OpenFlash: A Flash system Simulator for Performance and Energy Computation

Flash Subsystem Layer User Guide

Pierre Olivier <pierre.olivier@univ-brest.fr>

May 12, 2014

Contents

Introduction

OpenFlash is a tool for simulating NAND flash-based storage systems. It is dedicated to performance and energy computation and estimation, flash control systems comparison, prototyping, testing and validation. It is designed to cover the wide range of today's NAND applications, from embedded (mostly single chip) storage to SSD-based multi-chip, multi-channel and highly parallel storage.

This document describes the flash subsystem layer of OpenFlash. This layer corresponds to the simulation of the flash components, i.e. the NAND storage architecture without the flash control layer. The models implemented in this OpenFlash layer can be used to describe the structure, simulate the behavior and compute the performance and power consumption of flash subsystems. The NAND subsystem module of OpenFlash is decoupled from the whole tool and can be used as a standalone C++ API. The usage of this API is as follows:

1. The user input parameters to describe the desired flash subsystem ;
2. During the simulation, the described system is fed with an I/O trace which is a list of flash commands ;
3. Performance and power consumption values are computed during the simulation and presented as output at the end of the process.

The latest version of that sub-module can be found at the following address: *http://stockage.univ-brest.fr/~polivier/OpenFlash/OpenFlash.tar.gz*. The Doxygen generated technical documentation is available here:*http://stockage.univ-brest.fr/~polivier/sample_doc*

Alternatively the technical documentation can be generated from the sources and viewed locally in a web browser by typing `make view_doc` in a terminal at the root of the source tree.

This document is divided into three chapters. In the first chapter, we present the way we perceive the various characteristics of a NAND subsystem described and simulated in OpenFlash. Understanding these concepts is essential for a good use of the flash subsystem API. This first chapter describes the possibilities offered to the user wanting to describe a flash subsystem structure, functions / operations, performance and power consumption characteristics.

The second chapter is the core of this document. It explains how to describe a flash subsystem using the API, and how to feed it with a trace to perform a simulation computing performance and power consumption.

CHAPTER

1 The Characterization of a Flash-based Subsystem

In this chapter, the way OpenFlash model a NAND flash subsystem is presented. OpenFlash models the structure, functions, performance and power consumption behaviors of the NAND subsystem. Understanding the concepts presented in this chapter is essential to efficiently use the API offered by the OpenFlash flash subsystem API.

1.1 General concepts

OpenFlash implements models to describe a flash subsystem structure and behavior, and compute its performance / power consumption during a simulation. Before a simulation, the user inputs parameter values for these models to describe the desired flash subsystem. OpenFlash implements the following models:

- The *structural model* is used to describe the internal architecture of the depicted flash subsystem ;
- The *functional model* is used to describe the flash commands supported by the subsystem, and the way these commands are processed ;
- The *performance model* is used to compute execution times taken by the different flash event occurring during the simulation, and calculated by the functional model ;
- The *power consumption model* is very similar to the performance model, but it targets energy computation.

1.2 Structural model

The structural model is used to define the architectural characteristics of the simulated flash subsystem. As OpenFlash targets system level simulation, it does not go deep into the details of the described system. As for the flash subsystem layer, the flash page is the finest level of granularity considered by the software.

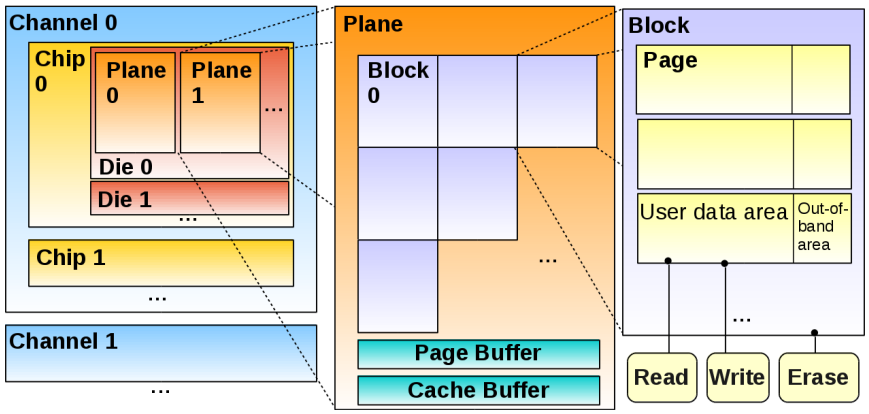

Figure 1.1: The flash subsystem structure implemented by OpenFlash

1.2.1 A flash chip

As depicted on Figure 1.1 page 5, OpenFlash models the architecture of a NAND flash chip as the organization of hierarchical elements :

- A flash *chip* is composed of one or several *dies* ;
- A *die* contains one or several *planes* ;
- A *plane* contains multiple *blocks*. A plane also contains a *page buffer* used to buffer data read / written during I/O operations. A plane can also contain a *cache buffer* whose goal will be explained in the functional model section ;
- A *block* contains multiple *pages* ;
- Finally, a *page* is divided between the *user data area* and the *Out-Of-Band* (OOB) area.

