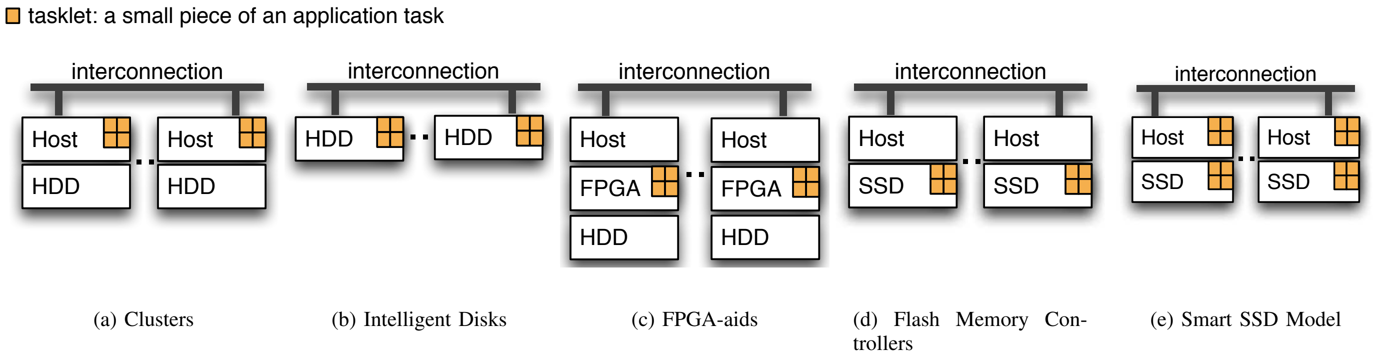
In this paper, we introduce the Smart SSD model, allowing host systems to fully exploit the performance of SSDs without requiring operating systems and applications to understand the particular characteristics of SSDs. This is achieved by offloading data-intensive tasks from a host application to the Smart SSD. Each Smart SSD has an internal execution engine for processing locally-stored data, and the host machine coordinates the sub-tasks as well as directly processing some parts of the tasks. By isolating data traffic within a device, the execution engine can schedule the I/O requests more efficiently; it can decide to fully utilize the I/O channels when the device is idle, or pause the data processing when there are many pending requests from users. This model also enables energy-efficient data processing, because power-hungry host-system resources such as DRAMs and CPUs are not used.

In addition to enhancing the firmware, we also built a prototype of the host infrastructure to leverage Smart SSDs. The communication between a Smart SSD and its host system is handled by an object-interface implemented on top of the SATA protocol to provide a compatible and flexible API to applications and operating systems. The Hadoop MapReduce framework [1] is used as an application interface to utilize Smart SSDs while hiding communication details.

The specific contributions of this paper include:

- A model that demonstrates the use of an SSD as a data processing node that can achieve both higher performance and energy savings through enabling efficient data flow and consuming extremely small amounts of host system resources.
- The first evaluation of in-storage processing (ISP) on a real (not simulated) MLC (multi-level cell) SSD device.
- An end-to-end evaluation of performance and energy covering the entire system.

This paper is organized as follows. In Section II, we survey earlier approaches to leveraging in-storage processing and relate it to our work. Then, we summarize the architecture of modern SSDs in Section III and the implementation details of our SmartSSD approach in Section IV. The methodology



(a) Clusters

(b) Intelligent Disks

(c) FPGA-aids

(d) Flash Memory Controllers

(e) Smart SSD Model

Fig. 1. Comparison of computing models: Smart SSD is designed to support generic and complex tasks, leveraging the parallelism not only within the device but also between the host and devices.

for experiments, applications, and experimental results are presented in Section V. We conclude the paper with a discussion on future SSD architectures for ISP and future research directions.

II. Related Work

The idea of using disk-resident processing to delegate system intelligence from host CPU to peripherals was first introduced a few decades ago for database machines. Early systems mainly focused on improving the performance of hard disks through dedicating a processor per head, track, or disk [15, 18]. However, they failed to survive because the high manufacturing cost could not be justified for marginal performance improvement. As computer systems were commoditized, the idea was resurrected in the late 1990s, when so-called intelligent disks (with in-storage processing) were investigated actively [10, 17]. The studies on intelligent disks regard a disk as a complete computer system for efficient data processing and assume that the aggregation of compute capacity of less powerful but scalable processors performs better than the host-based approaches with dumb disks (Figure 1(a)).

However, realizing in-storage processing with hard disks has been a difficult task, because it requires additional infrastructure. Keeton *et al.* introduced the concept of IDISK (intelligent disk) for scalable decision support databases [10], in which the data processing is offloaded from desktop processors to lower-power processors to improve cost-performance. Thus, IDISK is designed as a general purpose network node that can replace costly cluster nodes. However, it was not able to be realized at that time because of the physical space constraint and the limited power and cooling supply issues. Riedel *et al.* proposed a similar concept called Active Disk (Figure 1(b)) that combines on-drive processing and large memory to allow disks to execute application level functions in the device [16, 17]. Another Active Disk paper by Acharya *et al.* focused on the programming model and algorithms rather than the device architecture [2]. They proposed a streaming programming model for application development, a sandbox model for secure task execution, and operating system support both at the device level and at the host level. However, none of the previous approaches were implemented on commercially-available hardware. For example, Active Disk was designed to exploit internal disk components, but the experiments had to use an additional 64 MB bytes of RAM and an external processor to simulate a 4 GB disk.

More recently, there has been a resurgence in interest in in-storage processing, as exemplified by FAWN (Fast Array of Wimpy Nodes) [3], which uses low-power processors and flash to do data processing. However, FAWN is designed to handle PUT-GET style requests rather than coordinating and offloading arbitrary I/O processing tasks. In this architecture, *all* data processing is offloaded to the flash nodes, requiring each node to consider data placement. There is no central coordination in the system, requiring distributed protocols to spread work around. This model works for PUT-GET requests, but is poorly suited to Hadoop-style processing. In addition, since it requires raw accesses to flash, custom-designed embedded hardware is needed, decreasing portability.

The concept of an intelligent disk was also adopted in system-level incorporating more than a single device. Mueller *et al.* introduced an external module (*e. g.*, FPGA) attached to a disk (Figure 1(c)) to implement the system intelligence [12, 13]; IBM Netezza's Blade server [5] is a commercial version of this approach. In contrast, Oracle's Exadata is a commodity-based approach in which storage servers reduce the amount of traffic by filtering data [19]. One of the main disadvantage of this approach is that it is not easy for users to write their own processing logic. Given the high costs of these systems, Smart SSD and its support infrastructure can provide various kinds of functionality at significantly lower cost with easier programming model.

