Solid State Storage Technologies

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
NVMe (I)

- The industry standard interface for high-performance NVM storage
  - NVMe 1.0 specification in 2011 (now 1.3)
  - Supported by major OSes: Windows, Linux, Solaris, …

- PCIe-based
  - Low latency: direct connection to CPU
  - Scalable performance: 1 GB/s per lane, up to 32 lanes
  - No HBA required: reduced power & cost

- Form factors
  - Add-in-Card, M.2, BGA, etc.
NVMe (2)

- Deep queue: 64K commands/queue, up to 64K queues
- Streamlined command set: only 13 required commands
- One register write to issue a command ("doorbell")
- Support for MSI-X and interrupt aggregation
NVMeDirect Framework

All-Flash Array

- **Interfaces**
  - 10Gb/40Gb Ethernet (iSCSI) or 16Gb Fibre Channel or PCIe
  - SAS or NVMe SSDs

- **Functionalities**
  - Volume management
  - Virtualization support
  - RAID
  - Snapshot
  - Deduplication
  - Compression, …
Traditional Block Interface

- SATA/SCSI/SAS
  - Read (sector #, length)
  - Write (sector #, length, data)
  - No block-level liveness information
  - No high-level semantics on data
  - Several “unwritten contracts” do not hold for SSDs
    - Sequential accesses are several tens of times better than random accesses
    - Distant LBNs lead to longer seek times
    - Data written is equal to data issued
    - ...
Extending Block I/F

- **TRIM command**
  - “The data in the specified sectors is no longer needed”
  - ATA interface standard (T13 technical committee)
  - Non-queued command
  - SATA 3.1 introduces the Queued TRIM command
Atomic Write

- Transaction support for multi-block writes
  - Simplifies file systems and DBMSes

Multi-streamed SSD (I)

- Previous write patterns (= current state) matter
Multi-streamed SSD (2)

- Mapping data with different lifetime to different streams
- Standardized in T10 SCSI/SAS (2015), NVMe 1.3 (2017)
Multi-streamed SSD (3)

- Cassandra with Multi-streamed SSD
Multi-streamed SSD (4)

- Cassandra’s normalized updated throughput with 5 streams
**OSSD: Object-based SSD (1)**

- **OSD (Object-based Storage Device)**
  - Virtualizes physical storage as a pool of objects
  - Offloads space management to storage devices
  - Standardized as a subset of SCSI command set

![Block interface](image)

**Operations**
- read block
- write block

**Security**
- Weak
- Full disk

**Allocation**
- External

![Object interface](image)

**Operations**
- read object offset
- write object offset
- create object
- delete object

**Security**
- Strong
- Per Object

**Allocation**
- Local
OSSD: Object-based SSD (2)

- **OSD storage model**

  ![Diagram of OSD storage model](image)

  - Host
    - Application
    - System Call Interface
    - File System User Component
    - File System Storage Management
    - Sector/LBA Interface
    - Block I/O Manager
    - Physical Media
  
  - Storage Device
    - Traditional
    - OSD
      - OSD Interface
      - OSD Storage Management
      - Block I/O Manager
      - Physical Media
OSSD: Object-based SSD (3)

OSSD: Object-based SSD (4)

- Simplified host file system
  - No need for SSD-specific parameter tuning

- More efficient management of flash storage
  - Block-level liveness
  - Metadata separation
  - Object-aware storage management (allocation, dedup, ...)

- Application-aware storage management
  - Application hints, QoS

- Storage virtualization
  - Pooling, tiering, caching, backup, replication, etc.
In-Storage Computing (1)

- Samsung ISC SSD Prototype
  - Commodity SSD: Samsung PM1725 NVMe with the ISC feature
  - PCIe 3.0x4
  - 800 GB

- Software
  - C++11
  - C++STL
  - G++
  - Software emulator
In-Storage Computing (2)

- ISC Application Development Process

1. C/C++ - Support C++11/STL
2. X86 Compile
3. ISC SSD Emulation
4. ARM Cross compile
5. Download /isc/myprogram/ssdlet
6. Run /isc/myprogram/host
In-Storage Computing (3)

- ISC Dataflow Programming Model
In-Storage Computing (4)

- Example: Simple Key-Value Store

```cpp
class KVStore
    public SSDLet<IN_TYPE<SR(string), SR(string), SR(string)>,
    OUT_TYPE<SR(string)>>
{
    public:
    map<string, string> table;

    void run()
    {
        auto in_command = getInPort<0>();
        auto in_key = getInPort<1>();
        auto in_value = getInPort<2>();
        auto out_value = getOutputPort<0>();

        string command, key, value;
        while (true)
        {
            if (lin_command.get(command))
                break;

            if (command == "get")
            {
                if (lin_key_get(key))
                    break;
                out_value.put(table[key]);
            }
            if (command == "put")
            {
                if (lin_key_get(key) || lin_value_get(value))
                    break;
                table[key] = value;
            }
        }
    }
}

int main(int argc, char *argv[])
{
    SSD ssd("/dev/nvme0n1p1");
    module_id_t mid = ssd.loadModule(File(ssd, ".libkvstore.so");
    Application app(ssd);

    SSDLet kvstore(app, mid, "KVStore");
    auto out_command = app.connectTo<String>(kvstore.in(0));
    auto out_key = app.connectTo<String>(kvstore.in(1));
    auto out_value = app.connectTo<String>(kvstore.in(2));
    auto in_result = app.connectTo<String>(kvstore.out(0));
    app.start();

    string command, key, value;
    while (std::cin >> command)
    {
        if (command == "get")
        {
            out_command.put(command);
            std::cin >> key;
            out_key.put(key);
            in_result.get(value);
            std::cout << value << std::endl;
        }
        else if (command == "put")
        {
            out_command.put(command);
            std::cin >> key >> value;
            out_key.put(key);
            out_value.put(value);
        }
        else break;
    }
    return 0;
}
```