|  |  |
| --- | --- |
| 产品名称Product name | 密级Confidentiality level |
| Flint | 秘密 secret |
| 产品版本Product version | Total ?? pages 共??页 |
| V001R001 |

Flint: distributed execution framework

|  |  |  |  |
| --- | --- | --- | --- |
| Prepared by  拟制 | ??? 项目组  Parallel Exection Group | Date  日期 | 2015-10-26 |
| Reviewed by  评审人 |  | Date  日期 | 2015-11-20 |
| Approved by  批准 |  | Date  日期 | 2015-1-20 |

Revision Record

| Date  日期 | Revision Version  修订 版本 | CR ID / Defect ID CR号 | Sec No.  修改 章节 | Change Description  修改描述 | Author  作者 |
| --- | --- | --- | --- | --- | --- |
| 2015/10/26 | V1.0 |  | ALL | Write | Knizhnik.konstantin |

Catalog

[Flint: distributed execution framework 1](#_Toc416276527)

[Revision Record 2](#_Toc416276528)

[Table List 5](#_Toc416276529)

[**未找到目录项。**Figure List 5](#_Toc416276530)

[1 Introduction 9](#_Toc416276531)

[1.1 Purpose 9](#_Toc416276532)

[1.2 Scope 9](#_Toc416276533)

[1.2.1 Name 9](#_Toc416276534)

[1.2.2 Operators 9](#_Toc416276535)

[1.2.3 Applications 9](#_Toc416276536)

[1.2.4 Requirement description 9](#_Toc416276537)

[1) IT production Line requirement 9](#_Toc416276538)

[2) System requirement 9](#_Toc416276539)

[2 Flint Architecture 12](#_Toc416276540)

[3 Level 0 Design Description 13](#_Toc416276542)

[3.1 Software System Context Definition 13](#_Toc416276543)

[3.2 Design Considerations 13](#_Toc416276544)

[3.2.1 Design Alternatives 14](#_Toc416276545)

[3.2.2 Other Design Considerations 19](#_Toc416276546)

[3.2.3 Major work flow 20](#_Toc416276547)

[4 Level 1 Design Description 22](#_Toc416276548)

[4.1 System Architecture 22](#_Toc416276549)

[4.1.1 Description of the Architecture 23](#_Toc416276550)

[4.1.2 Representation of the BusinessFlow 24](#_Toc416276551)

[4.2 Decomposition Description 27](#_Toc416276552)

[4.2.1 Initail sub-system 27](#_Toc416276553)

[1) Overview 27](#_Toc416276554)

[2) Functions 28](#_Toc416276555)

[4.2.2 KV request sub-system 28](#_Toc416276556)

[1) Overview 28](#_Toc416276557)

[2) Functions 30](#_Toc416276558)

[4.2.3 Hot data sub-system 30](#_Toc416276559)

[1) Overview 30](#_Toc416276560)

[4.2.4 Read Cache sub-system 33](#_Toc416276561)

[1) Overview 33](#_Toc416276562)

[2) Functions 35](#_Toc416276563)

[4.2.5 Cold data metadata management sub-system 35](#_Toc416276564)

[1) Overview 35](#_Toc416276565)

[2) Functions 36](#_Toc416276566)

[4.2.6 Flash physical space management (FTL) sub-system 36](#_Toc416276567)

[1) Overview 36](#_Toc416276568)

[2) Functions 38](#_Toc416276569)

[4.2.7 Flash Concurrent IO management sub-system 38](#_Toc416276570)

[1) Overview 38](#_Toc416276571)

[2) Functions 39](#_Toc416276572)

[4.2.8 Flash physical space management (NVMe) sub-system 39](#_Toc416276573)

[4.2.9 Network 44](#_Toc416276574)

[3) Overview 44](#_Toc416276575)

[4) Functions 44](#_Toc416276576)

[4.3 Dependency Description 44](#_Toc416276577)

[4.4 Interface Description 45](#_Toc416276578)

[4.4.1 KV request sub-system and Flash Concurrent IO management sub-system 45](#_Toc416276579)

[4.4.2 Hot data sub-system and Flash Concurrent IO management sub-system 47](#_Toc416276580)

[4.4.3 In Flash Concurrent IO management sub-system 50](#_Toc416276581)

[5 Level 2 Design Description 51](#_Toc416276582)

[5.1 Initail sub-system 51](#_Toc416276583)

[5.1.1 Sub-system Design 51](#_Toc416276584)

[5.1.2 Module Design 51](#_Toc416276585)

[5.1.3 Work flow in the sub-system 52](#_Toc416276586)

[5.1.4 Interface in the sub-system 52](#_Toc416276587)

[5.2 KV request sub-system 53](#_Toc416276588)

[5.2.1 Sub-system Design 53](#_Toc416276589)

[5.2.2 Module Design 55](#_Toc416276590)

[5.2.3 Work flow in the sub-system 57](#_Toc416276591)

[5.2.4 Interface in the sub-system 66](#_Toc416276592)

[5.3 Hot data sub-system 67](#_Toc416276593)

[5.3.1 Sub-system Design 67](#_Toc416276594)

[5.3.2 Module Design 68](#_Toc416276595)

[5.3.3 Work flow in the sub-system 71](#_Toc416276596)

[5.3.4 Interface in the sub-system 74](#_Toc416276597)

[5.4 Read Cache sub-system 74](#_Toc416276598)

[5.4.1 Sub-system Design 74](#_Toc416276599)

[5.4.2 Module Design 75](#_Toc416276600)

[5.4.1 Work flow in the sub-system 76](#_Toc416276601)

[5.4.2 Interface in the sub-system 76](#_Toc416276602)

[5.5 Cold data metadata management sub-system 76](#_Toc416276603)

[5.5.1 Sub-system Design 76](#_Toc416276604)

[5.5.2 Module Design 76](#_Toc416276605)

[5.6 Flash physical space management (FTL) sub-system 78](#_Toc416276606)

[5.6.1 Sub-system Design 79](#_Toc416276607)

[5.6.2 Module Design 79](#_Toc416276608)

[5.6.3 Work flow in the sub-system 81](#_Toc416276609)

[5.6.4 Interface in the sub-system 84](#_Toc416276610)

[5.7 Flash Concurrent IO management sub-system 85](#_Toc416276611)

[5.7.1 Sub-system Design 85](#_Toc416276612)

[5.7.2 Module Design 85](#_Toc416276613)

[5.7.3 Work flow in the sub-system 86](#_Toc416276614)

[5.7.4 Interface in the sub-system 86](#_Toc416276615)

[5.8 Network 86](#_Toc416276616)

Table List

**未找到目录项。**Figure List

Flint: distributed execution framework

High Level Design Document

Keywords： RDD, distributed, HDFS, Hadoop, Spark, SQL

Abstract :  This document describes the high level design of the Flint which is considered as faster elternative to Hadoop/Spark. Flint allows to write queries in C++, completely eliminating JVM interpretation/JIT cost, garbage collection and serialization overhead. Flint is optimizaed for processing data in memory, avoiding materialization of intermediate results to the disk.

Flint’s program is written in C++ and the same code is executed at each node. There is no special coordinator node in Flint. Instead of it first node is playing role of coordinator of queries. It performs merge of partial results produced by other nodes and delivers final result to the user.

Communication between nodes is performed using Unix domain/TPC sockets using propertary protocol. Each node has one thread receiving data from network and multiple threads sending data to other nodes (one sender thread for each node). The unit of transfer is large enough block (64kb by default). Messages are packed in the block and placed in queues. Queue is high level mechanism of exchanging data between nodes. Producer fills the block and places it in destination queue (for example in sender’s queue). Sender retrives blockfrom queue and send it through netwrork to the resipient. Recevier thread reads data from network and place in distination queue. Queues are identified by ID, which is assign at the moment of queue creation. As far as all nodes are executing the same code, each node is assigned ther same set of queue’s IDs.

Flints implements classical map-reduce operators (filter, project, map-reduce, reduce, aggregate,…) as well as specialized operators used by SQL (join, semijoin, sort, …). Pipelined operators (like filter and project) are executed inside pipeline locally at each node. Other operators require exchanging of data between nodes. Then can shuffle, broadcast, scattern and gather their data

List of abbreviations. Describe abbreviations in this document, full spelling of the abbreviation and Chinese explanation should be provided.

| Abbreviations | Full spelling | Explanation |
| --- | --- | --- |
| RDD | Resilient Distributed Dataset | The basic abstraction in Spark. Represents an immutable, partitioned collection of elements that can be operated on in parallel.  . |
| SQL | SQL | Non-procedural query language. De-facto standard for modern database systems. |
| RDBMS | Relational Database Management System | System responsible for storing and processing of relations (tables). |
| HDFS | Hadoop distributed file system | Distributed files system used in Apache Hadoop. |

# Introduction

## Purpose

This document describes the high level design of the Flint distributed execution framework, which can be used standalone or as part of Spark, accelerating query execution.

### Name

Flint : Distributed execution framework

### Operators

Flint implements operators needed for execution of SQL queries and performing other distributed calculating. Also it provides framework for wrioting own operators and transports.

### Applications

Flint is intended to be used for execution of analytic queries for large volumes of data (big data). It can be considered either as competitor of Hadoop/SparkSQL, either extension/accelator of SparkSQL.

### Requirement description

#### IT production Line requirement

#### System requirement

* Must support

|  |  |  |  |
| --- | --- | --- | --- |
| **Service Requirment** | | | |
| **Requirement name** | **Challenge, problem and requirement description** | **Requirement category** | **Priority** |
| Support shared nothing configuration: data is stored locally at each node | Flint should allow to access files in local files system in internal raw format. Also it should provide utility for converting data fro CSV format to the internal repreosentation. | 01.function requirement | High |
| Support work with distributed file system | Flint must support work with HDFS | 01.function requirement | High |
| Support parquet file format | Flint should be able to work with parquet files stored either in local file system, either in DFS | 01.function requirement | High |
| Support integration with SparkSQL | It should be possible to use Flint as SparkSQL RDD, making it possible to move to Flint part of query execution plan. | 01.function requirement | High |
| Eliminate disk IO overhead | Flint should be able to process quieries without saving intermediate results to some storage device. | 02.performance requiement | High |
| Support work with dataset larger than size of memory | Decpit to the fact, then in-memory option is considered as primary Flint’s use case, it should be able to work with data sets not fitting in memory by shuffling data to the files | 02.performance requiement | Low |

* Optional support

Support high availablability: add/remove node without stopping the system

* Ignore support

Fault tolerance inside a query. It is normal to abort current query if failure happens during query execution.

* Administration utilities

Flint is not a production system so it caurrently doesn’t provide any monitoring or administration utilities..

* Technique Specification Summary
* Function：Support operators needed for distributed execution of SQL queries
* Performance: Should provide 3x better performance than SparkSQL.
* Integration: Should be available as SparkSQL RDD.
* Scalability: The performance increases with the amount of nodes.

# Flint architecture

illustrates processing of typeical SQL query. Flint builds pipeline of SQL operators which are executed concurrently at all nodes. Pipelined operators (like Project, Join) are executed independently at al nodes. Most of operations are inlines (thanks to using templates in Flint’s API). Result of each operator (row) is passed to the input of other (outer) operator. Right now Flint implements PUSH style processing (as in Volcano-style execution engine).

Fig. 1 SQL query processing pipeline

More complex SQL operators like join, group-by, aggregate, sort,… requires interaction between different nodes. Flints supports three main communication patterns:

1. Scatter-gather:

Data source is partitioned by some key and sent to the target nodes. And data nodes collect data from different nodes and process them. Scatter-gather is used to implement hash join when both joined tables are large enought. Another example of using scatter-gather is classical map-reduce which is used in SQL queries to perform aggregation with grouping.

1. Broadcast:

Data source is replicated to all nodes. Replication can be used to peform hash join when one joined table is much smaller than other.

1. Merge:

Each node performs local processing of the data and then sends partial results to coordinator where them are combined into final result. Merge is used to sort result set and to calculate grand aggregates.

shows how how scattering of data is implemented in Flint. There is separate scatter job which fetches input data source, extracts parititiong key, calculates hash code, divides it by modulo of number of nodes and and places record in the correspondent send queue. Each send query is connected with send job. So number of send queues and send threads is equals to number of nodes in the system minus one (no need to send data to self node).

And eachnode has single receive thread which collects data from all nodes. Recevied message includes ID of queue where data should be placed. As far as all nodes are executing the same query execution plan, sequences of obtained queue IDs are the same at all nodes. So we can specify target queue ID without any additional communication with this node.