Smart SSDs, as shown in Figure 1(e), use the host system to coordinate the tasks between SSDs and a host, and support executing arbitrary data processing functions within a drive. The in-storage processing in the Smart SSD model exploits its own components and also improves the I/O performance by leveraging internal parallelism efficiently: multiple independent I/O channels and embedded processors. Unlike intelligent disks, the performance of in-storage processing is not as affected by data fragmentation, and complex operations such

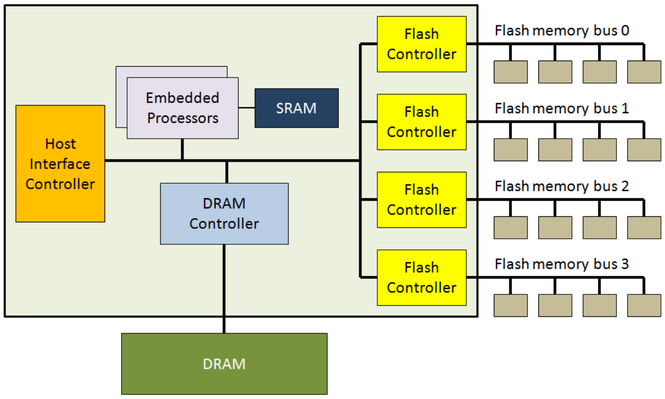


Fig. 2. The typical architecture of modern flash-based SSDs

as join and merge can be better handled due to powerful processors and large memory. This model works better than the device-only computing model because it can leverage a few powerful general purpose processors as well as the embedded processors in the SSDs.

Recently, several researchers have explored the potential benefits of SSD-based in-storage processing. Kim *et al.* investigated the benefits of ISP on SSD with the database scan operation [11]. They claimed that magnetic disk is no longer the bottleneck for storage architecture on legacy database machines because the fast storage medium and the increased parallelism in SSDs make the other components—embedded processors and memory—the bottleneck. Based on this intuition, a FMC (Flash Management Controller) with a scan function (Figure 1(d)) was simulated to filter data along the flash data path. However, due to the limited processing power, the limited amount of memory, and real-time contraints in a FMC, its applications were limited to relatively simple data processing such as filtering.

Boboila *et al.* proposed Active Flash to enable out-of-core data analytics [4], providing an analytical model that shows the performance-energy tradeoffs in moving data processing to SSDs in a HPC (High Performance Computing) context. Using an SSD simulator based on DiskSim, it shows that, with careful job scheduling that considers both the garbage collection overheads and idleness of SSDs, a significant amount of energy can be saved with only minor performance degradation. While the results are well aligned with our measurements on real SSDs, we found that the current SSD architecture is not sufficient to support complex tasks, as shown in Section V-C. We also provide the in-storage processing model, which defines a host and application interface.

III. The Smart SSD Architecture

SSD architectures have evolved as the need for performance and capacity has increased, but this evolution has typically been limited by the need to avoid changes in the host hardware and operating system. However, the performance of modern SSDs has now reached the maximum bandwidth of the SATA interface when large burst requests are given, requiring the host system to generate more flash-optimized requests for further performance improvements.

Adding in-storage processing capabilities to SSDs is one way to reduce the demand on the data path from applications to devices while minimizing the changes required to applications and host systems. By offloading the application tasks and reducing the use of host system resources, it enables energy-efficient data processing that can fully utilize the internal resources in SSDs. Harnessing more powerful processors and interfaces with higher I/O bandwidth on a host can improve the overall data processing performance. However, it is still not easy to reduce energy consumption in a power-hungry host environment without traffic control. The Smart SSD model provides an efficient way not only to improve performance by reducing the amount of data transfers from device but also to save energy by utilizing low power processors in device. In this section, we describe the hardware capability of modern SSDs, the Smart SSD model exploiting the current architecture, and the extended host-device communication.

A. Modern SSD Architecture

Figure 2 illustrates the general architecture of modern SSDs [4, 11]. An SSD consists of three major components: the SSD controller, DRAM, and the flash memory array. Further, the SSD controller is composed of three subcomponents: the host interface controller, the embedded processor, and the flash memory controller.

The host interface controller implements a bus interface protocol such as SATA, SAS, or PCIe. SATA 3.0 and SAS 2.0 support up to 6 Gbps of bandwidth while PCIe 3.0 can transfer 8 Gbps per lane. Typically, 32-bit RISC processors such as the ARM series are used as embedded processors [8], providing host command handling and a flash translation layer (FTL) to map a Logical Block Address (LBA) to a physical page number in the flash memory. The embedded processors and associated SRAM (for executable code storage) require much less energy to run the SSD firmware than would a standard host CPU.

The flash memory controller (FMC) is in charge of reliable data transfer between flash memory and DRAM, which is used to cache user data and to store metadata for the FTL. Its key functionality includes Error Correction Code (ECC) and Direct Memory Access (DMA). The FMC is also responsible for exploiting the parallelism in the flash memory by chip-level and channel-level interleaving techniques to improve I/O performance.

Finally, the flash memory array is the persistent storage medium; NAND flash is the most popular choice. NAND flash memory is composed of blocks, each of which consists of pages. A block is the unit of erase while the page is the unit of read and write. Flash memory arrays typically have multiple channels, allowing for a high degree of parallelism.

B. In-Storage Processing Model for SSD

While providing comparable or even better I/O performance and processing power than network-attached low-power cluster nodes, host system support for SSDs is still very limited; the TRIM command is the only standardized workaround to resolve sub-optimal performance issues in SSDs. In contrast to cluster nodes for which variable-length message passing is possible, current SSDs cannot handle requests smaller than a single sector, resulting in unnecessary data traffic and higher energy consumption. Maximizing the performance of SSDs is more difficult because operating systems are not designed to deliver large burst requests to devices. The need to change most of the low-level I/O subsystems makes it difficult for host systems to adapt to handle new storage devices efficiently.

In-storage processing is one way to alleviate this compatibility issue. Without changing any of the I/O subsystems in the operating systems, it allows SSDs to execute the I/O tasks internally, fully utilizing devices' internal components and their knowledge of the hardware and physical data organization. The Smart SSD model proposed in this paper is designed to support in-storage processing with minimal changes to the host system, providing the same device interface regardless of the underlying communication protocol. Instead, the details of the additional communications and task handling are hidden by a Hadoop framework, and the device exposes simple and flexible APIs that can take any in-storage processes that may need an arbitrary number of parameters to execute.

In the previous intelligent disk models depicted in Figure 1(b), a CPU in the disk component is used as the main processing unit, executing the application tasks while the host conducts minimal tasks such as coordination and scheduling. In contrast, we use a CPU scavenging model where the host and devices share the workloads considering their computational and I/O processing power as illustrated in Figure 1(e). Each compute node in this model runs one or more *tasklets*, where a tasklet is a unit of an application task that can be assigned and executed on either the host or a device independently and in parallel. For example, if a database table scan operation is conducted per segment, each per-segment scan can be a tasklet, and the scan operation is an application task. The unit of computation can be adjusted based on computational complexity, resource availability in a device, or other factors.

In this approach, the host system plays two roles: computing unit and coordinator (or scheduler). The host can assign a tasklet to a SSD based on its current utilization and the execution cost, which depends on the complexity of the algorithms, so highly complex tasklets can be performed at a host leveraging the powerful processors and large memory. For efficient scheduling, Smart SSDs can inform the host system about the current load of the device upon request. The mechanism for monitoring the load of each device depends on the type of host-device interface being used. The host may need to poll the devices in the SATA protocol, while additional background call-back connections can be established in SAS, and interrupts can be used for PCIe interfaces. In our prototype, which uses the SATA protocol, for example, the progress of a tasklet execution is reported to the host system as part of the return value of the read command to optimize the polling interval.