This structural flash chip model can be illustrated with some real world values. For example, let's take a look at the Micron NAND data-sheet available at the following URL:
http://download.micron.com/pdf/datasheets/flash/nand/4gb_nand_m40a.pdf .

In this example, the page size is 2048 bytes (user data area) + 64 bytes (OOB). One block contains 64 pages, and one plane contains 2048 blocks. According to the model, one chip can contains 1, 2 or 4 stacked dies. Finally, Each die contains two planes.

As some simple embedded NAND storage subsystems only include one chip, more complex storage subsystems exist. OpenFlash is able to simulate multi-chips, multi-channels storage subsystems like those found in SSDs.

1.2.2 Multi-chips, Multi-channels SSD storage subsystem

SSDs contains several NAND flash chips. Flash chips sharing the same I/O bus are regrouped into channels. A SSD can include multiple channels to perform true par-

allelism operations. Chips and channels organization are also depicted on Figure 1.1.

1.3 Functional model

The goal of the functional model is to depict the various operations supported by the flash subsystem. The operation implemented by OpenFlash are divided into four categories. The *legacy operations* are the basic NAND read, write and erase operations supported by all devices. *Intra-chip advanced operations* are supported by some devices and are performed inside one NAND chip. The *intra chip advanced combined operations* are a combination of several operation from the previous categories. Finally, *Multi devices* operations are performed on several chips or channels. Therefore, they are only supported in multi-chip / multi channels storage subsystems like SSDs.

The full list of NAND subsystem operations implemented by OpenFlash is as follows:

- **Legacy operations:**
  - *Legacy page read* & *write, legacy block erase* ;
- **Intra-chip advanced operations:**
  - *Copy back* or *internal data move* ;
  - *Cache read* / *write* ;
  - *Multi plane page read* & *write, multi plane block erase* ;
  - *Die interleaved page read* & *write, die interleaved block erase* ;
- **Intra-chip advanced combined operations:**
  - *Multi plane cache read* & *write* ;
  - *Die interleaved cache read* & *write* ;
  - *Multi plane copy back* ;
  - *Die interleaved copy back* ;
  - *Die interleaved multi plane page read* & *write, die interleaved multi plane block erase* ;
  - *Die interleaved multi plane cache read* & *write*
- **Multi devices operations:**
  - *Multi chips* commands ;
  - *Multi channels* commands.

In the following sections those operations are depicted in details. Back to our Micron NAND data sheet example, we can see that the described flash chip model supports of course legacy operations. In addition, its supports cache operations, copy back, and multi plane operations, as well as a combination of those advanced commands. Finally, according to the chip model, multiple dies models support die interleaved operations.

For each NAND operation implemented by the OpenFlash flash subsystem layer, we give a description of the operation and a list of the constraints applying to this operation. One obvious constraint which apply to all operation is the fact that the

addressed elements (page, block, plane, etc.) must be inside the address range of the described NAND subsystem.

1.3.1 Legacy operations

Legacy read

The legacy read command performs a read operation targeting a flash page. It is supported by all flash subsystems. There is no particular constraint for this operation, aside from the fact that the addressed page must be in the address range of the described flash subsystem. A legacy read operation reads the full page, i.e. the user data area *and* the OOB area. Separating the user data from the OOB data is the role of the control layer.

An example of legacy read is illustrated on Figure 1.1 page 5.

Legacy write

The legacy write command performs a write operation on a full page. It is supported by all flash subsystems. The following constraints apply:

1. One cannot write in a page already containing data (previously written), the page must first be erased before being re-written ;
2. Writes in a block should be sequential (i.e. starting from page 0 to the last page of the block) to avoid disturbance. This is a common problem for NAND flash.

The write operation is achieved on a full page and write the user data area and the OOB area. The content of the OOB area is generally determined by the flash control layer based on the user data written and the control layer internal data structures.