Recevier thread enqueues received block in speficied queue connected with gather job. This job implements reduce logic – it combines responces received fro multiple node and produce single output stream of data.

# Level 0 Design Description

## Flint RDD API

shows the high-level store system architecture in NISS group. The pink block shows the location of key-value store sub-system in the whole store system in NISS group. The key-value store sub-system (KV system in brief below) provides SET, GET and DEL API to application level and by reading/writing/deleting data from/to DRAM, SCM and Flash to build the KV system. This KV system doesn’t support the RANGE(key) operation.

Our KV system is also responsible to manage the space allocation of DRAM and SCM in order to obain NUMA-aware memory. Our KV system uses direct I/O approach to read/write/erase Flash, in other word, the physical address of Flash which stores kv item is recorded in the offset part of the cold hash table item. The physical space management, such as Flash garbage collection (Flash GC in brief below) and Flash wear-out balance, is integrated in our KV system. In other word, in our system, there isn’t Flash Translation Layer (FTL) and the Flash is used as Memory Technology Device (MTD).

Fig. 2 The high-level store system architecture in NISS

## Design Considerations

Our KV system exploits fully the special advantages of DRAM, SCM and NAND Flash and provides highest performance. The major design idea of the KV system as follow:

1. High perforamce

Out KVsystem contains DRAM, SCM and Flash. DRAM is used as hot data hash table, read cache and temporary read/write buffer for reading and writing Flash. SCM is used as storage of hot data and the cold data hash table. Some auxiliary Flash management information is also stored in SCM. The cold data which evict from the hot data is stored in Flash. The Flash is used as MTD. In order to increase the write performance of the Flash, the write size of Flash is block.

Based on the optimization of data partition, NUMA-aware memory allocation and TSX technology, the concurrent performance is improved and then single-node obtains high throughput.

The optimization for CPU cache also must be considered. Such as the bucket of hash table must meet the cache line size and prefetching data to cache line to combine the pipe line.

1. High concurrent

In order to obtain high performance, the high concurrent is very important in our KV system. The partition is the mjor idea. Each kv thread, which is deal with kv request, has NUMA-aware DRAM and SCM and independent KV data structure, which includes Read Cache, Hot data and Cold data Hash table. The TSX and/or elaborate desiged Lock is used.

1. Write consistent

The atomic write and/or TSX are used to provide write consistent. The write consistent constains concurrent consistent and failure consistent.

1. High performance Cache algorithm

Our system uses the Read Cache to reduce the read of Flash. In order to obtain high performance, the eviction algortithm of Read Cache must be designed elaborately.

1. High performance Hash table

There is Read Cacha Hash table, Hot data Hash table and Cold data Hash table in our KV system, those Hash tables decide the our KV system’s performance. Those Hash table must be designed elaborately.

The major challenge in our KV system is as follow:

1. High performance Hash table
2. Complex Lock

The lock is used to guarantee the conrrent consistent, at the same time, it must affect the performance slightly.

1. The asychronized Flash read/write model.

Client-Server model is used in Flash operation.

### Design Alternatives

shows the high level major flow of out KV system, The Clients connect to the server by NIC, which can routing the requests to the appropriate core directly.

Fig. High level major flow of KV system

shows the data flow in hybrid storage medium. The data which is written lately (Hot data below) is stored in the SCM firstly. When the SCM if full, the oldest data (Cold data below) is evicted from the SCM to Flash. When GET(key) operation is applied and the kv is stored in Flash, the kv will be cache in Read Cache, which is in DRAM.

Fig. The major of the KV system

1. Data Layout

Fig. 5 The layout of our KV system

shows data layout in the hybrid storage medium, which contains DRAM, SCM and Flash. DRAM is used as hot data hash table, read cache and temporary read/write buffer for reading and writing Flash. SCM is used as storage of hot data and the cold data hash table. Some auxiliary Flash management information is also stored in SCM. The cold data is stored in Flash. From function module viewpoint, the data layout can be divided into four modules: Read cache module, Hot data module, Cold data module and Flash Block manage module. More detail of those four modules’ data layout is described below. Each thread used to deal with the kv request has a independent whole Data structure mentioned above, in order to improve the concurrent performance. We refer that as partition in our KV system, which is very important technology in our KV system.

1. Partition

There are three Concurrent models: Excusive Read / Exclusive Write (EREW), Concurrent Read / Exclusive Write (CREW) and Concurrent Read / Concurrent Write (CRCW). The EREW is the simple model and it does not need Lock to guarantee the concurrent consistent. The disadvantage is the load unbalance. The CREW relieves the load unbalance problem but brings the concurrent inconsistent problem, so Read/Write Lock or Intel’s TSX will be used to guarantee the concurrent consistent. The CRCW relieve the load unbalance problem further, while it brings more challenge of concurrent consistent. The read/write model in this chapter only relates to DRAM and SCM, which is directly to the external operation, such as SET, GET and DEL. The model of read/write Flash will be described in special chapter below.

* Excusive Read / Exclusive Write

Fig. The architecture of EREW

Fig. 6 shows the architecture of EREW. Each core corresponds to independent thread and has independent DRAM and SCM, which is NUMA-aware. Based on the hash value of the key, the operation, such as SET, GET and DEL, is send to corresponding thread or core. Each thread has independent whole KV data structure, such as Hot data circular log, Hot data hash table, Cold data hash table, Cold data, Read Cache, Flash block manage auxiliary data structure and so on. Each thread works independently with other threads. Because the whole KV data structure is independent, there are not more than one operation to the data structure and it does not need Lock to guarantee the concurrent consistent.

* Concurrent Read / Exclusive Write

(a)

(b)

Fig. The architecture of CREW. (a) Concurrent Read. (b) Exclusive Write

Fig. 7 (a) shows the architecture of Concurrent Read in CREW model. In this model, the read operation of the same kv item may be applied in different core, so the concurrent consistent will be guaranteed by RW Lock. There is only one Read Cache for one CPU, which can increase the efficiency of Read Cache and relieve the load unbalance problem. (b) shows the architecture of Exclusive Write in CREW model, which is the same as the EREW model.

* Concurrent Read / Concurrent Write

Fig. The architecture of CRCW

shows the architecture of CRCW. There is only one KV data structure for one CPU.

1. NAND Flash Concurrent

。

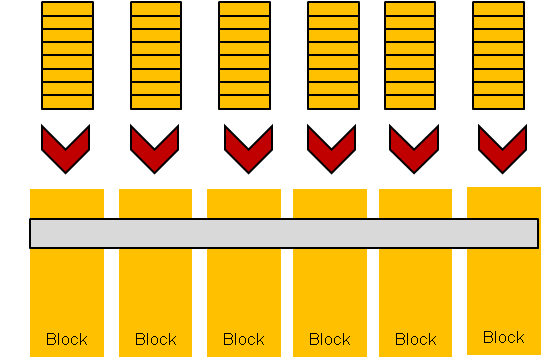
****

Fig. The architecture of CRCW

### Other Design Considerations

There are two usage models of our KV system: local model and remote model. For the local model, the application which uses the KV system and the KV system are run on the same computer. For the remote application model, the application which uses the KV system and the KV system are run on differment computers. In other word, the KV system is a server which provides kv operation server and the kv resqust is issued from the clients which runs the user’s application. The clients and the server are connected by network.

For the local model, the KV system provides kv function interface (SET, GET and DEL) and those functions run in the user appliaction’s process. When the SET, GET and DEL operation doesn’t need to read Flash, those functions can finise the SET, GET and DEL operation indepenently. The Evict, GC and read/write Flash runs in the KV background process. The Hot data, Read cache and the Cold data Hash table in the shared-memory area and are shared by the user appliaction’s process and the KV background process. The data shared and lock must be designed to have the ability to be shared between defferent processes.

For the remote model, the KV system provides kv function interface (SET, GET and DEL) and those functions run in the user appliaction’s process. Those functions only provides function that sending the request to the KV system and receving the responds. Those functions can not finised SET, GET and DEL function independently, even if doesn’t need to read Flash, which is different from those function in local model. The whole kv operation (SET, GET, DEL, Evict, GC and read/write Flash) is in a process with multi-threads and runs in the server.

### Major work flow



Fig. 10 Major work flow

# Level 1 Design Description

## System Architecture

(a)

(b)

Fig. 11 KV system architecture

shows the system architecture of our KV system. Our KV system can be divided into 7 sub-system: Initail sub-system, KV request queue sub-system, Hot data sub-system, Read Cache sub-system, Cold data metadata management sub-system, Flash physical space management (FTL) sub-system, Flash Concurrent IO management sub-system.

### Description of the Architecture

|  |  |  |
| --- | --- | --- |
| Sub-system | Module Nume( sub-module in the sub-system) | Function description |
| Initail sub-system | Pre-malloc | Malloc the physical memory of DRAM and SCM for KV system |
| Data structure build | Build the data structure used to manage and store the kv item |
| Failture recovery | Rebuild the data structure when reboot from failture. It is used to guarantee the failure consistent. |
| KV request sub-system | Request queue manage | Manage the kv request queue. |
| kv operation | Provide SET, GET and DEL operation |
| Concurrent | Provide high concurrent performance and concurrent consistent |
| Hot data sub-system | Lossless Hash table |  |
| Cirular log |  |
| Data evict |  |
| Read Cache sub-system | Lossy Hash table |  |
| Data List |  |
| Data evict |  |
| Cold data metadata management sub-system | Cuckoo Hash table |  |
| Valid bitmap |  |
| Flash physical space management (FTL) sub-system | GC |  |
| Flash Concurrent IO management sub-system | Read |  |
| Write |  |
| Network | Initial network |  |
| Get/return request |  |
| Request distribute | Send the kv request to the conrresponding core |

### Representation of the BusinessFlow

The work flow includes external function such as SET, GET and DEL, the internal function such as read Flash, write Flash, Garbage collection and Hash table operation, initial and network. The work flow must guarantee the concurrent consistent and the failure recovery consistent. For CREW and CRCW model, we will use the Intel’ TSX as an optimization version. We also design a version with lock in order to apply out KV system in the servers which cannot support Intel’ TSX. Software Transactional Memory is another approach to guarantee the concurrent consistent without Intel’ TSX. MICA uses an approach that combines the STM and the Lock. Deal to the read process, it uses the STM. Deal to the write process, it uses the Lock. It is an optional approach in our KV system without Intel’ TSX.

* **SET**

Fig. Primary Work Flow of SET

The primary work flow of SET is shown in . There are four steps for SET an item.

Step1. Write the kv item at the tail in Hot data Circular log, move the tail of the Hot data, and Return Write Success to the request port. If the failure occurs when just Return Write Success, there are two kv items whose keys are same and both those items are valid. Under the condition that there are more than one kv items whose key are same and both are valid, the last one is affirmed as valid kv item. We also check whether the last kv item of the Hot data exists in the Cold data. It the last kv item exists in the Cold data, we will mask the kv item in the Cold data invalid. The failure recovery consistent is not guaranteed under the condition that the failure happen after move the tail of the Hot data before Return Write Success to the request port. However, the consistent of concurrent operation is not guaranteed. A GET request of that kv item will bring inconsistent, because the index in the Hot Data Hash table, Cold Data Hash table or Read Cache points to the old kv item or not index in the Hot Data Hash table. So a RW Lock will be used to guarantee the concurrent consistent. The detail will be described in Lock version section below.

Step2. We check whether a kv item (old kv item) whose key is the same to that of the new kv item exists in the Hot Data. If there is an old kv item in the Hot Data, we mask the old kv item invalid, change the offset of the old Hot Data Hash table item to point the new kv item, and the SET request is completed. If there is not an old kv item, we build the corresponding hash item in the Hot data Hash table and then go to step3. The process of step 2 guarantees the failure recovery consistent, but the concurrent consistent also cannot be guaranteed. The analysis is the same as the analysis mentioned above.

Step3. We check whether an old kv item exists in the Cold Data. If there is an old kv item in the Cold Data, we mask the old kv item invalid and go to step4. If there is not an old kv item in the Cold Data, the SET request is completed. The process of step 3 guarantees the failure recovery consistent, but the concurrent consistent also cannot be guaranteed. The analysis is the same as the analysis mentioned above.

Step4. We check whether an old kv item is in the Read Cache. If there is an old kv item in the Read Cache, we mask the old kv item invalid. The process of step 4 guarantees the failure recovery consistent, but the concurrent consistent also cannot be guaranteed. The analysis is the same as the analysis mentioned above.