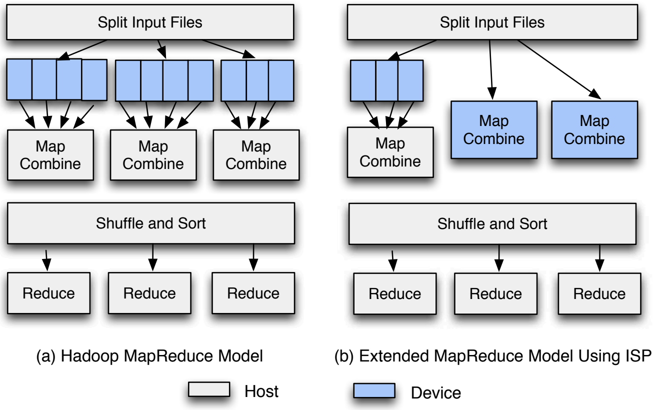


Fig. 3. Extended MapReduce framework

The host treats a Smart SSD as a single virtual processor, so the detailed hardware configuration of the device is not exposed to the host. If multiple embedded processors are available for ISP, the aggregated computing power of all the embedded processors is considered as the total computing power of the virtual processor. Tasklets can be assigned to one of the embedded processors of the virtual processor using cycle stealing. However, it is also possible to use dedicated processors for ISP for better performance. Regardless of the implementation, tasklets are scheduled between the processors considering priority and runtime execution costs.

C. Host-Device Communication

In-storage processing requires some additional I/O commands to manage and execute tasklets. One of the criteria in designing these low-level device command sets is that devices should be able to provide a generic and flexible interface to applications. It requires support for tasklet parameters of arbitrary size such as input/output addresses and search keywords to avoid adding a new command for each tasklet. In addition, the interface needs to be independent from the underlying disk interface similar to the way the virtual file system layer is used for multiple different file systems.

To provide a clean and simple interface to meet those requirements, Smart SSDs use an extended object-based interface [9], which contains an *execute_object* command, and runs on top of the existing device interface such as SATA and SAS. By doing so, applications can have the same sets of APIs regardless of the underlying communication protocol. This interface allows users to add or remove a tasklet, group the logical addresses to be processed (input object), and store and read the results of the execution. Depending on the support for bidirectional communication, the host systems poll the results or get notification from the device when the results become ready. The detailed protocol implemented in our prototype is depicted in Figure 5 and described in Section IV.

The Smart SSD model uses the MapReduce programming model as an application interface because the MapReduce model can provide independent sets of input data that can be mapped to the tasklets directly, allowing them to be executed in parallel without any message passing or locking problems. Figure 3 depicts the extended MapReduce model for in-storage processing. A typical MapReduce application consists of a pair of *map* and *reduce* functions written by users. As illustrated in Figure 3(a), the *map* function is invoked after the partitioned data is read into the host memory, and the key and value pairs for the data are generated. A *map* function takes the generated input key-value pairs and produces a set of intermediate key-value pairs. All intermediate values associated with the same intermediate key are combined, shuffled, and sorted, then passed to the *reduce* function. The *reduce* function accepts intermediate key-value pairs, determines the set of values with the same key, and merges the values together. This model generates a large amount of disk and network traffic since the input data needs to be read into the memory before being processed, and the split data file needs to be sent over the network to the remote map nodes.

Instead of reading the raw file data into the host, the extended MapReduce model allows users to create an on-device map function, which internally calls the tasklets in the device, as shown in Figure 3(b). After splitting the data files, the framework sends an *execute_command* request to the device to execute the corresponding tasklet using a given range of LBAs; our on-device prototype re-uses read and write functions of the FTL so storing LBAs are maintained by our Hadoop file system and given to the device transparent to applications. However, logical addresses can be object IDs rather than simpely LBAs if a device manages them on *write_object* requests, thus the same object-interface can be used with any types of physical transport layers. Each on-device map function performs combine and local reduce, before returning to the host system to minimize the disk traffic. The host system will then shuffle and sort the results from the on-device map functions, and invoke the *reduce* function. By doing so, as the size of input data increases, this in-storage processing model can save host CPU, I/O bandwidth, and memory resources. Specifically, since the amount of memory used to temporarily store input data files is dropped after the map function, it can also reduce the cache pollution at a host system.

D. Tasklet Programming

The SSD firmware does not usually provide general operating system features such as processes and dynamic memory allocations due to the limited size of SRAM and the unnecessary overhead from virtualizing hardware components. Rather, it directly accesses and manages the hardware resources. Therefore, porting full virtual machine-based programming languages such as Java or Python is not realistic, despite the advantage of secure execution of tasklets using sandboxing.

Instead, cross-compiled C code is the most efficient way of writing a tasklet, but it requires programmers to understand the details of the firmware implementation and hardware configuration. Moreover, the device can be exposed to many possible dangers such as overwriting of existing data and crashes, since providing a sandbox for native applications is difficult. Thus, only a small group of trusted people can write a tasklet and publicly deploy it for the target device.

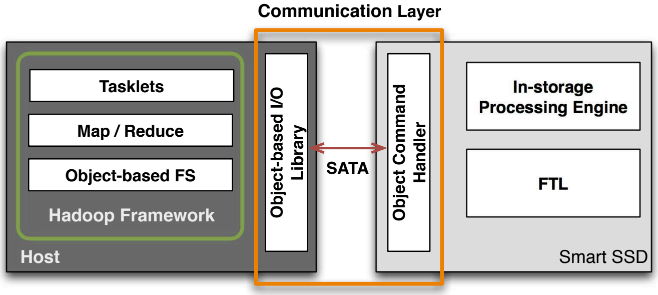


Fig. 4. In-storage processing architecture for Smart SSD

Similar to CUDA [14], when the standard API for in-storage processing is defined, the tasklets can be programmed by a simple script language where only the standard APIs provided by the device can be used. The interpreter can verify illegal accesses to protected resources and execute the tasklet at the same time, or optionally compile the tasklet natively once verified.

While in this work we focus on web-log analysis, as discussed in Section V-C, any applications that extract information from a large corpus of data can benefit from tasklets. For example, a desktop indexing tasklet can automatically tag incoming data, reducing the need for periodic crawling, and provide only the data blocks that a host-side application is interested in. As another example, multiple independent graphs stored across Smart SSDs can be processed concurrently. Thus, a host can collect only the matching leaf nodes from Smart SSDs, removing the need for transferring internal nodes.

IV. Implementation

Figure 4 depicts the architecture of our Smart SSD prototype. It consists of the three major components: the ISP engine, the Hadoop MapReduce framework, and the communication layer. The Smart SSD implements an event-driven execution engine for ISP, which can process the tasklets assigned by the host. The Hadoop MapReduce framework is used as an application framework. Finally, the communication layer implements an ISP protocol between the application framework and the Smart SSD. This section describes the implementation details of each component.

