Biscuit: A Framework for Near-Data Processing of Big Data Workloads

Boncheol Gu
Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, Duckhyun Chang

Memory Business, Samsung Electronics
Near-data processing moves computation to data.

- Computation is performed right at the data source.
- Efficient when the cost of moving data is very high.
Near-data processing moves computation to data.

- Computation is performed right at the data source.
- Efficient when the cost of moving data is very high

Most prior work focuses on proving the concept of NDP.

- Little attention to designing and realizing a practical framework
- Realistic large application studies were omitted.
Samsung NVMe SSD (PM1725)

- V-NAND TLC with Epic Controller
- Sequential Read up to 3,100 MB/s
  Sequential Write up to 1,800 MB/s
- 4KB sustained Random Read up to 750K IOPS
  4KB sustained Random Write up to 120K IOPS
- Power-loss Protection
A user-programmable NDP framework for SSDs and data-intensive applications
A user-programmable NDP framework for SSDs and data-intensive applications

- C++ language support (C++ standard library)
- Dynamic loading of user programs
- Mutithreading, multi-core support
- The 1st reported product-strength NDP system
### SSD Hardware

![Inside of PM1725](image)

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host interface</td>
<td>PCIe Gen.3 ×4 (3.2 GB/s)</td>
</tr>
<tr>
<td>Protocol</td>
<td>NVMe 1.1</td>
</tr>
<tr>
<td>Device density</td>
<td>1 TB</td>
</tr>
<tr>
<td>SSD architecture</td>
<td>Multiple channels/ways/cores</td>
</tr>
<tr>
<td>Storage medium</td>
<td>Multi-bit NAND flash memory</td>
</tr>
<tr>
<td>Compute resources</td>
<td>Two ARM Cortex R7 cores @750MHz with MPU</td>
</tr>
<tr>
<td>On-chip SRAM</td>
<td>&lt; 1 MiB</td>
</tr>
<tr>
<td>DRAM</td>
<td>≥ 1 GiB</td>
</tr>
</tbody>
</table>

**Hardware pattern matcher** on each flash memory channel.

- To search given bit patterns in the flash memory
SSD Hardware

[Inside of PM1725]

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host interface</td>
<td>PCIe Gen.3 ×4 (3.2 GB/s)</td>
</tr>
<tr>
<td>Protocol</td>
<td>NVMe 1.1</td>
</tr>
<tr>
<td>Device density</td>
<td>1 TB</td>
</tr>
<tr>
<td>SSD architecture</td>
<td>Multiple channels/ways/cores</td>
</tr>
<tr>
<td>Storage medium</td>
<td>Multi-bit NAND flash memory</td>
</tr>
<tr>
<td>Compute resources</td>
<td>Two ARM Cortex R7 cores @750MHz with MPU</td>
</tr>
<tr>
<td>On-chip SRAM</td>
<td>&lt; 1 MiB</td>
</tr>
<tr>
<td>DRAM</td>
<td>≥ 1 GiB</td>
</tr>
</tbody>
</table>

Hardware pattern matcher on each flash memory channel.

- To search given bit patterns in the flash memory

Limitations

- Low compute power, no cache coherence, a small amount of fast memory, no MMU, and restrictive synchronization primitives
Cooperative multithreading

- A limited form of multithreading (fiber as a scheduling unit)
- Less context switching overhead
- Safe resource sharing without locking
Biscuit Runtime

Cooperative multithreading

- A limited form of multithreading (fiber as a scheduling unit)
- Less context switching overhead
- Safe resource sharing without locking

Shared nothing architecture

- All data transmission among threads through I/O ports
- Enforced by the programming model and APIs
- C++11 move semantics supported
Cooperative multithreading

- A limited form of multithreading (fiber as a scheduling unit)
- Less context switching overhead
- Safe resource sharing without locking

Shared nothing architecture

- All data transmission among threads through I/O ports
- Enforced by the programming model and APIs
- C++11 move semantics supported

Dynamic loader for user programs

- User program as position-independent code (PIC)
- Symbol relocation to locate each program in a separate address space
Biscuit System Architecture

Host system
- host-side program
  - host-side Task
  - host-side Task
  - *libsisc*
- NVMe driver

SSD-side module
- *libslet*
  - SSD-side Task
  - SSD-side Task
  - SSD-side Task

SSD firmware
- fiber → fiber → fiber

Biscuit runtime
- I/O requests

Host interface controller
- CPUs
- SRAM
- DRAM
- FMC

Biscuit-enabled SSD
- Channel #0
  - NAND...
- Channel #(n_{ch}-1)
  - NAND...
Development Process