* **GET**

Fig. Primary work flow of GET

The primary work flow of GET is shown in . There are three steps for GET an item. The primary work flow guarantees the failure recovery consistent, but the concurrent consistent also cannot be guaranteed. The analysis is the same as the analysis mentioned above.

Step1. We check whether the needed kv item is in the Read Cache by Read Cache Hash table. If the needed kv item is in the Read Cache, we will return the kv item and the process is completed. If the needed kv item doesn’t existed in Read Cache, then go to step2.

Step2. We check whether the needed kv item is in the Hot data by Hot data Hash table. If the needed kv item is in the Hot data, we will return the kv item and the process is completed. If the needed kv item doesn’t existed in Hot data, then go to step3.

Step3. We check whether the needed kv item is in the Cold data by Cold data Hash table. If the needed kv item is in the Cold data, we will read the corresponding Flash Page to the Temporary R/W Buffer, find the needed data, return the kv item and the process is completed. If the needed kv item doesn’t existed in Cold data, return no valid item alarm and the process is completed.

* **DEL**

Fig. Primary work flow of DEL

The primary work flow of DEL is shown in . There are three steps for DEL an item. The primary work flow guarantees the failure recovery consistent, but the concurrent consistent also cannot be guaranteed. The analysis is the same as the analysis mentioned above.

Step1. We check whether the kv item exists in Hot Data. If the kv item exists in the Hot data, we will delete the corresponding Hot data Hash item, mask the kv items invalid in the Hot data Circular log, Return SUCC and the process is completed. If the kv item doesn’t existed in Hot data, then go to step2.

Step2. We check whether the kv item exists in Cold Data. If the kv item exists in the Cold data, we will delete the corresponding Cold data Hash item, mask the corresponding item valid bitmap invalid. Maybe the kv item occupies more than one Flash item, all the corresponding item valid bitmap must be mask invalid. Then go to step3. If the kv item does not exist in the Cold data, Return No Data and the process is completed.

Step3. We check whether the kv item exists in Read Cache. If the kv item exists in the Read Cache, we will delete the corresponding Read Cache Hash item, Return SUCC and the process is completed.

* **Failure Recovery Work Flow**

When it reboots from failure recovery, the process of the rebuild the Hot Data Hash as following: scan the Hot data Circular log and insert the corresponding hash item to the Hot Data Hash table.

We will continue the process of Evict and GC process which breaks down by the failure.

* **Network Function Work Flow**

The Network function work flow is the same as that in MICA.

## Decomposition Description

### Initail sub-system

#### Overview

Fig. Data structure for managing DRAM and SCM

shows the Data structure for managing DRAM and SCM, which is similar to the data structure used to manage the DRAM in MICA. The slight different is that we add the store\_type in the mehcached\_shm\_pages data structure to marking the memory page is DRAM or SCM. The Data structure for managing DRAM and SCM is stored in DRAM.

#### Functions

|  |  |
| --- | --- |
| Function number | Function Description |
| 1 | Malloc the physical memory of DRAM and SCM for KV system |
| 2 | Build the data structure used to manage and store the kv item |
| 3 | Rebuild the data structure when reboot from failture. It is used to guarantee the failure consistent. |

### 

### KV request sub-system

#### Overview

Fig. Framework of the kv request sub-system (local model)

For the local model, the user’s application and the KV system run on the same computer but different processes. Fig. 16 shows the Framework of the kv request sub-system for local model. The light red block represents the user’s application process and the light blue block represents the KV system’s background process. We provide function interface (SET, GET and DEL) to user’s application by library and those functions run in the user’s application process. When the SET, GET and DEL function need a Flash read, those functiosn add a read Flash request in the read Flash request queue and suspend the user’s application process or the corresponding threads in the user’s application process. The Flash Concurrent IO management sub-system finds the read Flash request and prcess that resquest. When the Flash Concurrent IO management sub-system finish the read Flash request, it add the respond in the read Flash finish queue and awake the corresponding user’s application process or the corresponding threads in the user’s application process.

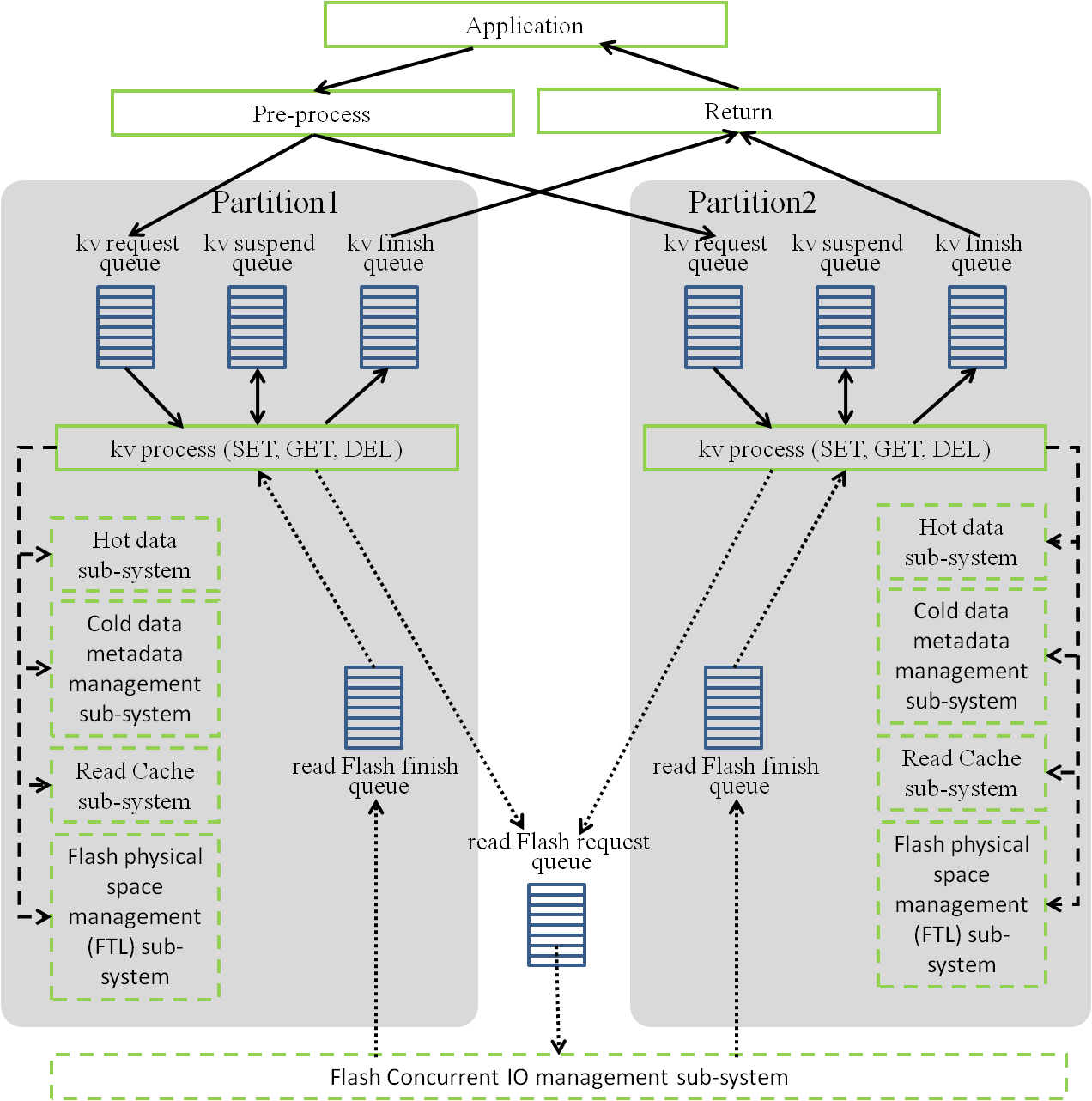
****

Fig. Framework of the kv request sub-system (remote model)

In order to reduce the workload unbalance with lock elision, we add a kv request queue to each kv thread. The kv request operation first put at the tail in the kv request queue and the corresponding thread get the kv request from the header of the request queue. The process of kv request maybe have interaction with the Hot data sub-system, Read Cache sub-system, Cold metadata management sub-system and Flash physical space management sub-system. Those interface functions are synthronous. The process of kv request maybe have interaction with the Flash Concurrent IO manangement sub-system, whose interface functions are asynthronous, so some queue must be used.

Because the read Flash process is asynthronous, a kv request will be suspended when that kv process of that kv request need read Flash process and other kv request will be processed. In order to achieve that requirement, the kv suspend queue and kv finish queue must be needed. The kv suspend queue stores the state of the suspended kv request and the kv finish queue store the finished kv request.

When a kv request from the application achieves the KV system, the pre-process distributes the kv request into different kv thread (Partition) and when Return thread polls the kv finished queue and return the result to the appliaction when the Return thread finds a finished kv request.

Because the read Flash process is asynthronous, when a kv need a read Flash process, that kv thread add the read Flash request at the tail in the read Flash request queue. When the kv thread finish a kv request or suspend a kv request, that kv thread will check whecher there is a new item in it own read Flash finish queue. If there is a new item, the kv thread obtain that item and continue to process the corresponding suspended kv request. If there is no new item, the kv thread will check whecher there is a new item in it own kv request queue.

#### Functions

|  |  |
| --- | --- |
| Function number | Function Description |
| 1 | Manage the kv request queue. |
| 2 | Provide SET, GET and DEL operation |
| 3 | Provide high concurrent performance and concurrent consistent |

### Hot data sub-system

#### Overview

Fig. The data layout of Hot data module

In order to improve high performance of SET, we use Hot data module to store the lately written kv item. shows the data layout of Hot data module. The Hot data module has two part: Hot data hash table and Hot data, which are stored in different medium according to those two part’s requirement and storage medium character. The Hot data hash table is stored in DRAM, because it will be read and write frequency and it can be reconstructed by scanning the Hot data when reboots from failure recovery. The Hot data is stored in SCM, because SCM is non-volatile storage medium. Comparing with DRAM, the data in SCM will not be lost when the KV system failure. Another reason of using the SCM store the hot data is that, the write speed is faster than Flash, which is another non-volatile storage medium. The SCM write granularity is byte, which is suitable to respond the SET request. While the write granularity of Flash is page, which is much larger than the size of kv item and not suitable to be written directly when responding SET request.

Comparing wtih B-tree or another tree structure, hash table can provide higher performance of insert and search, so we use hash table to index the hot data. We want use the Lossless concurrent hash table and City Hash function. The item in the Hot data hash table also has two parts: Tag and offset, which is similar to the hash table item if Read cache hash table. The Item offset in the Hot data hash item represent the virtual address where the corresponding kv item is stored. The last item in each bucket is a pointer which index to another bucket, which is different to the bucket in Read cache hash table. The hot data is organized used Circular log structure.

The kv item is organized using Circular log structure. The Circular log structure has two pointers: Head and Tail. The old items are evicted from the Head and the new items are inserted from the Tail. The kv item contains three parts: auxiliary information, key and value. The auxiliary information includes Initial size, Keyhash, Key/value length, Expire time and Valid mask. The Initial size is the total size of the kv item including auxiliary information, key and value. It is used on condition of in-place update. That Initial size item is invalid in our KV system of version 1.0. Because we use the Circular log to achieve the LRU evict approach. In other word, the least recently read item will be evicted when the Circular log is full. Because the Circular log evicts the item from the Head pointer, in order to guarantee the LRU, we add new item and update item from Tail pointer. If we used in-place update approach, the update item near the Head pointer will be evicted soon, which cannot satisfy the LRU. The Keyhash is the hash value of the key. In order to reduce most unnecessary key comparison, which computation is larger than the comparison of Keyhash, our KV system compares the Keyhash firstly. Only if the Keyhash matches, the full key is compared to verify whether the kv item is the requestd kv item. Because the length of the key and value are unfixed, so the length of key and value are recorded in the Key/value length item. The Expire time is used to record the operation time of the kv read request, which is used to statistics the performance of our KV system. The Valid mask is used to mask that kv item is valid or invalid. Because there is not special garbage collection operation for the invalid item when the item in the Read cache is update, the garbage collection of the invalid item is delayed and will be do with the eviction process when the Head of the Circular log points to the invalid item. That valid mask is used to mask that item is valid or invalid. That valid mask is used when the eviction process. If the valid mask shows that item is valid, the delete hash item corresponding to the kv item will be done. If the valid mask shows that item is invalid, the delete hash item process will not be applied.

In order to improve the concurrent performance, each core has an independent Hot data module and stored in the NUMA-aware DRAM and SCM.