A. In-Storage Processing (ISP) Engine

The ISP engine is an event-driven processing framework to execute tasklets. It takes the C program cross-compiled for the ARM processors. The tasklets are executed on a dedicated processor in the device. Since dynamic memory allocation is not supported by the firmware, the tasklets are preloaded into the device, and memory space for input and output objects is reserved. Each tasklet is an object from the user's perspective, and identified by an unique identifier called an object id. To

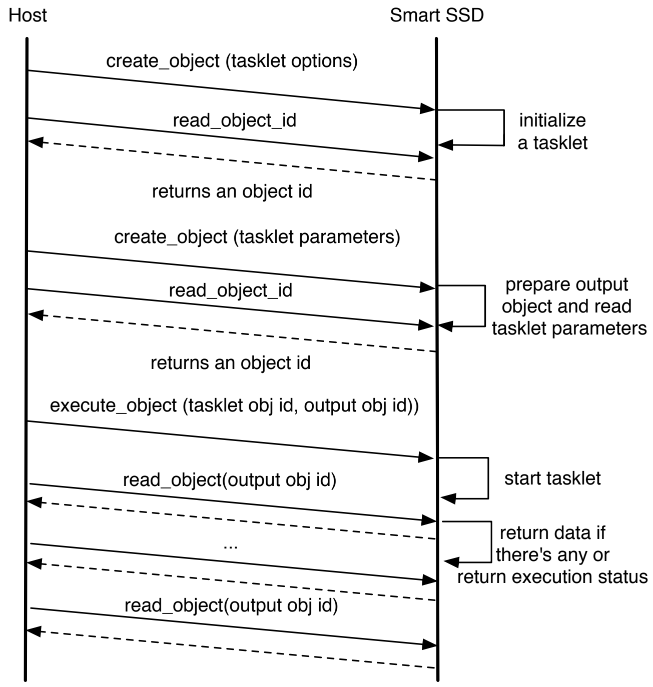


Fig. 5. Protocol diagram for ISP using the object interfaces

invoke the tasklet, the host passes an object id assigned to the tasklet at download time.

B. Extended Hadoop MapReduce Framework

Hadoop is an open-source project that provides a programming framework for large-scale distributed batch processing. Among various sub-components in this framework, we modified the Commons package of the framework to add an object-based file system for low-level device handling and to extend the MapReduce framework to support on-device map functions.

The goal of the object-based file system is to manage the raw SSD device by providing a resolution between an object name and an object identifier while hiding the low-level communications. Unlike the pure object-based file systems, which offload the entire block management layer to the device, it still manages the LBAs for reading and writing data blocks, because the FTL in the underlying SSDs is used. Therefore, while users can access data using an object identifier, the id and offset are translated into block numbers and sent to the device internally.

The extended MapReduce framework now supports *DeviceJob*, which internally manages the tasklets in the device, bypassing the file-read phase. Users can define a device job similar to how a normal job is configured; instead of using *InputFormat* and a host map function, applications can use *DeviceMapper* and *DeviceMapperFormat* where *DeviceMapper* represents one tasklet, and the *DeviceMapperFormat* converts the output of the *DeviceMapper* to a set of key/value pairs, which are then used by the *reduce* function.

C. Communication Layer

The communication layer is implemented on top of the SATA protocol. At the host end, the object-based I/O library handles the low-level requests and delivers the results to the object-based file system. In the device, the SATA command handler for the extended APIs is implemented in the SSD firmware. It interprets the vendor commands from the host and triggers the execution of appropriate tasklets stored in the device.

1) Object-based I/O library: The logical block addresses, encapsulated in an object request, are delivered to the device driver through a Unix system call, i.e., `ioctl` with the `ATA_PASS_THROUGH` command [6]. Each packet for vendor commands has a header that contains information such as command identifier, object id, data length, and others, followed by a payload containing the metadata for the commands. Since the `ioctl` operation is not supported in Java, this library is written in C and delivers the results to the Hadoop framework through the Java Native Interface (JNI), incurring the argument passing and copying overheads between a Java virtual machine and native C code. This interface interacts with a Hadoop file system and *MapReduce* job trackers, so it is not directly exposed to *MapReduce* applications. They can use the functionality by defining a device job specifying a type of in-storage processing and statis parameters for the type without knowing the details of communications.

2) Object command handler: The object command handler in the device firmware is responsible for handling create, execute, and read requests on tasklets. Internally, these commands are implemented as vendor specific commands, reserved by the SATA standard for extensibility. In our prototype, three vendor-specific commands are implemented: *create_object*, *execute_object*, and *read_object*.

Figure 5 shows the protocol these vendor-specific commands use to run tasklets. The *create_object* command is first called to create and initialize a target tasklet with static parameters related to the tasklet such as the size of one key-value entry. Then, another *create_object* command is called to create an output object, which contains the outputs of the tasklet and also can deliver some runtime parameters such as search keywords and the maximum number of outputs for the tasklet. Once a tasklet object and the corresponding output object are ready, the host can invoke *execute_object* to start the tasklet, and call *read_object* to get the results.

While implementing the object command handler, we encountered several design issues because the SATA protocol does not support bi-directional communication, which can read and write data by sending one I/O request. When implementing on top of the SATA protocol, any command that has an output needs to be split into two steps under a global lock for atomicity. For instance, the *create_object* needs to return an object ID after creating an object. So the actual implementation of this command consists of two vendor-specific commands: one for object creation and another for the retrieval of the created object ID.

Similarly, since establishing two concurrent connections and device-initiated connections are not possible in the SATA protocol, polling is the only option to retrieve the outputs of operation from a device. Each polling operation has slightly less latency than a read request, but this operation is synchronous, so frequent polling would reduce the I/O throughput of an application. Therefore, we let the device return the number of processed pages whenever a polling request comes in, and the host dynamically adjusts the polling intervals to avoid a flood of requests.

V. Experiments

In this section, we explore the benefits and limitations of the Smart SSD using several benchmarks and applications. Since the benefits of Smart SSDs expect to come from the reduced host system resource usages and data transfers between a host and a device, the experiments are designed to characterize the in-storage processing with Smart SSD and aid us in identifying the classes of applications that can leverage the current Smart SSD architecture. First, we run a micro-benchmark that measures computing power and memory access latency of a device. Then, we measure the overhead of the object-based communication layer implemented on top of the SATA interface, focusing on the energy efficiency, data processing performance, and host resource usage. Based on the results, we identify the hardware components that limit the range of applications, and propose a future SSD architecture for in-storage processing.

A. System Configuration

Smart SSD: The Samsung SSD used in this paper has a 3 Gb/s SATA interface, 16 I/O channels, and a capacity of 200 GB. This SSD is equipped with two ARM processors, and its SLC flash memory arrays have an 8 KB page size. We used a commodity SSD that is currently on the market, and did not change any hardware components to support in-storage processing. However, we extended the firmware of the prototype to support the object command handlers and application tasklets. An internal read operation for tasklets is configured to fetch 128 pages at a time. Throughout the experiments, all I/O requests are sent sequentially to the device, and Native Command Queueing (NCQ) is turned off.

