Chameleon: A High Performance Flash/FRAM Hybrid Solid State Disk Architecture

Jin Hyuk Yoon, Eyee Hyun Nam, Yoon Jae Seong, Hongseok Kim, Bryan S. Kim, Sang Lyul Min, and Yookun Cho

IEEE Computer Architecture Letters, Vol. 7, No. 1
Aug. 5, 2008
Speaker: Sehwan Lee
Introduction

- Chameleon
  - SSD (NAND flash memories) + Ferroelectric RAM

- FRAM
  - Non-volatile
  - Fast random reads
  - In-place-updates
Chameleon Hybrid SSD Architecture

Chameleon SSD Architecture Overview

The Chameleon SSD architecture uses a hybrid write buffering scheme that combines both block-level and page-level write buffers. In Chameleon, block-level write buffers are associated with logical blocks and are maintained in FRAM. The block-level write buffer is responsible for managing the space for block-level write operations. In contrast, page-level write buffers are used to manage page-level write operations and are maintained in SRAM. Each page-level write buffer is associated with a physical block in NAND flash memory, and the block-level write buffer is associated with a logical block. The block-level write buffer is used to maintain the block mapping table, which is an important FTL metadata that maps the storage space visible to the host to the physical blocks in NAND flash memory. The block mapping table is maintained in FRAM since it is subject to small random updates.

The extended DMA between the host interface and the flash memory subsystem is transparent to the flash memory. It contains ECC encoder/decoder units for correcting bit-flip errors from NAND flash memory. The controller provides a high-bandwidth data channel between the host and flash memory. It contains ECC encoder/decoder units for correcting bit-flip errors from NAND flash memory. The controller translates a high-level flash memory command from the host interface into a sequence of low-level flash memory operations. For this purpose, the controller implements both bus-level and chip-level interleavings to increase the effective read/write bandwidth of the flash memory.

The Chameleon SSD architecture also includes a host interface that implements device-side storage system protocols such as ATA and SCSI. The flash controller is responsible for managing various types of non-volatile data whose details will be explained next. The host interface implements device-side storage system protocols such as ATA and SCSI. The flash controller provides a high-bandwidth data channel between the host interface and the flash memory. It contains ECC encoder/decoder units for correcting bit-flip errors from NAND flash memory. The controller translates a high-level flash memory command from the host interface into a sequence of low-level flash memory operations. For this purpose, the controller implements both bus-level and chip-level interleavings to increase the effective read/write bandwidth of the flash memory.

The Chameleon SSD architecture uses NAND flash memory and FRAM to manage various types of non-volatile data whose details will be explained next. The host interface implements device-side storage system protocols such as ATA and SCSI. The flash controller provides a high-bandwidth data channel between the host interface and the flash memory. It contains ECC encoder/decoder units for correcting bit-flip errors from NAND flash memory. The controller translates a high-level flash memory command from the host interface into a sequence of low-level flash memory operations. For this purpose, the controller implements both bus-level and chip-level interleavings to increase the effective read/write bandwidth of the flash memory.

The host-visible storage space is divided into logical blocks whose size is equal to the physical block size times the degree of interleaving, respectively. The most important role of the FTL is to manage the space for block-level write operations. In Chameleon, the embedded processor together with SRAM is maintained for each page. Unlike the block-level write buffers where mapping information is associated with each write buffer block, the request is simply appended at the end of the log. Moreover, the boot time will proportionally increase with the capacity, causing a serious scalability problem. For example, even a 16 GB SSD (16 GB / 128 KB x 20^4) because of a scalability problem. For example, even

Fig. 1 shows the overall architecture of the Chameleon SSD.
**Chameleon Hybrid SSD Architecture**

**Chameleon SSD Architecture Overview**

- **Data blocks**
  - The Chameleon SSD manages various types of non-volatile data whose details will be managed by the FTL.
  - It provides an execution environment for the FTL.
  - The FPGA provides an execution environment for the FTL.

- **Types of Non-volatile Data in Chameleon**
  - The FTL is responsible for managing the block mapping table, which maintains the mapping between the host-visible storage space and the data storage on the flash memory.
  - The FTL manages the physical block size, the page size, and the degree of interleaving.

- **Write buffers**
  - Block-level write buffers: These buffers are used to store data temporarily before it is written to the flash memory.
  - Page-level write buffers: These buffers are used to store data temporarily before it is written to the flash memory.

- **Extended DMA**
  - The extended DMA between the host interface and the flash memory is used to optimize the performance of small random writes from the host.

- **Block merge operation**
  - When a new host write request arrives, the FTL selects the least recently used (LRU) write buffer block and allocates a new buffer block.
  - The data in the next write pointer is copied to the new buffer block.

- **Flash memory SSDs**
  - The capacity of flash memory SSDs is generally larger than 16 GB, but this can cause scalability issues.