An example of legacy write is illustrated on Figure 1.1 page 5.

The presented write constraints are core constraints of NAND flash memory. They will apply to each advanced operation presented in the remainder of this document.

Legacy erase

The erase operation is achieved on a whole block: all the pages contained in the targeted block are erased and become ready to be written. Apart from the address range, no particular constraint apply.

An example of legacy erase is illustrated on Figure 1.1 page 5.

1.3.2 Intra chip advanced operations

Copy back

The copy back operation may be also referred as *internal data move*. It consists in using the page buffer to copy data from a page to another inside one plane, without

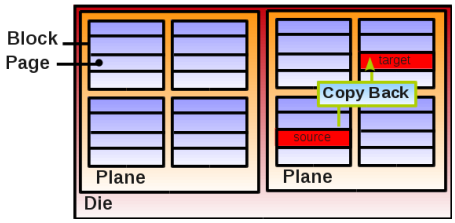

Figure 1.2: An example of a copy back operation

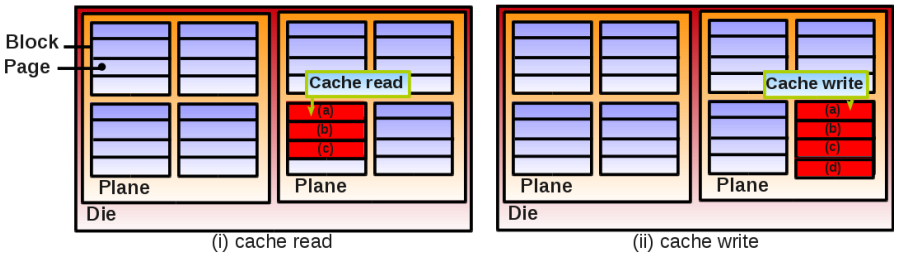

Figure 1.3: Examples of cache read (left) and cache write (right) operations

occupying the I/O bus. The following constraints apply:

1. The page index in their blocks of the source (read) page and the target (written) page must be both odd or both even ;
2. Source and target must be in the same plane of a chip ;
3. Write operation constraints apply on the target page.

An example of copy back operation is illustrated on Figure 1.2.

Cache read

Cache read is an advanced command made possible by the introduction of a cache buffer in addition to the page buffer inside each plane of the NAND chip (see Figure 1.1 page 5). The cache buffer is connected to the page buffer. It is then possible, when reading several pages from the same block in a sequential way, to pipeline I/O transfers (A) from the NAND array to the page buffer, and (B) from the cache buffer to the I/O bus. This allow to increase the performance of sequential page read operations.

The following constraints apply to the cache read operation:

1. The pages must be read in a sequential way ;
2. The entire set of pages read must not exceed the containing block boundary.

An example of cache read operation is illustrated on the left side of Figure 1.3.

Cache write

The cache write operation works the same way as the cache read command. In addition to the previously explained constraints applying to the cache read operation, the cache write command is subject to the write operation constraints: every page in the written set must be free (erased). If the cache write operation starts in a block a an offset different from 0, the pages of index 0 up to that offset must have previously been written to satisfy the constraint saying that writes in a block must always be sequential.

An example of a cache write operation is illustrated on the right side of Figure 1.3.

Multi plane read

Multi plane read operation allow reading data from pages in different planes of the same die. The transfer between NAND array and the page buffer is realized in parallel in each targeted plane. Strong constraints apply on multi plane read operations (and on multi plane operation in general):

1. Targeted pages to read must belong to different planes in the same die ;
2. The addresses of the pages to read in each plane is represented by a couple (*block index in the plane*, *page index in this block*). All the pages targeted by the multi plane read operation must have the same address couple (*block*, *page*) in the targeted planes ;
3. The multi plane operation is performed on *all* the planes of the same die.

Examples of a valid multi plane read operation, and an invalid multi plane read operations are presented on the top of Figure 1.4. On the top left side the multi plane operation is valid because the in-plane address (*block*, *page*) inside each plane of the targeted die is the same. The top right side illustrates a multi plane operation which is not feasible because the addresses inside each plane are different.