Host: Our experiments are conducted on a desktop machine with one 3.3 GHz Intel i5-2500 processor with 4 cores and 4 GB of DDR3 DRAM. We use a single SSD connected to a 3 Gb/s SATA HBA (Host Bus Adaptor) while a separate HDD is dedicated to the operating system and the Hadoop framework. The SSD and HDD do not share the HBA. This system runs Ubuntu Linux 11.04 and a modified version of Apache Hadoop 0.20.2 with default parameters.

B. Microbenchmark

Since the SSDs on the market do not target general purpose computing at all, they have the minimal hardware specifications that meet the design performance requirements. The devices have less computing power and higher DRAM access latency than the hosts while multiple DMA controllers take care of data transfers between host and device. The minimal hardware specifications, however, make the application tasklets more sensitive to the computation and memory access patterns in the case of in-storage processing. For this reason, understanding the performance characteristics of in-storage processing with our SSD device is crucial to achieving not only high energy efficiency but also high performance.

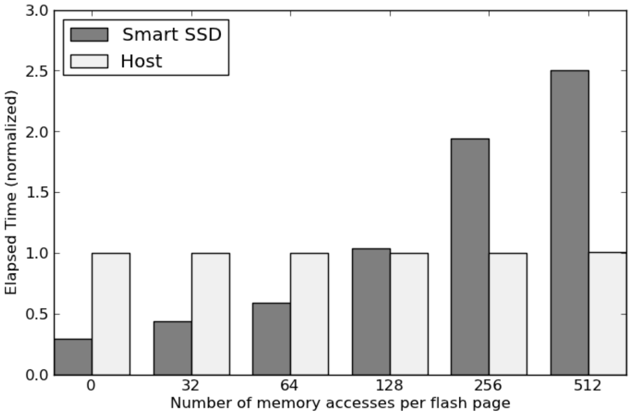


Fig. 6. Performance of in-storage processing: for each memory access, Smart SSD reads one 4 byte integer and performs one integer comparison. The performance of the Smart SSD is normalized to the access time at the host side.

1) Read performance: By device specification, the internal read bandwidth of our device is around 2.5× faster than the I/O bandwidth between a host and a device. However, when including firmware overhead, our internal measurement shows the internal read performance is 1.8× faster than the I/O bandwidth between a host and a device, even when the 16 I/O channels are fully utilized. This not only means that SSDs can hide their firmware management latency from a host system, but shows the potential performance benefits of in-storage processing. When considering that it is difficult to utilize the whole I/O bandwidth to process a certain I/O task, there is a possibility that in-storage processing can improve the performance of applications by more than a factor of two.

2) DRAM Access Latency: To measure the internal memory access latency, we measure the read performance with 1 GB of randomly generated data varying the number of DRAM accesses and comparisons per 8 KB flash page. For each memory access, the benchmark reads one 4 byte integer and performs one integer comparison. For example, when the number of memory accesses per page is 32, it reads 128 bytes per 8 KB page, and performs 32 integer comparisons. As shown in Figure 6, the overall trend is that the execution time of the benchmark increases in proportion to the number of memory DRAM accesses and comparison operations while the performance of host is nearly constant. We measure the same tendency when increasing the number of comparisons, meaning that the DRAM access latency is high. This is mainly due to the limitations of the current SSD architecture where

L1/L2 caches are not available and the bandwidth between CPU and DRAM is not high enough. It is to be noted that DRAMs inside the device are not designed to be accessed by the embedded processors. Instead, there is a small amount of SRAMs that contain the firmware and provide memory space for request processing, and DRAMs are used as a cache for read and write, whose contents are not directly accessed by CPUs most of the time. Although this design decision is natural for normal SSDs where processors are not involved in transferring the contents in DRAM to a host during basic read/write operations, it seems that hardware enhancements are required for Smart SSDs in order to broaden their target applications.

This tendency shows that under the current SSD architecture, Smart SSD can only benefit when the number of accesses per page is small (*e. g.*, less then 128 for this device). The requirement for a small number of DRAM accesses can be interpreted in two ways. Applications that are interested in a small portion of data out of a large entity (*e. g.*, one field out of large log entry or one column out of large tuple) can benefit from Smart SSDs so most data in a page are skipped. The pages transferred for such applications just pollute the caches in the host and waste host resources. The second group of applications have a small number of reads and a large amount of accesses and processing, so the processor can copy the data from DRAM to SRAM before processing. These groups of applications can benefit from the high internal read performance of SSD, amortizing the high DRAM latency penalty over the amount of accessed pages. This limitation can be easily alleviated by adopting CPU caches and increasing the bandwidth between a tasklet processor and DRAM.

3) Interface Overhead: Our custom Smart SSD protocol is designed and implemented on top of the SATA protocol so it can be used without modifying the current interface and the OS kernel. However our communication protocols add some overhead over the standard SATA commands because they must handle the polling interval and bi-directional communication issues.

Figure 7 shows the overhead of the commands supported in our prototype. We measure the turnaround time of each function at two locations: the Hadoop framework and JNI device communication module. Since the SATA protocol does not support bi-directional communication and variable-size requests, unlike object-based devices, a create command issues two I/O requests of 512 bytes each. Compared to a normal read command, which takes less than 215 us for one sector read, our results show that the *create_function* and *create_object* commands, each of which uses *create_object* commands with different metadata, take slightly more time. On the other hand, *execute_object* and *read_object* exhibit latency similar to that of a normal read command because the parameters for these operations can be embedded in one SATA command.

Due to the memory allocation and copies between Java and C, the JNI module adds 150–380 μus of latency to each function, depending on the amount of data passed to the function. Compared to the other operations, *read_object* experiences a long latency in the JNI wrapper due to the output data copied between C and Java and piggybacked information processing such as the polling interval.