- **Boot time**
  - The boot time for Chameleon is 2.621 seconds, assuming only one spare area read for each physical block at boot time.
  - The minimum boot time is approximately 2.621 seconds for a given configuration.

- **Chameleon Hybrid SSD Architecture**
  - The Chameleon Hybrid SSD architecture uses a hybrid write buffering scheme, combining block-level and page-level write buffers to optimize performance.
  - The architecture includes an embedded processor, SRAM, and FRAM for managing the FTL and flash memory operations.
  - The host interface communicates with the FTL controller via extended DMA to handle host write requests efficiently.

**Fig. 1. Chameleon SSD architecture.**
Chameleon Hybrid SSD Architecture

- Two Types of Non-volatile Data in Chameleon
  - Data blocks
  - Write buffers
    - Block-level write buffers
    - To optimize the performance of sequential writes from the host
    - Page-level write buffers
    - To optimize the performance of small random writes from the host
Chameleon Hybrid SSD Architecture

- Two Types of Non-volatile Data in Chameleon

NAND Flash array
Chameleon Hybrid SSD Architecture

- Two Types of Non-volatile Data in Chameleon

NAND Flash array

Data Blocks
- About 16 GB
Chameleón Hybrid SSD Architecture

- Two Types of Non-volatile Data in Chameleón

NAND Flash array

Page-level Write Buffers

- Size: 32MB
Chameleon Hybrid SSD Architecture

- Two Types of Non-volatile Data in Chameleon

NAND Flash array

```
```

Block-level Write buffers

- Size: 8MB
Chameleon Hybrid SSD Architecture

- Two Types of Non-volatile Data in Chameleon

NAND Flash array

- Size: 2MB
- To absorb small random writes (meta data)

FRAM
Chameleon Hybrid SSD Architecture

- Two Types of Non-volatile Data in Chameleon
  - Data blocks
    - Block Mapping Table
    - The size of logical block = the degree of interleaving * The size of physical block
    - FTL metadata are stored/maintained in...
      - FRAM
      - The spare area of each page
        - scalability problem
        - large boot time
Chameleon Hybrid SSD Architecture

- Two Types of Non-volatile Data in Chameleon
  - Write buffers: Block-level write buffers

- Three cases of write requests
  - The next host write request == The next write pointer
  - The next host write request > The next write pointer
  - The next host write request < The next write pointer
Two Types of Non-volatile Data in Chameleon

- Write buffers: Block-level write buffers

Three cases of write requests:

- The next host write request == The next write pointer
- The next host write request > The next write pointer
- The next host write request < The next write pointer
Two Types of Non-volatile Data in Chameleon

- Write buffers: Block-level write buffers

Three cases of write requests

- The next host write request == The next write pointer
Chameleon Hybrid SSD Architecture

- Two Types of Non-volatile Data in Chameleon
  - Write buffers: Block-level write buffers

- Three cases of write requests
  - The next host write request == The next write pointer
  - The next host write request > The next write pointer

The logical block number associated with each write-level buffer

Next write pointer
Two Types of Non-volatile Data in Chameleon

- Write buffers: Block-level write buffers

Three cases of write requests

- The next host write request == The next write pointer
- The next host write request > The next write pointer
- The next host write request < The next write pointer
Two Types of Non-volatile Data in Chameleon

Write buffers: Page-level write buffers

- Handling methods
  - All the write buffer blocks form a log as in a log-structured file system
  - Data from the host write request is simply appended at the end of the log

- How to replenish free pages?
  - The number of valid pages in the log < threshold
    - Choosing the smallest number of valid pages and erasing a block
  - The number of valid pages in the log > threshold
    - Choosing the largest number of valid pages and merging blocks
Chameleon Hybrid SSD Architecture

- Processing of Host Requests and Miscellaneous Issues
  - Host write request processing
    - Criterion
      - If the request start at a local block boundary and the number of requested pages is above a given threshold
  - Host read request processing
  - Bad block handling
  - Wear-leveling
    - Implicitly and explicitly
C. Processing of Host Requests and Miscellaneous Issues

Wear-leveling:

- In the implicit technique, when the SSD is idle, the SSD selects the free write buffer block with the smallest erase count and uses that block to service write requests. This helps in spreading the wear across the SSD. If the number of valid pages is above a given threshold, the request is directed to the block-level write buffer; otherwise, it is directed to the page-level write buffer. This approach is implemented in Chameleon since it also has data blocks and block-level write buffers.

- In the explicit technique, when the SSD is idle, the SSD selects the free write buffer block with the smallest erase count and uses that block to service write requests. This helps in spreading the wear across the SSD. If the number of valid pages is above a given threshold, the request is directed to the block-level write buffer; otherwise, it is directed to the page-level write buffer. This approach is implemented in Chameleon since it also has data blocks and block-level write buffers. Chameleon uses both implicit and explicit approaches for selecting the write buffer block to be used for servicing the write requests.