The Hot data sub-system has another function is that it provides evicting data from the Hot data to Cold data. The major work flow is described below.

Fig. Primary work flow of Evict

The primary work flow of Evict is shown in . There are three steps for Evict. The primary work flow guarantees the failure recovery consistent, but the concurrent consistent also cannot be guaranteed. The analysis is the same as the analysis mentioned above.

Step1. Read data from the Head position of the Hot Data Circular log to structure a Flash block and store the data in the DRAM’s Temporary R/W Buffer. Go to Step3.

Step2. Tell the Flash write interface function the logical address of the DRAM’s Temporary R/W Buffer and the physical address of the Flash block which will be write. Call the Flash write interface function to write Flash, wait write success return and go to step3. The failure maybe occurs after writing some part or writing the whole block. When reboot from the failure recovery, there are two copies of the same kv items. In order to guarantee the failure consistent, we will erase the block which may be written before the failure and delete the corresponding Hash item.

Step3. Build Cold data Hash Item, Delete Hot data Hash item, move Hot Data Circular log head pointer. The failure consistent is guaranteed by the approach mention above.

The Evict process need read/write Flash operation, bacuase the write Flash Functions are asynchronous, the queue must be used which is similar to the read Flash operation mention above.

Fig. Primary work flow of Evict

|  |  |
| --- | --- |
| Function number | Function Description |
| 1 | Storage Hot data |
| 2 | Provide SET, GET, DEL operation |
| 3 | Evict data from the Hot data to Cold data |

### Read Cache sub-system

#### Overview

Fig. The data layout of Read cache module

In order to provide high performance of GET, we use Read cache to store those kv items which are read from Flash lately. shows the data layout of the Read cache. The whole Read cache module is stored in DRAM

Comparing wtih B-tree or another tree structure, hash table can provide higher performance of insert and search, so we used hash table to manage the read cache kv items. The hash items are organized using Lossy concurrent hash table and the hash function is City Hash. There are two parts in each hash item: Tag and Item offset. The tag is a compressed version or the mathematical transform of key. It is used to reduce most unnecessary kv items reads. When doing lookup request, only if the tag matches, the full key in kv item is retrieved to verify whether the kv item is correct. Comparing with the entire key recorded in the hash item, the tag can reduce the storage space and be suitable to the application using unfixed length key. The item offset in the hash item represents the virtual address where the kv item is stored.

The free space for kv item is organized using list. Based on the space size, the free space is divided into different categories. The kv item contains three parts: auxiliary information, key and value. The auxiliary information includes Initial size, Keyhash, Key/value length, Expire time and Valid mask. The Initial size is the total size of the kv item including auxiliary information, key and value. It is used on condition of in-place update. That Initial size item is invalid in our KV system of version 1.0. The Keyhash is the hash value of the key. In order to reduce most unnecessary key comparison, which computation is larger than the comparison of Keyhash, our KV system compares the Keyhash firstly. Only if the Keyhash matches, the full key is compared to verify whether the kv item is the requestd kv item. Because the length of the key and value are unfixed, so the length of key and value are recorded in the Key/value length item. The Expire time is used to record the operation time of the kv read request, which is used to statistics the performance of our KV system. The evict algorithm is LRU, so the double link list will be used.

In order to improve the concurrent performance, each core has an independent Read cache module and stored in the NUMA-aware DRAM.

#### Functions

|  |  |
| --- | --- |
| Function number | Function Description |
| 1 | Storage Read Cache data |
| 2 | Provide SET, GET, DEL operation |
| 3 |  |

### Cold data metadata management sub-system

#### Overview

Fig. The data layout of Cold data module

The Cold data, which are evicted from the Hot data, is stored in Flash. Comparing with the hot data in SCM, the majority of the kv items are the cold data which is stored in Flash. Comparing with SCM, though the read/write performance of Flash is less, the price of Flash is lower and the capacity of Flash is larger, so Flash is suitable for storing the Cold data. The kv items in Flash are indexed by Cold hash which is stored in SCM. Because the Flash cannot be in-place update, so an item valid bitmap is stored in SCM. The item valid bitmap records whether the corresponding Flash logical minimum index granularity is valid or not. Because the Cold hash table and the item valid bitmap are read/write frequency and cannot be lost when failure recovery, SCM is suitable for storing them. Because the Flash read/write granularity is page and the size of kv item is much smaller than the size of Flash page, the Temporary R/W buffer is used to suit for Flash read/write operation. The Temporary R/W buffers are stored in DRAM, because those buffers are read/write frequency and the data in those buffer can be lost when failure recovery. shows the data layout of Cold data module.

Comparing wtih B-tree or another tree structure, hash table can provide higher performance of insert and search, so we use hash table to index the cold data. We want use the cuckoo hash table and City Hash function. The item in the Cold data hash table also has two parts: Tag and offset, which is similar to the hash table item if Read cache hash table. The Item offset in the Cold data hash item represents the physical address where the corresponding kv item is stored.

Because the Flash cannot be in-place update, so deleted or updated process products invalid item. The garbage collection of the invalid kv item is delayed and applied when the garbage collection of the block which contains the invalid kv items. In other word, there is not special garbage collection for the invalid kv item. In order to mask whether a kv item is valid, we used a item valid bitmap to record. The item valid bitmap is stored in SCM, one logical Flash minimum index granularity correspond to one bit valid mask. If the logical Flash minimum index granularity in Flash is invalid, the corresponding bit is set to false. Considering the bit number limitation of the item offset in hash table and the size of the item valid bitmap, the Flash minimum index granularity may be 32B or 64B or 128B. For example, if the total capacity of Flash is 2TB and the Flash minimum index granularity is 32B, the bit number of the item offset in hash table will be 36bit and the size of the item valid bitmap will be 256MB. If the bit number of the offset is two large, a cache line (64Byte) only contains few hash items.

Because the Flash write granularity is page and the size of kv item is much smaller than the size of Flash page, the Temporary R/W buffer is used to combine the kv items to satisfy the page size and increase the Flash write efficiency. Because the Flash read granularity is also page, so a Flash page is read to the Temporary R/W buffer firstly and then select the correct kv item from it.

#### Functions

|  |  |
| --- | --- |
| Function number | Function Description |
| 1 | Provide SET, GET, DEL operation |
| 2 |  |
| 3 |  |

### Flash physical space management (FTL) sub-system

#### Overview

Fig. The data layout of Block manage module

Because there isn’t FTL in our KV system and the Flash is used as MTD, the Flash garbage collection and wear-out balance must be considering in KV system. In order to optimize the performance of garbage collection and wear-out balance, we store the information of the Flash erase times and the valid data ratio of each block in SCM. The list of Flash block which can be used is also stored in SCM. shows the data layout of Block manage module in SCM.

Another function of Flash physical space management (FTL) sub-system is GC.

Fig. Primary work flow of GC

The primary work flow of GC is shown in . There are five steps for GC. The primary work flow guarantees the failure recovery consistent, but the concurrent consistent also cannot be guaranteed. The analysis is the same as the analysis mentioned above. The failure consistent is guaranteed by the approach mention above.

Step1. Read the data which will be erased to DRAM’s Temporary R/W Buffer.

Step2. Select the valid kv items based on the Item valid bitmap and write to another DRAM’s Temporary R/W Buffer for writing Flash.

Step3. Add some evicted hot data into the DRAM’s Temporary R/W Buffer for writing Flash to integrate a Flash block.

Step4. Write the data into Flash and wait for success return.

Step5. Change Cold data Hash item offset to the new position, build Cold data Hash Item for the evicted hot data, Delete Hot data Hash item for the evicted hot data, move Hot Data Circular log head pointer.

#### Functions

|  |  |
| --- | --- |
| Function number | Function Description |
| 1 | Provide Flash block malloc |
| 2 | Provide GC |
| 3 | Provide wear-out balance |

### Flash Concurrent IO management sub-system

#### Overview

#### Functions

|  |  |
| --- | --- |
| Function number | Function Description |
| 1 |  |
| 2 |  |
| 3 |  |

### Flash physical space management (NVMe) sub-system

4.2.8.1 NVMe basics and its benefits for NVKV

NVM Express is a scalable host controller interface designed for Enterprise and client systems that use PCI Express SSDs.

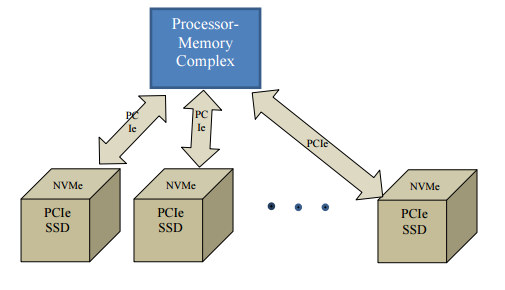


Fig. PCIe attached SSD with NVMe interface

The deployment of SSDs, with performance capabilities orders of magnitude greater than previous storage devices, especially the low latency characteristics of the devices, drove the transition from physical attachments based on traditional storage busses to physical interconnects more closely tied to the processor-memory complex, namely the PCIe bus.

With a storage device moving from a legacy storage interconnect to the low latency system interconnect the need for a new storage device interface that could span both the storage domain and function equally well within the system interconnect domain and unlock the full potential of these new devices was required. NVMe is that new interface.

The interface was also designed to be highly parallel and highly scalable. The scalability, parallelism and inherent efficiency of NVMe allow the interface to scale up and down in performance without losing any of the benefits. These features allow the interface to be highly adaptable to a wide variety of system configurations and designs from laptops to very high end, highly parallel servers.

Another important feature of the NVMe interface is its ability to support the partitioning of the physical storage extent into multiple logical storage extents, each of which can be accessed independently of other logical extents. These logical storage extents are called Namespaces. Each NVMe Namespace may have its own pathway, or IO channel, over which the host may access the Namespace. In fact, multiple IO channels may be created to a single Namespace and be used simultaneously (Note that an IO channel, i.e. a submission/completion queue pair is not limited to addressing one and only one Namespace; see the NVMe specification for details).

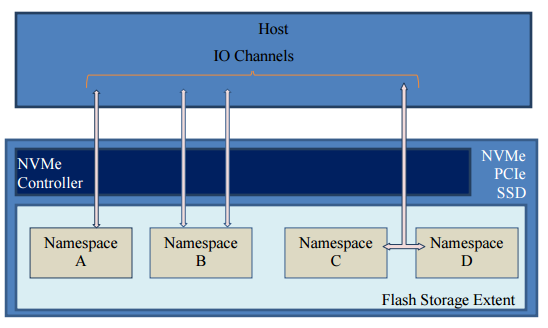


Fig. Namespace and IO channel mapping

The ability to partition a physical storage extent into multiple logical storage extents and then to create multiple IO channels to each extent is a feature of NVMe that was architected and designed to allow the system in which it is used to exploit the parallelism available in upper layers of today’s platforms and extend that parallelism all the way down into the storage device itself.

Multiple IO channels that can be dedicated to cores, processes or threads eliminate the need for locks, or other semaphore based locking mechanisms around an IO channel. This ensures that IO channel resource contention, a major performance killer in IO subsystems, is not an issue.