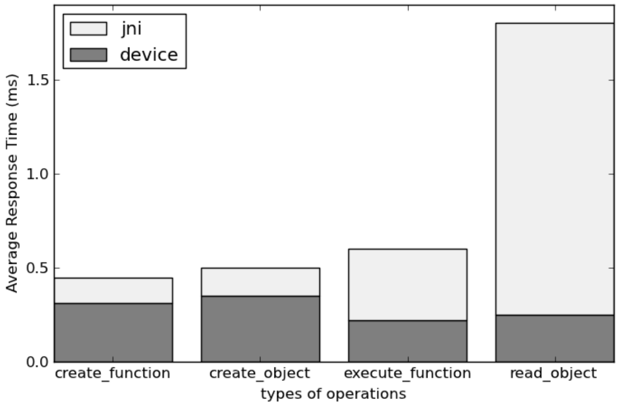


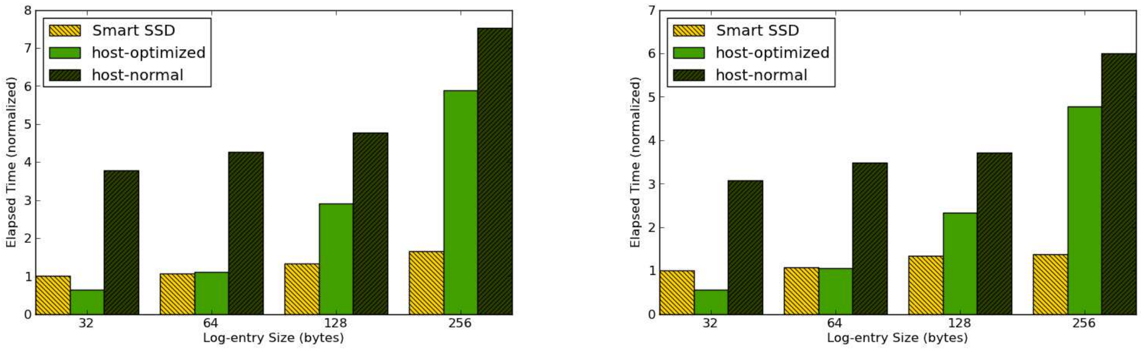
Fig. 7. Overhead of an object interface implemented on top of SATA protocol; the raw I/O latency of the create command is higher than that in the execute and read commands, because it requires two vendor-specific commands. The JNI overhead of the read command is high due to the additional processing for polling interval and memory copy overhead.

C. Applications

As discussed in the previous section, applications should have high data selectivity and low computing complexity to benefit from the use of our prototype Smart SSD. We understand that this guideline, due to the hardware constraints of the current SSD architecture, limits the range of applications. However, we expect that the applications that can leverage Smart SSD will grow over time because the computing capability of SSDs continues to increase. SSDs need more computing power in terms of the number of cores or clock rate and more DRAM space as the capacity and interface bandwidth of SSDs increase.

This section focuses on the potential benefits that Smart SSD can deliver with real applications. For this purpose, we evaluate Smart SSD with two applications: a web-log analyzer that uses a fixed size data structure, and a data filter that returns the only page that contains valid data. These applications include searches through a fixed size data structure or accesses to a specific position within a flash page to retrieve information. As a result, the ISP engine can avoid a full page scan to find data of interest.

We use a real dataset named WorldCup98 which contains the requests made to the 1998 World Cup website [20]. We collected 7,000,000 distinct log entries combining the available dataset, and generated four data sets by adding padding to each log entry to create different sized log-entries; this changes the size of the raw workloads that need to be read into either a host DRAM or a device DRAM. In our example, the size of the smallest log entry is 32 bytes, and that of the largest entry is 256 bytes. The sizes of the data set with the smallest log entries and the largest entries are 240 MB and 1.8 GB, respectively.



(a) Number of accesses per geographical region

(b) Top 5 file types accessed per geographical region

Fig. 8. Performance of log-analysis with two query scenarios. Smart SSD benefits from high internal bandwidth and no data transfer between a host and a device, losing performance slowly as the dataset size increases. However, the host-only computing models suffer from large data movements on larger datasets, decreasing performance.

We create two query scenarios that retrieve information from the log. The first scenario is to keep track of the number of accesses per geographical region, which requires one access to a field in each entry and storing of intermediate results. This query needs a reduce step to sum up the intermediate results per region. In the second scenario, a tasklet keeps track of the number of accesses per file type per region, and picks the top N file types per region. During the processing, it accesses one more field per entry (2 in total), and the results are not only merged but also sorted. Therefore, the second query requires more DRAM accesses than the first one.

1) Performance: To evaluate the performance of in-storage and in-host data processing, we implement three versions of Hadoop applications. *Host-normal* denotes a log analysis MapReduce application that uses a normal SSD and the default Hadoop protocol, and *host-optimized* is the modified version of Hadoop that minimizes the overhead of merging and sorting by storing intermediate results sorted and compacted only in in-memory data structures designed for log-analysis. With Smart SSD, the intermediate results of the *Smart SSD* application are stored and sorted inside the device, and are not sent to the host.

Figure 8 shows the performance of two scenarios with three types of clients. It shows that the performance gap between *Smart SSD* and the others becomes wider with larger log entries. As the log entry size becomes larger, the number of entries per page decreases, which reduces the number of DRAM accesses accordingly. In addition, the amount of data to be sent to the host increases with larger entries. The high data transfer cost dominates the DRAM access penalty with large log entries. These results are consistent with the results of the micro-benchmark in the previous section. One interesting observation in this experiment is the performance in the case of the 32 byte entry. *Host-optimized* shows better performance than the *Smart SSD*, because the *Smart SSD* suffers from too many DRAM accesses, but receives little benefit from traffic reduction; 256 DRAM accesses are made per page while 240 MB of data are transferred through the 3 Gbps channel. While the same amount of data is transfered in *host-normal* and *host-optimized*, *host-normal* exhibits much lower performance due to the large amount of intermediate data, incurring frequently flushing of data, and high merging and sorting overhead.

The second query (Figure 8 (b)) shows a similar performance trend to the first query. One difference is that the *Smart SSD* catches up with the *host-optimized* slowly, and the performance gap is narrow due to the additional DRAM access per entry and the sorting process. This sensitivity to the number of DRAM accesses can be improved by instruction-level optimizations, specifically considering the behavior of the load operation. However, we believe that small upgrades of the SSD architecture can alleviate this problem further, providing more flexibility in determining the tasks to offload. In terms of I/Os to flash, while both host-side clients need to read the whole data set from the device, *i. e.*, 1.8 GB with a log entry size of 256 bytes, the Smart SSD client does not generate any data traffic over the system bus. This allows other devices in the host system to use the system bus while Smart SSD processes data in parallel.

2) CPU Usage: To evaluate the CPU overhead on the host side, we measure the CPU utilization of the host system while the three versions of Hadoop applications are running as shown in Figure 9. We illustrate the one with a log-entry of 256 bytes since all configurations show a similar tendency. The kernel is mostly waiting for the data from the device in the idle state, and the user state indicates the CPU time spent on the Hadoop application. The kernel is processing interrupts, and I/O and process scheduling in the kernel state.

As shown in Figure 9, the system with the Smart SSD client consumes very few host CPU cycles compared to the host-only versions of the applications. Even though the host CPUs occasionally wake up to check if the results from Smart SSD

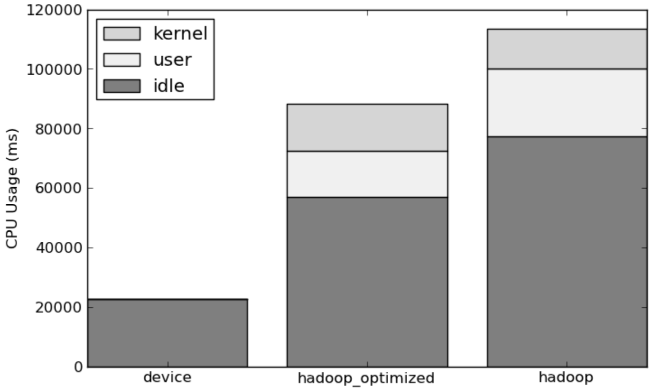


Fig. 9. CPU usage (including all cores). The Smart SSD version of the application uses almost no host resources while the host versions of applications require kernel time to process I/Os and user time to process map processes.

are available, they are mostly in the idle state waiting for a job to be finished so the CPUs can process other computation jobs or processes concurrently.