Host-side task

① Write the codes

② X86 compile

③ ARM cross compile

SSD-side task

④ Copy the module into `/var/isc/slets (/dev/nvme0n1)

⑤ Run the host program

Host Computer

Boncheol Gu et al.
Development Process

1. Write the codes
2. X86 compile
3. ARM cross compile
4. Copy the module into /var/isc/slets (/dev/nvme0n1)
5. Run the host program
Development Process

1. Write the codes

2. X86 compile

3. ARM cross compile

4. Copy the module into /var/isc/slets (/dev/nvme0n1)

5. Run the host program
Development Process

① Write the codes

② X86 compile

③ ARM cross compile

④ Copy the module into /var/isc/slets (/dev/nvme0n1)

⑤ Run the host program
Experimental Setup

H/W setup

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>Dell PowerEdge R720 server</td>
</tr>
<tr>
<td>CPU</td>
<td>2 Intel Xeon(R) CPU E5-2640 (12 threads per socket) @2.50GHz</td>
</tr>
<tr>
<td>Memory</td>
<td>64 GiB DRAM</td>
</tr>
<tr>
<td>OS</td>
<td>64-bit Ubuntu 15.04</td>
</tr>
</tbody>
</table>

Basic performance results

- Communication latency, data read bandwidth and latency

Application level results

- String search, pointer chasing, and DB scan/filtering

Boncheol Gu et al.
### Experimental Setup

#### H/W setup

<table>
<thead>
<tr>
<th>System</th>
<th>Dell PowerEdge R720 server</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2 Intel Xeon(R) CPU E5-2640 (12 threads per socket) @2.50GHz</td>
</tr>
<tr>
<td>Memory</td>
<td>64 GiB DRAM</td>
</tr>
<tr>
<td>OS</td>
<td>64-bit Ubuntu 15.04</td>
</tr>
</tbody>
</table>

#### Basic performance results

- Communication latency, data read bandwidth and latency

#### Application level results

- String search, pointer chasing, and DB scan/filtering

#### Notations

- **Conv**: system configuration with a default conventional SSD
- **Biscuit**: system configuration with the Biscuit framework on the SSD
Basic Performance Results - Data Read Bandwidth

- **Conv**: transfer data to the host-side program
- **Biscuit**: transfer data to the SSD-side module (i.e., "internal read")

Biscuit exploits the underutilized internal bandwidth.
Data analytics with a real DB engine

- MariaDB 5.5.42 (XtraDB)
- TPC-H dataset at a scale factor of 100 (160 GiB)

\(^1\) a fraction of pages that satisfy filter conditions from the total number of pages in the table
Data analytics with a real DB engine

- MariaDB 5.5.42 (XtraDB)
- TPC-H dataset at a scale factor of 100 (160 GiB)
- We modified the query planner to
  1. identify a candidate table amenable for offloading;
  2. estimate its selectivity\(^1\) using a sampling method;
  3. determine whether the table is indeed a good target (based on a selectivity threshold);
  4. and finally offload the identified filter to the SSD.

\(^1\)a fraction of pages that satisfy filter conditions from the total number of pages in the table
Query 1\(^1\) (single filter predicate, selectivity for shipdate: 0.02)

SELECT l_orderkey, l_shipdate, l_linenumber
FROM lineitem
WHERE l_shipdate = '1995-1-17'

Query 2\(^1\) (more complex WHERE clause, selectivity for shipdate: 0.04)

SELECT l_orderkey, l_shipdate, l_linenumber
FROM lineitem
WHERE (l_shipdate = '1995-1-17' OR l_shipdate = '1995-1-18')
    AND (l_linenumber = 1 OR l_linenumber = 2)

High filtering ratio with a low computational complexity

\(^1\)“Ibex - An Intelligent Storage Engine with Support for Advanced SQL Off-loading”, VLDB 2014

Boncheol Gu et al.
• Biscuit achieves speed-ups of about $11 \times$ and $10 \times$.
  
  • Large internal bandwidth of Biscuit
  • Rapid scanning of the dataset by the hardware pattern matcher

• Execution times on Biscuit were very consistent.
Some extra work, such as synchronizing the buffer cache, performed after the query execution (included in the results).

Biscuit consumes more power during query processing.
- It keeps the target SSD busy and exploits nearly full bandwidth.

The power is of the whole system including the server and the target SSD.
Biscuit achieves significantly lower energy consumption thanks to its reduced execution time.
• Running all queries, Conv takes nearly two days, while Biscuit takes about 13 hours (3.6x speed-up).
  • Top 5 queries take 70+% of total execution time.
Conclusions

• We presented the design and implementation of Biscuit, an NDP framework built for high-speed SSDs.