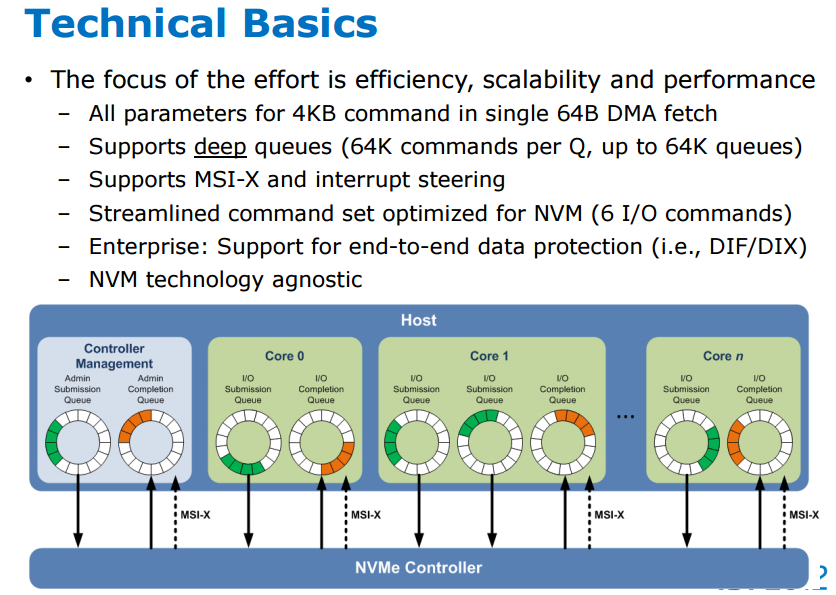


Fig. Cores and Queues affinity

* + - 1. Intel P3700 SSD capabilities and limitations

Complete product specifications can be found under the following location: <http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-dc-p3700-spec.pdf>. We mark only the most important features here:

* NVMe 1.0c compliant (<http://www.nvmexpress.org/wp-content/uploads/NVM-Express-1_0e.pdf>);
* PCIe Gen3 X4;
* Variable Sector Size (referenced to as Logical Block (LB) in the document): **512** (default, a minimum addressable entity which is also an atomic write size for the controller), **520** (extended), 528 (extended), **4096**, 4104(extended), 4160(extended), 4224(extended) Bytes. The controller supports 6 physical formats of LB size and associated metadata size. There may be performance differences between different physical formats;
* Fused operations are not supported (fuse operations enable a more complex command by “fusing” together two simpler commands and run it as an atomic unit);
* 512B write atomicity only;
* Only one namespace is supported:

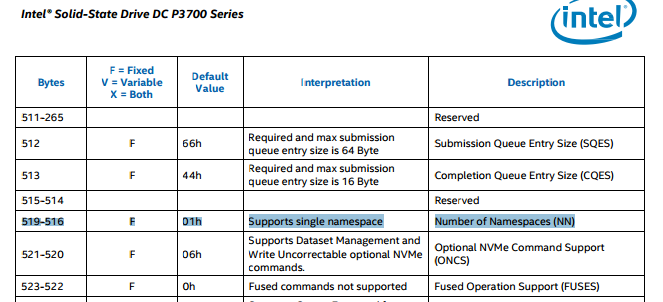


Fig. Intel P3700 SSD supports only 1 namespace and doesn’t support fused operations

* Hot plug is supported (on surprise hot removal during IOs, P3700 Series will ensure the integrity of already committed data on the media and commit acknowledged writes to the media).

4.2.8.3 NVMe space management sub-system

Intel P3700 SSD uses internal FTL (a firmware which is a part of SSD flash controller and is not S/W accessible) and therefore we do not need any additional GC procedure for this case in our system. The only component which is added is space defragmentation routine. NVMe helps us to greatly simplify the design of (cold) space management sub-system.

FTL is the main control software in SSDs that gives an illusion of general hard disks, hiding the unique characteristics of NAND flash memory from the host. One primary technique of FTL to achieve this is to map Logical Block Addresses (LBA) from the host to physical addresses in flash memory which is not directly accessible. When a write request arrives, FTL writes the arrived data to a page in an “erased” state and updates the mapping information to point to the location of the up-to-date physical page. The old page that has the original copy of data becomes unreachable and obsolete. A read request is served by reading the page indicated by the mapping information.

Another vital FTL function is called Wear Leveling. Because NAND-flash cells are wearing off, one of the main goals of the FTL is to distribute the work among cells as evenly as possible so that blocks will reach their P/E cycle limit and wear off at the same time. This ability also simplifies our system design, because we do not need to keep track of how much times each page was programmed and erased.

Yet another important function of FTL is garbage collection (GC). GC is defined as a process that erases dirty blocks which have obsolete pages and recycles these pages. If a block selected to be erased has valid pages, those pages are migrated to other blocks before erasing the block. All these actions are transparent to end-user, although we can provide some assistance to this process by issuing write operations of new data (which is not associated with any LB in use) in a log-based fashion – this will to shorten a data handling path inside FTL, because writing to a free LB which is in the end of the log in most cases won’t raise the process of “logical-to-physical” (LBA-to-PBA) remapping.

At the same time, in order to reduce space defragmentation, a defragmentation process call frequency and Cold Data Hashtable offset remapping, it may seem that we can update data inside particular LB which is somewhere in the middle of the log (SSD FTL allows us to do so). This approach, in general, requires to read a particular LB, update the selected value(s) it holds and write it back without even knowing the details of how the underlying FTL will handle this situation. But we should keep in mind that a physical flash page can be written to only if it is in the “free/erased” state (which is not true in case of updating). When data is updated, the content of the page is copied by FTL into internal SSD buffer, the data is updated, and the new version is stored in a “free” page, an operation called **read-modify-write** - operations of such kind, especially if they are numerous, produce a very bad impact on the overall SSD performance. The data is not updated in-place, as the “free” page is a different page than the page that originally contained the data. Once the data is persisted to the drive, FTL performs remapping operation and marks the original page as “stale” (it will remain as such until it is erased by FTL’s GC as a part of a whole flash physical block containing it, because pages are erased on per-block basis).

P3700 SSD uses 4K physical sector size which is naturally flash page aligned. If we use 4K LB size (this is called a native mode), all I/O operations correspond directly to 4K physical sectors, meaning no I/O can ever be less than 4K in size. By default, P3700 SSD uses 512B LB size, which is provided for emulation mode for backward compatibility (lots of software, including filesystems, still using 512B I/O). A process reading data in a block of 512 bytes will actually cause all 4k of a physical sector to be read, with any other unwanted blocks then discarded. But such read operations do not bring significant overhead. The things become worse in case of write operation as described above - a process writing data in a block of 512 bytes causes much more additional unseen work to take place in the background (FTL), since the storage system has to read all 4k of data into memory, change the single 512 byte block and then write the whole 4k sector back to SSD again. This is called a **partial write** and the action required is known as a **read-modify-write** operation. This situation should be avoided.

Another issue is alignment. If, for example, a number of write I/Os issued to storage do not begin at the start of a physical sector, misalignment results in additional writes as the last logical block overlaps into a new physical sector – and consequently results in at least one extra I/O, which will be a partial write. This is called a **misaligned I/O**. By its very nature, misaligned I/O can cause big problems with performance simply because it massively increases the amount of I/O taking place, as well as causing partial I/O pretty much everywhere.



Fig. 29 Misaligned I/O example

Our NVMe H/W has one drawback – it supports only one namespace. It means, that we won’t be able to benefit from system partition based block allocation when a namespace containing a range of LBs is mapped to a certain sub-partition thus exploiting the full nature of NVMe internal parallelism. For our system we will use the whole range of LBs and keep one common log for all partitions. Therefore simultaneous access to the same LB from different partitions can possibly lead to concurrency issues (possible ?) TODO: ………...

To summarize all of the above we elaborate the following access patterns which will allow us to get the maximum from the underlying NVMe H/W (Intel P3700 SSD) for our system:

* Use native 4K LB size which is equal to physical flash page size. This will guarantee that there will be zero partial or misaligned I/O;
* Ensure that logical writes are truly aligned to the physical memory - align the partition to the physical page size of the drive (4K) to reach the maximum possible write performance. Fortunately it’s very easy to avoid misalignment by doing simple things: starting in the right place and using the right I/O size (never write less than a page). Of course, the easiest method is to use native LB size, where misalignment is not possible;
* Invalidate obsolete data in a batch (this is important for defragmentation process which is responsible for reclaiming SSD free space for the system). When some data needs to be deleted, it is better to wait and invalidate it in large batches in a single operation. This will allow the FTL’s GC to handle larger areas at once and will help minimizing internal fragmentation. For this, we use Item Valid Bitmap in SCM the same way as for Flash Physical Space Management system;
* To improve the read performance, write related data together (is it possible for us? can we apply reftag/apptag fields for this? How much profit will it bring if implemented? – refer to Protection Information Format for details). Read performance is a consequence of the write pattern. When a large chunk of data is written at once, it is spread across separate NAND-flash chips. Thus we must try to write related data in the same page whenever possible, so it can later be read faster with a single I/O request, by taking advantage of the internal parallelism;
* Buffer hot data. Because the Flash read/write granularity is page and the size of kv item is much smaller than the size of Flash page, the Temporary R/W buffer and Read Cache are used to suit for Flash read/write operations. The Temporary R/W buffers and Read Cache are stored in DRAM, because those buffers are read/write frequency and the data in those buffer can be lost when failure recovery. This approach is already put into Cold Data management sub-system design and we are going to reuse it for NVMe case;
* “In-place” update of LB:
  + Variant A: Never perform “in-place” update of LB whose valid item ratio is less than or equal to 50% (this threshold may be different) – place updated item to the end of the log (through putting it into R/W buffer). This is a tradeoff between performance and space defragmentation. As we already mentioned, read-modify-write introdues a bad impact on flash performance, especially when it is performed too often. But at the same time we want to use SSD available space as efficient as possible without calling system defragmentation process too often, because it also negatively affects system performance. So when the majority of items belonging to particular LB we want to update are valid, we may perfrom “in-place” update thus postponing the system defragmentation routine. It is also not reasonable to update a single item but rather perfrom a batch update of items belonging to the same LB whenever possible. If the majority of items inside LB are invalid, then there is no need in “in-place” update, since such LB is a candidate for defragmentation routine already.
  + Variant B (preferable): Never perform “in-place” update of LB and always put a new version of item to the end of the log. This approach is easier to implement and in most cases it will do faster, but will make SSD defragment faster.

TODO: Defragmentator design and logic

TODO: Concurrency issues for one common SSD log (?)

### Network

#### Overview

#### Functions

|  |  |
| --- | --- |
| Function number | Function Description |
| 1 |  |
| 2 |  |
| 3 |  |

## Dependency Description

## Interface Description

### KV request sub-system and Flash Concurrent IO management sub-system

##### Read a Flash Page

##### add a Read Flash Page request from the read Flash request queue.

|  |  |
| --- | --- |
| Interfance specification | The read is asychronous calling. The kv thread adds the read request into the read Flash queue then responds to other kv request witout waiting the read Flash function return. |
| Function | add\_read\_flash\_page\_request |
| Input | int64\* DRAM\_logical\_address, int64 flash\_physical\_address, int32 partition\_ID, int32 suspend\_queue\_index  the DRAM logical address of the temporary R/W buffer where the read data will be put,  the Flash physical address where the data will be read,  the partition number  the index of the suspend queue, where the suspend item will be stored. |
| Output | NULL |
| Return | NULL |
| Relative data | A read Flash request circular queue : the read Flash request is added at the tail in the read Flash request circular queue. |
| Calling module | KV request sub-system |
| Called module | Flash Concurrent IO management sub-system |
| Reamark | Lock or atomic read/write will be used to guarantee the concurrent consistent. Each kv thread has it own read Flash request queue in order to increase the concurrent performance. If the read Flash request queue is full, the calling thread is block. |

##### Obtain a Read Flash Page request from the read Flash request queue.

|  |  |
| --- | --- |
| Interfance specification | The read is asychronous calling. The Flash read thread obtain the Flash page read request from the read Flash request queue then do the Flash read process. |
| Function | obtain\_read\_flash\_page\_request |
| Input | NULL |
| Output | int64\*\* DRAM\_logical\_address, int64\* flash\_physical\_address, int32\* partition\_ID, int32\* suspend\_queue\_index  the DRAM logical address of the temporary R/W buffer where the read data will be put,  the Flash physical address where the data will be read  the partition number  the index of the suspend queue, where the suspend item will be stored. |
| Return | NULL |
| Relative data | A read Flash request circular queue : the read Flash request is obtain at the head in the read Flash request circular queue. |
| Calling module | Flash Concurrent IO management sub-system |
| Called module | Flash Concurrent IO management sub-system |
| Reamark | Lock or atomic read/write will be used to guarantee the concurrent consistent. Each kv thread has it own read Flash request queue in order to increase the concurrent performance. |

##### Add a Read Flash Page finish in the read Flash finish queue.

|  |  |
| --- | --- |
| Interfance specification | The read is asychronous calling. The read Flash thread adds the read Flash finish mark in the finish queue. |
| Function | add\_read\_flash\_page\_finish |
| Input | int64 DRAM\_logical\_address, int32 suspend\_queue\_index  the DRAM logical address of the temporary R/W buffer where the read data will be put, which is used as a tag. Using this tag, the kv thread can identify which read request is finished.  the index of the suspend queue, where the suspend item will be stored. |
| Output | NULL |
| Return | NULL |
| Relative data | A read Flash finish circular queue : the read Flash finish is added at the tail in the read Flash finsih circular queue. |
| Calling module | Flash Concurrent IO management sub-system |
| Called module | Flash Concurrent IO management sub-system |
| Reamark | Lock or atomic read/write will be used to guarantee the concurrent consistent. Each kv thread has it own read Flash finish queue in order to increase the concurrent performance. If the read Flash finish queue is full, the calling thread is block. |