With a log-entry of 256 bytes, the host versions of the application are in the idle and kernel state for three or four times longer than the *Smart SSD* versions, waiting for the data from the device and delivering it to an application. The CPU time consumed in these periods increases in proportion to the size of data to be transferred and the number of other active process running in the background. Although large burst requests are preferred to maximize the throughput of SSDs, these requests may also increase the kernel and user time, creating an interval between I/O requests and thus lowering the utilization of devices' internal parallelism. For the two host versions, the large amount of intermediate data makes *host-normal* experience more idle time due to I/O waits than *host-optimized*.

While storage systems using SSDs can solve this problem by redesigning the data layout, I/O scheduler, and applications for SSDs, the Smart SSD model allows devices to handle this throughput issue by merely rewriting the applications to internally create I/O requests based on the number of available internal bandwidth at the time of the execution. The minimized use of the host CPUs further improves the overall energy efficiency as will be described later.

3) Energy Efficiency: Since Smart SSD uses low power processors and minimizes the use of host resources to process a tasklet, it is expected to consume less power than the host-based approaches. We plugged the host machine into a power meter device to measure its entire power consumption so we can measure the total energy consumed by the host system. Then, we ran the first query scenario for three versions of the application and measured the power consumption during the run. In order to avoid effects from other applications or the previous state of an execution, we injected a pause until the system became idle before resuming the next step. As shown in Figure 10, for each log-entry size, the corresponding dataset is synchronously written to the device, and the contents in the host page cache are flushed in the *wr* phase. Then, we force the application to sleep for 5 seconds, and then start the tasklet in the *exec* phase. When the tasklet terminates, we inject another pause before resuming with the next dataset.

Figure 10 shows the power consumption of each version of the log analyzer. In the idle state, the system consumes around 44 watts. In the exec phase, however, the power consumption of the host versions of applications increases by around 35 watts, and *host-optimized* and *host-normal* versions consume 75–82 watts in total. On the other hand, in-storage processing only increases the total energy consumption by 0.5–0.8 watts, saving more than 50% of the energy that the total host system spent. This is because the data movement is minimized in the Smart SSD model, and thus the host CPUs and I/O subsystem are not involved during the processing phase. Note that the *wr* phase is not needed when a tasklet works with existing data. In terms of energy consumption, offloading data processing or filtering jobs to the device before entering a big computation could be helpful even though the actual performance of the tasklet in the device is slower than in the host. This approach allows the combination of low-power processing in the device, as done in FAWN [3], with host-based processing for complex tasks to lower overall power consumption.

4) Data Filtering: To mitigate the impact of slow DRAM access in the current SSD architecture, Smart SSDs can be used to conduct a simple data filtering when excessive DRAM accesses are expected within a page. For instance, instead of identifying all matches in the page, we can send the page with the first match, and let the host do the complex computation with the page. As a result, the device can reduce the number of data transfers while the host processes less data. Since these filtering jobs can be issued to the devices before the host system starts long computations or processing other data, this tasklet could improve the overall data processing rate in a large storage system.

To show the effects of data filtering, we design a tasklet that returns the logical block addresses (LBAs) that meet a certain criteria given by a user application, instead of collecting and returning the results. Although the tasklet can also return the actual filtered data instead of the LBAs, we decided to return LBAs in order to allow the host system to choose the appropriate time to process these data. Once a match is found during the search, this tasklet stores the LBA of the page without looking at the values of other structures in the same page. Performance-wise, it can still suffer from slow memory accesses when data selectivity is very low but because this tasklet is processed in parallel with the host tasklets, it can improve the overall efficiency of the system by saving host resources.

We simulate an application that reads and compares one integer variable within a 32 byte data structure, just like the case of 256 accesses in Figure 6. On a 1 GB dataset with 60% selectivity, the performance of the data filtering tasklet

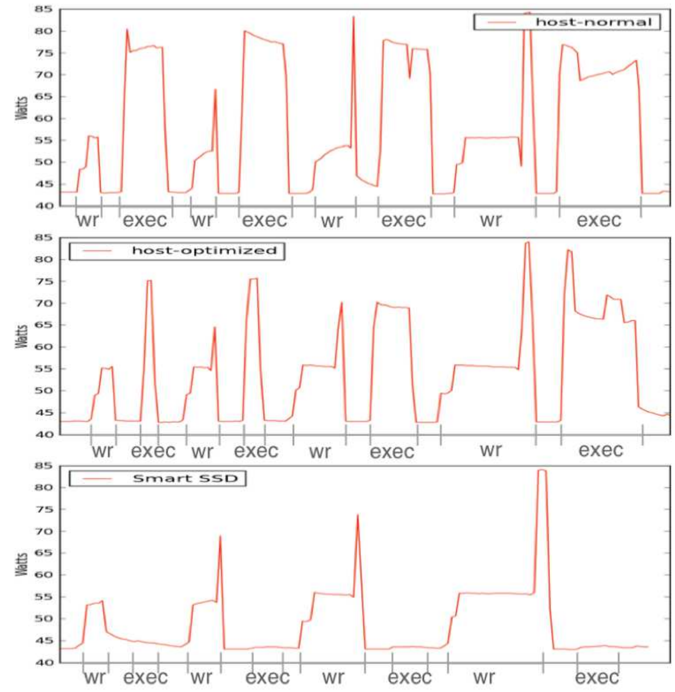


Fig. 10. For each dataset, data to be processed is first written (*wr* phase) before executing the tasklet (*exec* phase). In the *exec* phase, the system that uses Smart SSD consumes less than 50% of the energy compared to the system that uses host-side processing, including idle power consumption, due to the internal, low-power processing.

becomes 40% faster than doing the comparisons for the entire data on the device, but it was 18% slower than doing it on the host. So, depending on the characteristics of data and computation, data filtering can be used to compensate for the low performance of the embedded processors while reducing the total amount of data transferred to the host.

D. Future SSD Architecture for In-Storage Processing

While the current SSD architecture provides enough processing power to support basic read and write operations, more hardware components and software layers are required to fully utilize the benefit of in-storage processing. Most importantly, CPU caches and high bandwidth between CPUs and DRAM need to be provided to avoid performance degradation due to the memory accesses. Adopting an application processor (or core) for tasklets would be helpful to avoid possible interferences between workloads generated by in-storage processing and user applications. This allows device tasklets and host jobs to work at the same time on the same device, thus making the system more balanced in terms of resource usage.

VI. Future Work

We are exploring the use of Smart SSDs as a data processing engine in a large distributed environment such as social-network graph traversing or automatic replication management leveraging the internal information about the reliability of each flash cell. While the benefits from multiple Smart SSDs are expected to be similar to the aggregation of the benefits from individual Smart SSDs, one of the main issues in those systems we want to look at is the scheduling policy across multiple Smart SSDs and hosts. Since the performance of a job depends on the complexity of a job and the current load of a host and a device, we need to find a cost function for each job and assign a job to the device or the host depending on the cost. We are also looking at the use of Smart SSDs in large storage systems as a more advanced cache that has a large capacity and is capable of finding an entry quickly with extremely low power.