- The request belongs is already in one of the two types of write buffer if the difference in the erase count between the two is above a given threshold. In practice, these two techniques effectively prevent any single physical block from failing prematurely due to excessive erasures.

- On the other hand, in the explicit technique, when the SSD is idle, the free write buffer block with the smallest erase count is used. On host write request is directed to the block-level write buffer, the wear-leveling techniques to guarantee an even wear-out of the initial blocks and the page-level write buffer block with the smallest number of valid pages is selected and all the valid pages are copied to the block-level write buffer block with the largest erase count among those blocks that have been garbage-collected. After the copy operation, the write buffer block that has been garbage-collected is erased and added to the free buffer. After the block merge operation, all the pages in the log belonging to the logical block are considered as invalid. The valid pages in the log is selected and a block merge operation is performed using a spare physical block reserved at the SSD initialized. Similarly, a bad physical block at run time is handled.

- If the number of valid pages is above a given threshold, the logical block with the largest number of valid pages in the log. If the number of valid pages in the log is selected and all the valid pages are copied to the block-level write buffer block with the largest erase count among those blocks that have been garbage-collected. After the copy operation, the write buffer block that has been garbage-collected is erased and added to the free buffer. After the block merge operation, all the pages in the log belonging to the logical block are considered as invalid. The valid pages in the log is selected and a block merge operation is performed using a spare physical block reserved at the SSD initialized. Similarly, a bad physical block at run time is handled.

- In Chameleon, bad physical blocks at run time are handled. In the garbage collection, the data block associated with the logical block cannot be relocated from one type of write buffer to the other. The handling of a read demand consists of (i) searching the on-chip buffer for the desired logical block; (ii) if the logical block is found and is mapped out when the SSD is initialized, it is directed to one of the two types of write buffer based on the following criterion – if the request starts at a logical block boundary and the number of requested pages is above a given threshold, the request is directed to the block-level write buffer; otherwise, it is directed to the page-level write buffer. This straightforward manner. If the logical block to which the request belongs is already in one of the two types of write buffer explained in the previous subsection, a write request from the host is processed in a straightforward manner. If the logical block to which the request belongs is already in one of the two types of write buffer explained in the previous subsection, a write request from the host is processed in a straightforward manner. If the logical block to which the request belongs is already in one of the two types of write buffer explained in the previous subsection, a write request from the host is processed in a straightforward manner.
Prototype Implementation and Performance Evaluation

- Performance Evaluation Method
  - PCMark04 Benchmark program
    - Window XP Startup
    - Application Loading
    - General HDD Usage
    - File Copying
Evaluation Results

**PERFORMANCE EFFECT OF FRAM**

<table>
<thead>
<tr>
<th></th>
<th>Chameleon w/o FRAM</th>
<th>Chameleon with FRAM</th>
<th>% gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCMark04 HDD Score</td>
<td>10585</td>
<td>12841</td>
<td>21.3%</td>
</tr>
<tr>
<td>Windows XP Startup</td>
<td>21.2 MB/s</td>
<td>25.0 MB/s</td>
<td>17.9%</td>
</tr>
<tr>
<td>Application Loading</td>
<td>18.3 MB/s</td>
<td>22.9 MB/s</td>
<td>25.1%</td>
</tr>
<tr>
<td>General HDD Usage</td>
<td>14.7 MB/s</td>
<td>18.0 MB/s</td>
<td>22.4%</td>
</tr>
<tr>
<td>File Copying</td>
<td>30.4 MB/s</td>
<td>33.9 MB/s</td>
<td>11.5%</td>
</tr>
</tbody>
</table>
Prototype Implementation and Performance Evaluation

Evaluation Results

<table>
<thead>
<tr>
<th></th>
<th>block-level write buffer (X), page-level write buffer (X)</th>
<th>block-level write buffer (C), page-level write buffer (X)</th>
<th>block-level write buffer (X), page-level write buffer (C)</th>
<th>block-level write buffer (C), page-level write buffer (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows XP Startup</td>
<td>24.4</td>
<td>22.3</td>
<td>16.8</td>
<td>28.9</td>
</tr>
<tr>
<td>Application Loading</td>
<td>25.0</td>
<td>22.9</td>
<td>18.0</td>
<td>31.9</td>
</tr>
<tr>
<td>General Usage</td>
<td>7.5</td>
<td>5.3</td>
<td>4.4</td>
<td>14.5</td>
</tr>
<tr>
<td>File Copying</td>
<td>18.3</td>
<td>5.6</td>
<td>5.1</td>
<td>33.9</td>
</tr>
</tbody>
</table>

PCMark04 HDD Score

- **Windows XP Startup**: 3431
- **Application Loading**: 4165
- **General Usage**: 12287
- **File Copying**: 12841
Conclusions

- Chameleon
  - FRAM+SSD (NAND Flash array)
  - Although the size of FRAM is not enough for maintaining bulk data, the use of write buffering at the page-level is important
  - Practical implementation of Chameleon