##### Obtain a Read Flash Page finish retuen from the read Flash finish queue.

|  |  |
| --- | --- |
| Interfance specification | The read is asychronous calling. The kv thread obtains the read Flash finish mark fron the finish queue. |
| Function | obtain\_read\_flash\_page\_finish |
| Input | NULL |
| Output | int64\* DRAM\_logical\_address, int\* suspend\_queue\_index  the DRAM logical address of the temporary R/W buffer where the read data will be put, which is used as a tag. Using this tag, the kv thread can identify which read request is finished.  the index of the suspend queue, where the suspend item will be stored. |
| Return | NULL |
| Relative data | A read Flash finish circular queue : the read Flash finish is obtained at the head in the read Flash finsih circular queue. |
| Calling module | KV request sub-system |
| Called module | Flash Concurrent IO management sub-system |
| Reamark | Lock or atomic read/write will be used to guarantee the concurrent consistent. Each kv thread has it own read Flash queue in order to increase the concurrent performance. |

### Hot data sub-system and Flash Concurrent IO management sub-system

##### Write a Flash Block

##### Add a Write Flash Page request in the write Flash request queue.

|  |  |
| --- | --- |
| Interfance specification | The write is asychronous calling. The kv thread adds the write request into the write Flash queue then responds to other kv request witout waiting the write Flash function return. |
| Function | add\_write\_flash\_page\_request |
| Input | int64\* DRAM\_logical\_address, int32 partition\_ID, int32 suspend\_queue\_index  the DRAM logical address of the temporary R/W buffer where the read data will be put,  the thead number  the partition number  the index of the suspend queue, where the suspend item will be stored. |
| Output | NULL |
| Return | NULL |
| Relative data | A write Flash request circular queue : the write Flash request is added at the tail in the write Flash request circular queue. |
| Calling module | Hot data sub-system |
| Called module | Flash Concurrent IO management sub-system |
| Reamark | Lock or atomic read/write will be used to guarantee the concurrent consistent. Each kv thread has it own write Flash request queue in order to increase the concurrent performance. If the write Flash request queue is full, the calling thread is block. |

##### Obtain a Write Flash Page request from the write Flash request queue.

|  |  |
| --- | --- |
| Interfance specification | The write is asychronous calling. The Flash write thread obtain the Flash block write request from the write Flash request queue then do the Flash write process. |
| Function | obtain\_write\_flash\_page\_request |
| Input | NULL |
| Output | int64\*\* DRAM\_logical\_address, int32\* partittion\_ID, int32\* suspend\_queue\_index  the DRAM logical address of the temporary R/W buffer where the read data will be put,  the ID of the queue  the partition number  the index of the suspend queue, where the suspend item will be stored. |
| Return | NULL |
| Relative data | A write Flash request circular queue : the read Flash request is obtain at the head in the write Flash request circular queue. |
| Calling module | Flash Concurrent IO management sub-system |
| Called module | Flash Concurrent IO management sub-system |
| Reamark | Lock or atomic read/write will be used to guarantee the concurrent consistent. Each kv thread has it own write Flash request queue in order to increase the concurrent performance. |

##### Add a Write Flash Page finish in the write Flash finish queue.

|  |  |
| --- | --- |
| Interfance specification | The write is asychronous calling. The write Flash thread adds the write Flash finish mark and Flash physical address in the finish queue. |
| Function | add\_write\_flash\_page\_finish |
| Input | int64 DRAM\_logical\_address, int64 Flash\_physical\_address, int32 suspend\_queue\_index  the DRAM logical address of the temporary R/W buffer where the read data will be put, which is used as a tag. Using this tag, the kv thread can identify which read request is finished,  The Flash physical address where the data is written.  the index of the suspend queue, where the suspend item will be stored. |
| Output | NULL |
| Return | NULL |
| Relative data | A write Flash finish circular queue : the write Flash finish is added at the tail in the write Flash finsih circular queue. |
| Calling module | Flash Concurrent IO management sub-system |
| Called module | Flash Concurrent IO management sub-system |
| Reamark | Lock or atomic read/write will be used to guarantee the concurrent consistent. Each kv thread has it own write Flash finish queue in order to increase the concurrent performance. If the write Flash finish queue is full, the calling thread is block. |

##### Obtain a Write Flash Page finish return from the write Flash finish queue.

|  |  |
| --- | --- |
| Interfance specification | The write is asychronous calling. The kv thread obtains the write Flash finish mark from the finish queue. |
| Function | obtain\_write\_flash\_page\_finish |
| Input | NULL |
| Output | int64\* DRAM\_logical\_address, int64\* Flash\_physical\_address, int32\* suspend\_queue\_index  the DRAM logical address of the temporary R/W buffer where the read data will be put, which is used as a tag. Using this tag, the kv thread can identify which read request is finished,  The Flash physical address where the data is written,  the index of the suspend queue, where the suspend item will be stored. |
| Return | NULL |
| Relative data | A write Flash finish circular queue : the write Flash finish is obtained at the head in the write Flash finsih circular queue. |
| Calling module | Hot data sub-system |
| Called module | Flash Concurrent IO management sub-system |
| Reamark | Lock or atomic read/write will be used to guarantee the concurrent consistent. Each kv thread has it own write Flash queue in order to increase the concurrent performance. |

### In Flash Concurrent IO management sub-system

##### Calling the GC function, which is use to finish the GC process

|  |  |
| --- | --- |
| Interfance specification | GC process |
| Function | Flash\_GC |
| Input | int64 Flash\_physical\_address list  The Flash physical address list where the data is GC. |
| Output | NULL |
| Return | NULL |
| Relative data | GC request queue, Free Block list |
| Calling module | Flash Concurrent IO management sub-system |
| Called module | Flash Concurrent IO management sub-system |
| Reamark |  |

# Level 2 Design Description

## Initail sub-system

### Sub-system Design

* **Initial DRAM and SCM**

The initial process of DRAM and SCM is very similar to the initial process of DRAM in MICA. The slight different is that we mask the memory block belongs to DRAM or SCM.

* **Malloc DRAM and SCM**

The malloc process of DRAM and SCM is very similar to the malloc process of DRAM in MICA. The slight different is that we distinct the memory type belonging to DRAM or SCM.

* **Failure Recovery Work Flow**

When it reboots from failure recovery, the process of the rebuild the Hot Data Hash as following: scan the Hot data Circular log and insert the corresponding hash item to the Hot Data Hash table.

We will continue the process of Evict and GC process which breaks down by the failure.

### Module Design

|  |  |
| --- | --- |
| Module name | Duty |
| Initial DRAM and SCM | The initial process of DRAM and SCM is very similar to the initial process of DRAM in MICA. The slight different is that we mask the memory block belongs to DRAM or SCM. |
| Malloc DRAM and SCM | The malloc process of DRAM and SCM is very similar to the malloc process of DRAM in MICA. The slight different is that we distinct the memory type belonging to DRAM or SCM |
| Failure Recovery Work Flow | When it reboots from failure recovery, the process of the rebuild the Hot Data Hash as following: scan the Hot data Circular log and insert the corresponding hash item to the Hot Data Hash table.  We will continue the process of Evict and GC process which breaks down by the failure |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

* **Initial DRAM and SCM**
* **Data Structure**

**struct shm\_page**

**{**

**char path[PATH\_MAX];**

**void \*addr;**

**void \*paddr;**

**size\_t numa\_node;**

**size\_t in\_use;**

**};**

struct shm\_page is used to record the information of each 2MB DRAM or SCM page.

### Work flow in the sub-system

### Interface in the sub-system

|  |  |
| --- | --- |
| Interfance specification | Initial the DRAM and SCM, mask the numa domain for each 2MB physical memory. |
| Function | shm\_ DRAM\_SCM\_init |
| Input | size\_t num\_numa\_nodes , // the number of the numa nodes  size\_t page\_size\_DRAM , // the size of physical memory page of DRAM  size\_t num\_pages\_to\_try\_DRAM , // the number of DRAM page will be try to find suitable page  size\_t num\_pages\_to\_reserve\_DRAM, // the number of DRAM page will be reserved  size\_t page\_size\_SCM , // the size of physical memory page of SCM  size\_t num\_pages\_to\_try\_SCM , // the number of SCM page will be try to find suitable page  size\_t num\_pages\_to\_reserve\_SCM); // the number of SCM page will be reserved |
| Output |  |
| Return |  |
| Relative data | #define SHM\_DRAM\_MAX\_PAGES (65536)  #define SHM\_DRAM\_MAX\_ENTRIES (8192)  #define SHM\_DRAM\_MAX\_MAPPINGS (16384)  #define SHM\_SCM\_MAX\_PAGES (65536)  #define SHM\_SCM\_MAX\_ENTRIES (8192)  #define SHM\_SCM\_MAX\_MAPPINGS (16384)  static size\_t shm\_DRAM\_page\_size;  static uint64\_t shm\_DRAM\_state\_lock;  static struct shm\_page shm\_DRAM\_pages[SHM\_DRAM\_MAX\_PAGES];  static struct shm\_entry shm\_DRAM\_entries[SHM\_DRAM\_MAX\_ENTRIES];  static struct shm\_mapping shm\_DRAM\_mappings[SHM\_DRAM\_MAX\_MAPPINGS];  static size\_t shm\_DRAM\_used\_memory;  static size\_t shm\_SCM\_page\_size;  static uint64\_t shm\_SCM\_state\_lock;  static struct shm\_page shm\_SCM\_pages[SHM\_SCM\_MAX\_PAGES];  static struct shm\_entry shm\_SCM\_entries[SHM\_SCM\_MAX\_ENTRIES];  static struct shm\_mapping shm\_SCM\_mappings[SHM\_SCM\_MAX\_MAPPINGS];  static size\_t shm\_SCM\_used\_memory;  static const char \* shm\_path\_prefix = "/mnt/huge/mehcached\_shm\_"; |
| Calling module |  |
| Called module |  |
| Reamark |  |

## KV request sub-system

### Sub-system Design

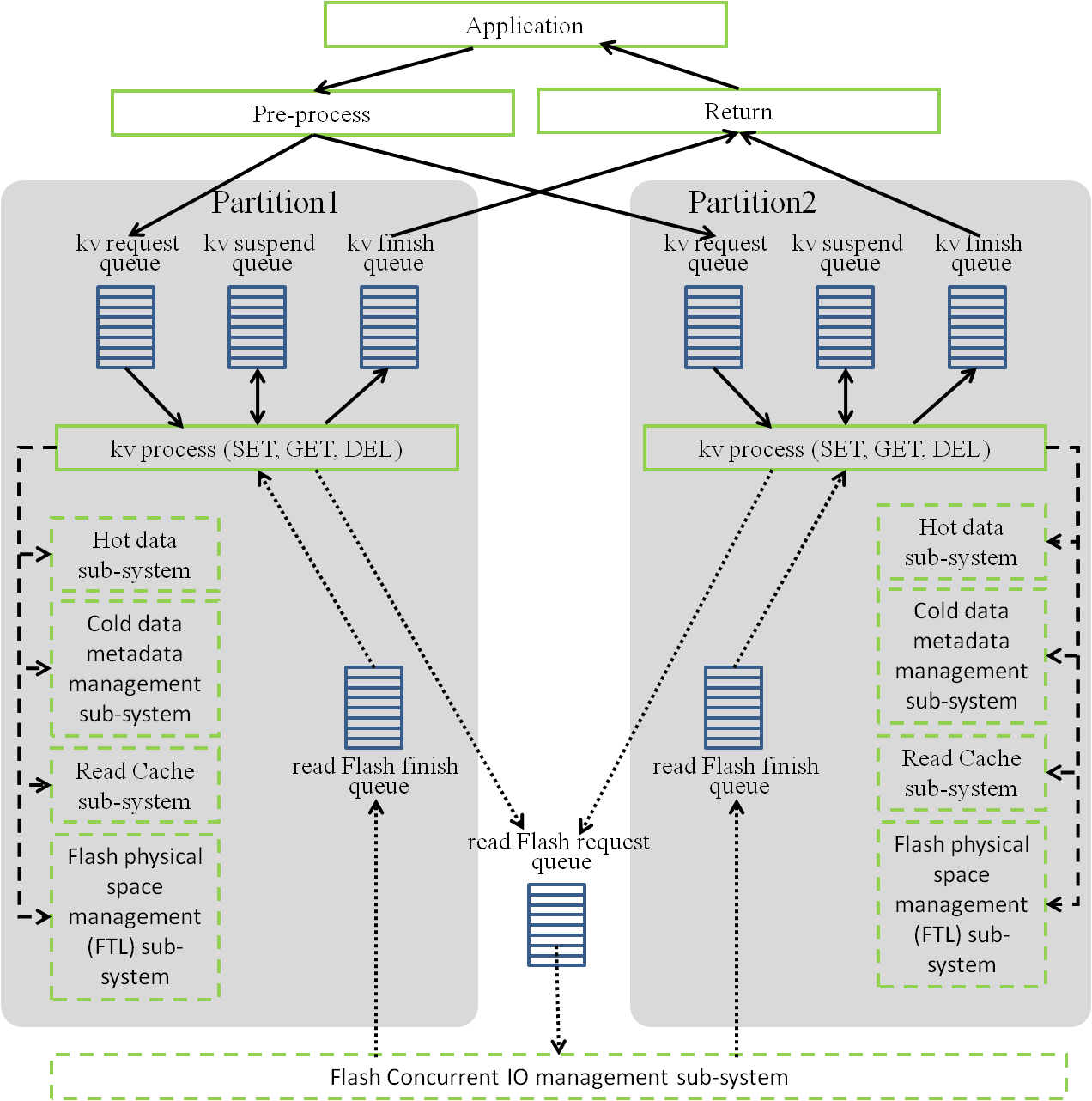
****

Fig. Framework of the kv request sub-system

In order to reduce the workload unbalance with lock elision, we add a kv request queue to each kv thread. The kv request operation first put at the tail in the kv request queue and the corresponding thread get the kv request from the header of the request queue. The process of kv request maybe have interaction with the Hot data sub-system, Read Cache sub-system, Cold metadata management sub-system and Flash physical space management sub-system. Those interface functions are synthronous. The process of kv request maybe have interaction with the Flash Concurrent IO manangement sub-system, whose interface functions are asynthronous, so some queue must be used.