There remain several other optimizations that could improve the usability of Smart SSD. For example, the current object-based I/O library can be implemented inside the storage stack in the kernel, thus allowing the devices to access tasklets using a standard POSIX interface, *i. e.* `fadvise()`. Once we find a cost function for in-storage processing, this layer can provide transparent job scheduling as well.

As for software, instead of using a cross-compiled C binary, support for scripting language in writing tasklets would be helpful in providing security mechanisms, such as sandboxing and authentication, to prevent malicious tasklets from destroying data or even devices. However, we believe that adding this feature to the current commodity SSDs is not practical because the firmware is not a general operating system; it is a state machine optimized for read and write requests, and it does not support processes, virtual memory, or even dynamic memory allocation. Therefore, for sandboxing, running a virtual machine based language, such as Java or Python, is not possible. Developing a new compiler or an interpreter that runs inside the firmware can cause high latencies in processing normal read and write requests. Our current implementation can check illegal parameters to the FTL functions, but cannot check for illegal memory manipulation that can cause the device to crash.

Having an application processor that can run a lightweight operating system could be helpful to alleviate this problem, allowing application processors to run interpreters and issue I/O requests to the firmware processors. Then, an abstraction layer of hardware components and FTL functions in the device could be provided to tasklet developers so they can dynamically add or remove their own tasklets without having specific knowledge about the implementation of a SSD firmware.

VII. Conclusion

Modern datacenters face a common challenge that the power supply limits their computing capacity. A trend to tackling this challenge is to consolidate servers through system optimization and low power technologies, releasing more space and reducing operating cost. Smart SSDs can contribute to this server consolidation by providing low power data processing capability. The ARM processor employed by SSDs consumes a fraction of power compared to the general purpose processors. Even though new Intel processors such as Ivy Bridge-based Intel Xeon processors and Atom-based Centerton processors are expected to consume less power, servers are still power-hungry in general. In this paper, we showed the potential of Smart SSD to not only reduce power consumption but also improve performance.

Unlike rotational hard drives where mechanical movement determines the performance, the performance of SSDs has improved with more powerful processors and increasing hardware components such as channels and memory as well as better FTL algorithms. Despite the functionality that makes it a complete low-power computing system with CPUs, memory, and storage, a host system still thinks of an SSD as a dumb I/O device, passing up the chance of optimizing its data flow. To address this shortcoming, we explored the potential benefits of SSDs as data processing nodes and identified the limitations of the current SSD architecture. We presented a multi-functional storage device, Smart SSD, that harnesses the processing power of a device using an object-based communication protocol. Smart SSDs rely upon tasklets: independent I/O tasks of an application running inside the device. To allow applications to better use SSDs, we developed a programming interface to execute tasklets based on MapReduce. We implemented the Smart SSD features in the firmware of a Samsung SSD and modified the Hadoop Core and MapReduce framework to use tasklets as a *map* or a *reduce* function. To evaluate our prototype, we ran a microbenchmark and a log-analysis application on 7,000,000 entries in both a device and a host. We found that under the current SSD architecture, excessive memory accesses will make the tasklet execution slower than in the host due to the high memory latency and low processing power. However, the results with a log-analysis example show that our Smart SSD prototype consumes 2% of the energy compared to the host versions of applications in processing a given workload, saving more than 50% of the total energy while providing up to 2–3× better performance, depending on the entry size.

ACKNOWLEDGMENTS

We would like to thank our colleagues in the Storage Systems Research Center (SSRC) and the System Architecture Lab and Flash Solution Lab at Samsung for their input and guidance.

This research was supported in part by the National Science Foundation under awards IIP-0934401 and CCF-0937938. We would also like to thank the industrial sponsors of the SSRC, including EMC, Hewlett Packard Laboratories, Hitachi, Huawei, IBM Research, LSI, NetApp, Northrop Grumman, Permabit, and Samsung, for their generous support.

REFERENCES

[1] A. Bialecki and M. Cafarella and D. Cutting and O. Malley. Hadoop: A framework for running applications on large clusters built of commodity hardware. http://lucene.apache.org/hadoop/, 2005.

[2] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms, and evaluation. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, Oct. 1998.

[3] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 1–14, Big Sky, MT, Oct. 2009.

[4] S. Boboila, Y. Kim, S. Vazhkudai, P. Desnoyers, and G. Shipman. Active flash: Out-of-core data analytics on flash storage. In *Proceedings of the IEEE Conference on Mass Storage Systems and Technologies (MSST 2012)*, Apr. 2012.

[5] N. Corporation. The Netezza Data Appliance architecture: a platform for high performance data warehousing and analysitics, 2010.

[6] Curtis E. Stevens. ATA command pass-through. ftp://ftp.t10.org/t10/document.04/04-262r1.pdf, 2005.

[7] A. Huffman. Nvm express, revision 1.0c. www.nvmexpress.org, Feb. 2012.

[8] S. E. Inc. Samsung solid state drive basics. http://www.samsung.com/global/business/semiconductor/product/flash-ssd/overview, 2010.

[9] Y. Kang, J. Yang, and E. L. Miller. Object-based SCM: An efficient interface for storage class memories. In *Proceedings of the 27th IEEE Conference on Mass Storage Systems and Technologies*, May 2011.

[10] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (IDISKs). *SIGMOD Record*, 27(3), Aug. 2004.

[11] S. Kim, H. Oh, C. Park, S. Cho, and S.-W. Lee. Fast, energy efficient scan inside flash memory SSDs. In *Proceeedings of the International Workshop on Accelerating Data Management Systems (ADMS)*, Sept. 2011.

[12] R. Mueller and J. Teubner. FPGA: What's in it for a database? In *Proceedings of the international conference on Management of data*, SIMOD'09, 2009.

[13] R. Mueller, J. Teubner, and G. Alonso. Data processing on FPGAs. *Proceedings of the VLDB Endowment*, 2(1):910–921, Aug. 2009.

[14] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, Mar. 2008.

[15] E. A. Ozkarahan, S. A. Schuster, and K. C. Smith. RAP—associative processor for database management. In *AFIPS*, 1975.

[16] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. Active disks for large-scale data processing. *IEEE Computer*, 34(6), 2001.

[17] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *Proceedings of the international conference on Very Large Data Bases*, VLDB'98, Aug. 1998.

[18] S. Y. W. Su and G. J. Lipovski. CASSM: A cellular system for very large data bases. In *International Conference on Very Large Data Bases*, pages 456–472, Sept. 1975.

[19] R. Weiss. A technical overview of the Oracle Exadata database machine and exadata storage server. www.oracle.com/us/products/database/exadata-technical-whitepaper-134575.pdf, March 2012.

[20] Worldcup98 internet traffic archive. http://ita.ee.lbl.gov/html/contrib/WorldCup.html.

[21] J. Yang, D. B. Minturn, and F. Hady. When poll is better than interrupt. In *Usenix Conference on File and Storage Technologies (FAST 2012)*, Feb. 2012.