• With Biscuit, we pursued achieving high programmability on distributed resources including processing units of SSDs as well as host CPUs.

• Biscuit is the first reported product-strength NDP system implementation.

• We successfully ported Biscuit on small and large data-intensive applications including MariaDB.

• Biscuit accomplished the performance improvement of up to 166x for TPC-H queries (average 6.1x improvement).
Biscuit: A Framework for Near-Data Processing of Big Data Workloads

Boncheol Gu
Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, Duckhyun Chang

Memory Business, Samsung Electronics
Programming Model

Host-side program (coordinator)

App. 1

SSDlet

in

out

SSD-side module (computation units)

// do computation

App. 2

proxy

libsisc

Biscuit runtime instance

SSDlet

instance

libsisc

// do computation

SSD-side module (computation units)

Computation model

• Specifying computation using its input and output data

• **Libslet** to define SSD-side tasks and perform file I/O.

Boncheol Gu et al.
Programming Model

Computation model
- Specifying computation using its input and output data
- **Libslet** to define SSD-side tasks and perform file I/O

Coordination model
- Creating and managing tasks
- Establishing producer/consumer relationship
- **Libsisc** to invoke and coordinate execution of SSD-side tasks
• A simple C++ program written with Biscuit APIs.
• A unit of execution independently scheduled, represented by the **SSDLet** class

**Example (User-defined SSDlet)**

```cpp
class UserTask
  : public SSDLet<IN_TYPE<int32_t>,
    OUT_TYPE<int32_t, bool>,
    ARG_TYPE<double>> {
public:
  void run() override {
    auto in = getInputPort<0>();
    auto out0 = getOutputPort<0>();
    auto out1 = getOutputPort<1>();
    double& value = getArgument<0>();

    // do some computation
  }
}
```

• A user task is derived from the SSDLet class.
  • Its **run** method is overridden to describe the execution of the task.
  • The type information is used to perform **strict type checking** at compile/run time.
Wordcount Example

- Mapper
- Shuffler
- Reducer

<word, frequency>

wordcount module
filename
host-side
program

Mapper
Mapper
Shuffler
Reducer
Reducer

<word, vec>
<word, vec>

host-side program

<count, frequency>
class Mapper : public SSDLet<OUT_TYPE<std::pair<std::string, uint32_t>>, ARG_TYPE<File>> {
  
public:
  void run() {
    auto& file = getArgument<0>();
    FileStream fs(std::move(file));
    auto output = getOutputPort<0>();
    while (true) {
      ...
      if (!readline(fs, line)) break;
      line.tokenize();
      while ((word = line.next_token()) != line.cend()) {
        // put output (i.e., each word) to the output port
        if (!output.put({std::string(word), 1})) return;
      }
    }
  }
}
```c
int main(int argc, char *argv[]) {
    SSD ssd("/dev/nvme0n1");
    auto mid = ssd.loadModule(File(ssd, "/var/isc/slets/wordcount.slet"));

    // create an Application instance and proxy SSDLet instances
    Application wc(ssd);
    SSDLet mapper1(wc, mid, "idMapper", make_tuple(File(ssd, filename)));
    SSDLet shuffler(wc, mid, "idShuffler");
    SSDLet reducer1(wc, mid, "idReducer");
    ...
    // make connections between SSDLets and from Reducers back to the host
    wc.connect(mapper1.out(0), shuffler.in(0));
    wc.connect(shuffler.out(0), reducer1.in(0));
    auto port1 = wc.connectTo<pair<string, uint32_t>>(reducer1.out(0));
}
...  
// start application so that all SSDlets would begin execution  
wc.start();  
pair<string, uint32_t> value;  
while (port1.get(value) || port2.get(value)) // print out <word,freq> pairs  
    cout << value.first << "\t" << value.second << endl;  
// wait until all SSDlets stop execution and unload the wordcount module  
wc.wait();  
ssd.unloadModule(mid);  
return 0;  
}
Communication through ports

- (a) Inter-SSDlet ports: among SSDlet instances belonging to a single Application instance
- (b) Host-to-device ports: between an SSDlet instance and a host program
- (c) Inter-application ports: between two SSDlets from different Application instances

The APIs are strongly typed and implicit type conversion is not allowed.
• Libslet bridges the Biscuit runtime and SSDlet instances.
• An SSDlet instance obtains a function table from the runtime.
  • Key functions of memory management, scheduling and file I/O.
• Biscuit receives functions necessary to create and manage SSDlet instances.
• Symbol relocation is performed locating each SSDlet in a separate address space.