Because the read Flash process is asynthronous, a kv request will be suspended when that kv process of that kv request need read Flash process and other kv request will be processed. In order to achieve that requirement, the kv suspend queue and kv finish queue must be needed. The kv suspend queue stores the state of the suspended kv request and the kv finish queue store the finished kv request.

When a kv request from the application achieves the KV system, the pre-process distributes the kv request into different kv thread (Partition) and when Return thread polls the kv finished queue and return the result to the appliaction when the Return thread finds a finished kv request.

Because the read Flash process is asynthronous, when a kv need a read Flash process, that kv thread add the read Flash request at the tail in the read Flash request queue. When the kv thread finish a kv request or suspend a kv request, that kv thread will check whecher there is a new item in it own read Flash finish queue. If there is a new item, the kv thread obtain that item and continue to process the corresponding suspended kv request. If there is no new item, the kv thread will check whecher there is a new item in it own kv request queue.

### Module Design

|  |  |
| --- | --- |
| Module name | Duty |
| Pre-process  module | distributes the kv request into different kv thread (Partition) based on hash value of the key |
| Return  module | return the result of the finished kv request to the appliaction |
| kv request queue module | Store the kv request which are not processed |
| kv suspend queue module | stores the state of the suspended kv request |
| kv finished queue module | stores the state of the finised kv request |
| SET module | SET process |
| GET module | GET process |
| DEL module | DEL process |
|  |  |
|  |  |

* **Pre-process module**
* **Return module**
* **kv request queue module**

Fig. 26 shows the structure of the kv request queue. The kv request queue uses the circular queue. Each item in the kv request queue contains <operation type, key address, key length, value address, value length, extra information address, extra information length> The key, value and extra information are stored in the address where key address, value address and extra information address points respectively.

Fig. structure of the kv request queue

**struct kv\_request\_queue\_item**

**{**

**uint8\_t operation\_type; // SET=0,SET=1,DEL=2**

**uint8\_t \*key\_address; // key**

**uint32\_t key\_length; // key length**

**uint8\_t \*value\_address; // value**

**uint32\_t value\_length; // value length**

**uint8\_t \*****extra\_info\_address; // the appended information, such as network package**

**uint32\_t extra \_info\_length; // appended information length**

**};**

**struct kv\_request\_queue**

**{**

**struct kv\_request\_queue\_item \*item;**

**uint32\_t num\_item;**

**uint32\_t num\_item\_mask;**

**uint32\_t head;**

**uint32\_t tail;**

**uint32\_t lock;**

**};**

* **kv suspend queue module**

shows the structure of the kv suspend queue. The kv suspend queue uses the circular queue. Each item in the kv suspend queue contains < operation type, key, Temporary R/W buffer logical address, Hash bucket index, Hash item index, offset in Flash page>.

Fig. structure of the kv suspend request queue

* **kv finished queue module**

shows the structure of the kv request queue. The kv request queue uses the circular queue. Each item in the kv request queue contains <operation type, key address, key length, value address, value length, extra information address, extra information length> The key, value and extra information are stored in the address where key address, value address and extra information address points respectively.

Fig. structure of the kv finished request queue

### Work flow in the sub-system

* **SET**
* Lock version
* Synchronous Flash read (synchronous Flash read is not used in our KV system with MTD Flash )

In order to guarantee the concurrent consistent, we use the RW Lock. The detail of the SET process as following:

The lock design must guarantee the concurrent consistent, as the same time the concurrent performance must be considering. Unnecessary lock must be discarded in order to increase the concurrent performance. We use the Read and Write Lock to guarantee the concurrent consistent of Hash table and others data structure which may be accessed by multithreads at the same time. The rules of lock’s optimization are that: (1) must guarantee the concurrent consistent; (2) must guarantee no deadlock; (3) discard unnecessary lock; (4) the critical section as short as possible; (4) use Read Lock not Write Lock if Read Lock can guarantee the concurrent consistent.

Our KV system support EREW, CREW and CRCW models. In order to obtain high performance for each model, the Lock must be optimized for each model. We can used #ifdef method to configure the lock in our KV system.

Fig. Work Flow of SET with lock and synchronous Flash read

Bucket address is already 64-byte aligned. Automatic operation must be used to guarantee the failure consistent.

* Asynchronous Flash read (asynchronous Flash read is used in our KV system with MTD Flash )

Fig. 30 shows the work flow of SET with lock and asynchronous Flash read. Fig. 30(a) is the process when the kv thread obtain a SET request from the kv request queue. Because the Flash read is asynchronous, that process maybe be suspended after add a read Flash request and add a kv suspend queue. The oranger color block presents a process that it is only done when it can be done. If that process can not be done, it will be block until it can be done. Take “The tail of read Flash request queue ++” for example, if the tail-head>=queue length, that process must be block tail-head<queue length. (b) is the process when the kv thread obtain a SET request from the kv suspend queue.

(a)

(b)

Fig. Work Flow of SET with lock and asynchronous Flash read

* STM version

The work flow of SET using STM is the same as that using lock.

* TSX version
* **GET**
* Lock version
* Synchronous Flash read (synchronous Flash read is not used in our KV system with MTD Flash )

Fig. Work Flow of GET with lock and synchronous Flash read

* Asynchronous Flash read (asynchronous Flash read is used in our KV system with MTD Flash )

(a)

(b)

Fig. Work Flow of GET with lock and asynchronous Flash read

* STM version
* Synchronous Flash read (synchronous Flash read is not used in our KV system with MTD Flash )

Fig. Work Flow of GET with STM and synchronous Flash read

* Asynchronous Flash read (asynchronous Flash read is used in our KV system with MTD Flash )

(a)

(b)

Fig. Work Flow of GET with STM and asynchronous Flash read

* TSX version
* **DEL**
* Lock version
* Synchronous Flash read (synchronous Flash read is not used in our KV system with MTD Flash )

Fig. Work Flow of DEL with lock and synchronous Flash read

* Asynchronous Flash read (asynchronous Flash read is used in our KV system with MTD Flash )

(a)

(b)

Fig. Work Flow of DEL with lock and asynchronous Flash read

* TSX version

### Interface in the sub-system

|  |  |
| --- | --- |
| Interfance specification |  |
| Function |  |
| Input |  |
| Output |  |
| Return |  |
| Relative data |  |
| Calling module |  |
| Called module |  |
| Reamark |  |

## Hot data sub-system

### Sub-system Design

Fig. The data layout of Hot data module

Fig. 37 shows the data layout of the Hot data, more detail has been analyised above. The Evict process needs write Flash operation, bacuase the read/write Flash Functions are asynchronous, the queue must be used. The work flow is shown in . When the kv thread which belongs to the request sub-system processes the SET kv request, it can find the free space of the Hot data circular log is less than the threshold. When the free space of the Hot data circular log is less than the threshold, the kv thread puts an evict request in the evict request queue. When the evict thread which belongs to the Hot data sub-system finds there are new evict request in the evict request queue, it will process the evict request. Because the write Flash operation is asynchronous, when the evict thread needs write Flash, it adds the write Flash request to the write Flash request queue and add an evict suspend item which records the state of the suspend evict request in the evict suspend queue, then it will process another evict request. When the write Flash thread which belongs to the Flash Concurrent IO management sub-system finds write Flash request in write Flash request queue, it will process the write Flash request. When the write Flash thread finsihs write Flash request, it will add the the write Flash finish item to the write Flash finish queue. When the evict thread finds a write Flash finish item in the write Flash finish queue, it will finds the corresponding suspend evict request in the evict suspend queue and continue the process of the suspend evict request.

Fig. Work flow of evict

### Module Design

|  |  |
| --- | --- |
| Module name | Duty |
| Lossless Hash table | Provide Hash operation of the Hot data |
| Cirular log | Manage the free space which used to store the hot kv item |
| Data evict | Evict the kv item from the Hot data to the Cold data |
| Evict request queue | Manage the new evict request |
| Evict suspend queue | Manage the suspend evict request |

* **Lossless concurrent hash table module**

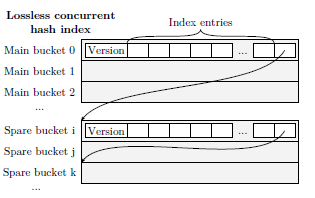
****

Fig. 39 Lossless concurrent hash table

Because the kv items in the Hot data cannot be abandoned arbitrarily, when the bucket is full, another Spare bucket will be listed to the bucket. shows the data structure of the Lossless concurrent hash table. The last item in each bucket is a pointer which indexes the Spare bucket position.

The same as the approach of MICA, we use the approach that combines the software transactional memory and lock. Deal with the read process, we use the STM technology. Deal with the write process, we use the Lock method.

In order to support that mixed approach, the Hash bucket contains a version, which is used as the timestamp in STM technology and lock in Lock method. The last bit of the version is used by lock and the rest bits are used as timestamp.

The process of obtaining the timestamp as follow:

1. Read operation using STM technology

The process of read a date is that first get the version, read the data, then get the version and comparison that with the first version. If both the version is same, return the data, otherwise redo the read operation. The functions of read the version as follow:

**Static uint32\_t mehcached\_read\_version\_begin(const struct mehcached\_bucket \*bucket)**

**{**

**while (true)**

**{**

**uint32\_t v = \*(volatile uint32\_t \*)&bucket->version;**

**memory\_barrier();**

**if ((v & 1U) != 0U) continue; //present the bucket is not written**

**return v;**

**}**

**}**

**Static uint32\_t mehcached\_read\_version\_end(const struct mehcached\_bucket \*bucket)**

**{**

**memory\_barrier();**

**uint32\_t v = \*(volatile uint32\_t \*)&bucket->version;**

**return v;**

**}**

1. Write operation using Lock
2. Lock in exclusive write mode

**Static void mehcached\_lock\_bucket(struct mehcached\_bucket \*bucket)**

**{**

**(\*(volatile uint32\_t \*)&bucket->version)++;**

**memory\_barrier();**

**}**

1. Lock in concurrent write mode

**Static void mehcached\_lock\_bucket(struct mehcached\_bucket \*bucket)**

**{**

**while (1)**

**{**

**uint32\_t v = \*(volatile uint32\_t \*)&bucket->version & ~1U;**

**uint32\_t new\_v = v | 1U;**

**if (\_\_sync\_bool\_compare\_and\_swap((volatile uint32\_t \*)&bucket->version, v, new\_v))**

**break;**

**}**

**}**

1. Unlock

**Static void mehcached\_unlock\_bucket(struct mehcached\_bucket \*bucket)**

**{**

**memory\_barrier();**

**(\*(volatile uint32\_t \*)&bucket->version)++;**

**}**

* **Circular log** **module**
* Circular log

It is the same as MICA

* kv item data structure

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Initial size | Keyhash | Key/value length | Expire time | Valid mask |
| key | | | | |
| value | | | | |

Fig. kv item data structure

shows the data structure of kv item and more detail is described below, which is the same as MICA.

**struct mehcached\_item**

**{**

**struct mehcached\_alloc\_item alloc\_item;**

**uint32\_t kv\_length\_vec; // key\_length: 8, value\_length: 24; kv\_length\_vec == 0: empty item**

**#define MEHCACHED\_KEY\_MASK (((uint32\_t)1 << 8) - 1)**

**#define MEHCACHED\_KEY\_LENGTH(kv\_length\_vec) ((kv\_length\_vec) >> 24)**

**#define MEHCACHED\_VALUE\_MASK (((uint32\_t)1 << 24) - 1)**

**#define MEHCACHED\_VALUE\_LENGTH(kv\_length\_vec) ((kv\_length\_vec) & MEHCACHED\_VALUE\_MASK)**

**#define MEHCACHED\_KV\_LENGTH\_VEC(key\_length, value\_length) (((uint32\_t)(key\_length) << 24) | (uint32\_t)(value\_length))**

**// the rest is meaningful only when kv\_length\_vec != 0**

**uint32\_t expire\_time;**

**uint64\_t key\_hash;**

**uint8\_t data[0];**

**};**

* **Evict request queue module**

Fig. 41 shows the data structure of the evict request queue. The evict request queue uses the circular queue. Each item in the evict request queue contains < partition number>.

Fig. Data structure of the evict request queue

* **Evict suspend queue module**

shows the data structure of the evict suspend queue. The evict suspend queue uses the circular queue. Each item in the evict suspend queue contains < partition number>.

Fig. Data structure of the evict suspend queue

### Work flow in the sub-system

* **Write Flash / Evict**
* Lock version
* Synchronous Flash read/write (synchronous Flash read/write is not used in our KV system with MTD Flash )

The Design of the Evict must consider the situation that the KV system Update or Delete the data which is ready to write to Flash. If we add Write Lock to all the data which is ready to write to Flash and is already in Temporary R/W buffer, all requests of those data whose hash item is in the same bucket as the hash item of the data in Temporary R/W buffer are delay, because the Flash write process is time expensive. Even if we add Read Lock, the Write and Delete requests are delay. That approach affects the responding. Another approach is that check valid for all data which are written to Flash when delete the Hot data Hash table and mask the invalid kv item which are changed during the Flash write process.

Fig. Work Flow of Evict with lock and synchronous Flash read/write

* Asynchronous Flash read/write (asynchronous Flash read/write is used in our KV system with MTD Flash )

(a)

(b)

Fig. Work Flow of Evict with lock and synchronous Flash read/write

* TSX version

### Interface in the sub-system

|  |  |
| --- | --- |
| Interfance specification |  |
| Function |  |
| Input |  |
| Output |  |
| Return |  |
| Relative data |  |
| Calling module |  |
| Called module |  |
| Reamark |  |

## Read Cache sub-system

### Sub-system Design

Fig. The data layout of Read cache module

shows the data layout of the Read cache. The whole Read cache module is stored in DRAM.

### Module Design

|  |  |
| --- | --- |
| Module name | Duty |
| Lossy Hash table | Provide Hash operation of the Read Cache data |
| Double link free space management | Manage the free space which used to store the Read Cache kv item |
| Data evict | Evict the kv item from the Read Cache |
|  |  |

* **Lossy concurrent hash table Design**

Because kv items in the Read cache are only used to improve the GET performance and the kv item can be abandoned arbitrarily. We use the Lossy concurrent hash table. The data structure of the Lossy concurrent hash table is shown in . The item number of each bucket is fixed. If a bucket if full, an item will be evict. There are two approaches to choose which item will be evicted. First approach is based on the key. We use the key mod the item number in each bucket to compute the evicted item index. For example, each bucket has 8 items and key is 27, the mod value is 3, so the item 3 will be evict and the corresponding kv item in Circular log will be mast invalid.

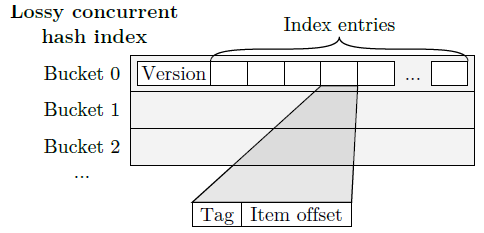
****

Fig. 46 Lossy concurrent hash table

### Work flow in the sub-system

### Interface in the sub-system

|  |  |
| --- | --- |
| Interfance specification |  |
| Function |  |
| Input |  |
| Output |  |
| Return |  |
| Relative data |  |
| Calling module |  |
| Called module |  |
| Reamark |  |

## Cold data metadata management sub-system

### Sub-system Design

### Module Design

|  |  |
| --- | --- |
| Module name | Duty |
| Cuckoo Hash table | Provide Hash operation of the Cold data |
|  |  |

* **Cuckoo Hash table** **Design**

In order to save memory space and reduce the unnecessary reads for uncorrected kv item, we use tag, which is a compressed version or mathematical transform version of key. While using tag instead of key also bring some challenge for cuckoo hash. When hash confliction occurs, it requests the key to compute other hash value and move the hash item to the alternative bucket. However, the key is stored only on Flash and it is too expensive to reading it from Flash. Even worse, it is iteration process of solving the cuckoo hash confliction, which brings many times of reading Flash only for inserting a key.

There are two approaches to solve that challenge. First approach is that store the other hash value of key as the tag. Second approach is that compute the other hash value using the know hash value and tag instead of key.

* Storing the other hash value of key as the tag

This approach stores the other hash value as the tag in the hash item. In other word, that approach stores the alternative bucket index as tag. For example, a key x has two hash value h1(x) and h2(x) based on two hash function h1() and h2(). If using bucket h1(x) to store the hash item, the tag of that hash item is h2(x). If using bucket h2(x) to store the hash item, the tag of that hash item is h1(x).The tag has two purposes. Firstly, it can read the unnecessary Flash reading for uncorrected kv item. Secondly, it is the alternative bucket index when hash confliction occurs.

* Computing the other hash value using the know hash value and tag

The computation of two candidate buckets b1 and b2 for key x by

That approach uses the known hash value of key and the hash value of tag to compute the other unknown hash value of key.

Those two approaches can solve the challenge that computing the alternative bucket index must require know the key. However, both approaches mentioned above only have two hash functions. shows the relationship between the hash table space utilization and the number of hash function and cells per hash bucket. From the table, we can know that using 4-way set associative hash table with 2 hash function can obtain 93% space utilization.



Fig. The relationship between the hash table space utilization and the number of hash function and cells per hash bucket.

* Cockoo Hash table structure

shows the Cockoo Hash table structure we used in our KV system.

****

Fig. Cuckoo Hash table structure

* **Insert Process in Cuckoo Hash table**

****

Fig. 49 Cuckoo path

shows process of inserting a hash item. We use the approach proposed in. Firstly find the Cuckoo path and then move the Hash item backward along the cuckoo path. These process can guarantee the failure consistent.

## Flash physical space management (FTL) sub-system

### Sub-system Design

### Module Design

|  |  |
| --- | --- |
| Module name | Duty |
| GC |  |
|  |  |
|  |  |
|  |  |

The process of the Flash garbage collection is that copy the valid item in the old block to pages in the new block and erases the old block. From the efficiency of space collection, the block which contains fewest valid data should be applied garbage collection. From the wear-out balance collection, the block which has fewest erasure times should be applied garbage collection. So the tradeoff between the efficiency of space collection and wear-out balance is challenge task. In our KV system, we divided the garbage collection into three parts: choosing the block to be erased, choosing the block to be written, and choosing the block to be replaced.

Fig. Data structure of the Block manage.

* **Choosing the block to be erased**

The delete and update requests bring invalid kv items in Flash. Because the Flash cannot be in-place update, the invalid mask is stored in SCM. The space of the invalid kv items will be collected when the Flash garbage collection is applied. When the Flash garbage collection will be applied and which Flash block will be chosen to be erased follows the approach listed below.

There are two threshold for deciding the time when the Flash garbage collection begin: Optional garbage collection threshold value and Constrained garbage collection threshold value.

When the number of the free block is less than the Optional garbage collection threshold value, the Delay garbage collection will be applied. When process of the delete and update requests is complete, we will check the Flash block valid ratio of the corresponding block which contains the old kv item. If the valid ratio is less than the Collection threshold, the garbage collection will be applied to that block.

When the number of the free block is less than the Constrained garbage collection threshold value, the Immediate garbage collection will be applied. We will compare the valid ratio with the Collection threshold block by block and record the minimum ratio block position until find a block whose valid ratio is less than the Collection threshold value and then erase that block. The start comparison position is the next position where the last Immediate garbage collection ends. If the number of the comparison is larger than the Maximum Immediate garbage collection threshold value, the minimum valid ratio block will be erased.

* **Choosing the block to be written**

There are multiple free block lists. Each free block list presents different erased times. The erased block from the garbage collection mentioned above is listed to the corresponding free block list according to its erase times. We choose the block of fewest erase time as the block which will written.

The number of the free block lists is limited and each list represents a range of the erased times. For example, there are 5 free block lists. The first list represents the erase times less 100, the second list represents the erase times from 100 to 199, the third list represents the erase times from 200 to 299, the fourth list represents the erase times from 300 to 499, the fifth list represents the erase times over 500. The representative range of each list increases with the used time of the KV system because the average erase time of Flash block increase with the used time of the KV system. Such as, at beginning, the first lists represents the erase times less 100 and sometime later, the first list represents the erase times less 200.

* **Choosing the block to replaced**

Some Flash blocks contain unchanged data, so the valid ratio of those block are always higher than the Collection threshold value. The garbage collection is always not applied to those block and the erase times is much less than the average erase times. In order to balance the wear-out, at the Immediate garbage collection process, we will find the block which erase time is much less than the average erase times. We will copy the valid data to block which listed in the most used time list, such as the fifth list in the above example.

Fig. Data structure of the GC request queue

Fig. Data structure of the GC suspend queue (read/write)

### Work flow in the sub-system

* **Garbage collection**

shows the data layout of the Hot data, more detail has been analyised above. The Evict process needs read/write Flash operation, bacuase the read/write Flash Functions are asynchronous, the queue must be used. The work flow is shown in . When the kv thread which belongs to the request sub-system processes the SET kv request, it can find the free space of the Hot data circular log is less than the threshold. When the free space of the Hot data circular log is less than the threshold, the kv thread puts an evict request in the evict request queue. When the evict thread which belongs to the Hot data sub-system finds there are new evict request in the evict request queue, it will process the evict request. Because the read/write Flash operation is asynchronous, when the evict thread needs read or write Flash, it adds the read or write Flash request to the read Flash request queue or write Flash request queue respectively and add an evict suspend item which records the state of the suspend evict request in the evict suspend queue, then it will process another evict request. When the read/write Flash thread which belongs to the Flash Concurrent IO management sub-system finds read/write Flash request in the read Flash request queue or write Flash request queue, it will process the read/write Flash request. When the read/write Flash thread finsihs read/write Flash request, it will add the the read or write Flash finish to the read Flash finish queue or write Flash finish queue respectively. When the evict thread finds a read or write Flash finish item in the read Flash finish queue or write Flash finish queue, it will finds the corresponding suspend evict request in the evict suspend queue and continue the process of the suspend evict request.

* Lock version
* Synchronous Flash read/write (synchronous Flash read/write is not used in our KV system with MTD Flash )

Fig. Work Flow of GC with lock

* Asynchronous Flash read/write (asynchronous Flash read/write is used in our KV system with MTD Flash )

(a)

(b)

(c)

* TSX version

### Interface in the sub-system

|  |  |
| --- | --- |
| Interfance specification |  |
| Function |  |
| Input |  |
| Output |  |
| Return |  |
| Relative data |  |
| Calling module |  |
| Called module |  |
| Reamark |  |

## Flash Concurrent IO management sub-system

### Sub-system Design

### Module Design

|  |  |
| --- | --- |
| Module name | Duty |
| Read Flash request queue |  |
| Read Flash finish queue |  |
| Write Flash request queue |  |
| Write Flash finish queue |  |

* **Read Flash request queue**
* **Read Flash finshe queue**
* **Write Flash request queue**
* **Write Flash finish queue**

### Work flow in the sub-system

### Interface in the sub-system

## Network

The Network function work flow is the same as that in MICA.