Multi plane write

This command is similar to the multi plane read operation, aside from the fact that targeted pages are written. The multi plane operation constraints apply, in addition to the standard write constraints:

1. Targeted pages to write must belong to different planes in the same die ;
2. The addresses of the pages to write in each plane is represented by a couple (*block index in the plane, page index in this block*). All the pages targeted by the multi plane write operation must have the same address couple (*block*, *page*) in the targeted planes ;
3. The multi plane operation is performed on *all* the planes of the same die ;
4. Writes must occur on free pages, and writes in each block must be sequential. For example if a multi plane write operation targets the second page of the

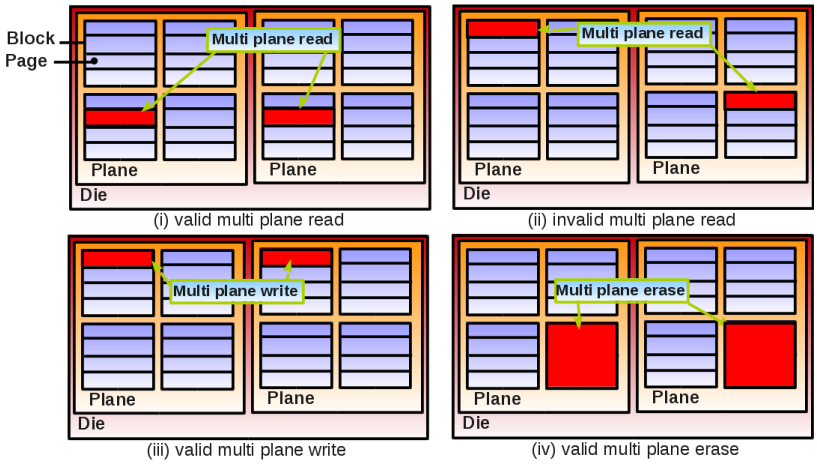

Figure 1.4: Examples of multi plane operations: a valid multi plane read (i), an invalid multi plane read (ii), a valid multi plane write (iii) and a valid multi plane erase (iv).

first block in each plane of a die, one must ensure that the first page of each block of those planes has previously been written.

On the bottom left of Figure 1.4, a multi plane write operation is illustrated.

Multi plane erase

Multi plane erase allow erasing in parallel several blocks, each one in a different plane of the same die. Multi plane constraints apply:

1. Targeted blocks must belong to different planes of the same die ;
2. The targeted block addresses (i.e. their index in each plane) must all be equal ;
3. Multi plane erase operation is performed on *all* the planes of the same die ;

On the bottom right of Figure 1.4, a multi plane erase operation is illustrated.

Die interleaved read

The die interleaved read command allows reading in parallel several pages, each one belonging to a different die of the same chip. Unlike the multi plane commands, there is no particular constraint on the page address inside each targeted die. Moreover, the set of targeted die can be a subset of the total number of dies present in the chip.

The following constraints apply on the die interleaved read command:

1. Each targeted page must belong to a different die of the same chip.

A die interleaved read operation is illustrated on Figure 1.5.

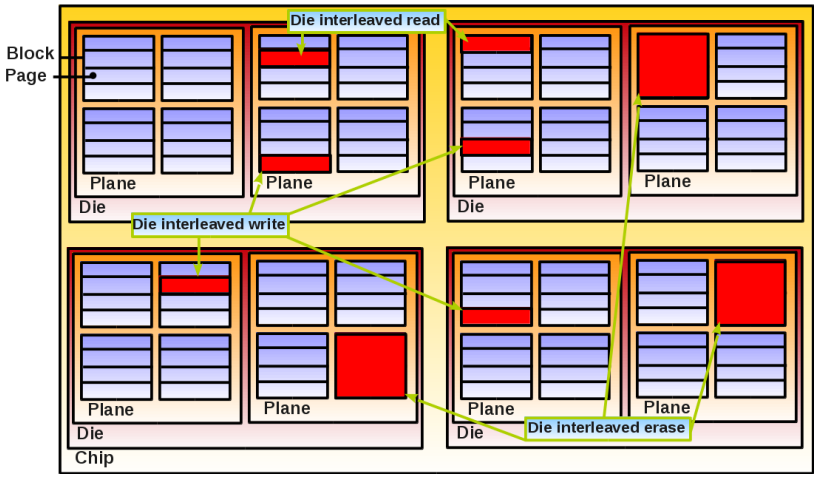

Figure 1.5: Examples of die interleaved read, write and erase operations

Die interleaved write

Die interleaved write command works in a similar way to the die interleaved read command, but performs write operation. The standard write constraints apply. Thus, die interleaved write constraints are:

1. Each targeted page must belong to a different die of the same chip ;
2. Writes must target free pages and writes in each blocks must be sequential.

An example of die interleaved write operation is presented on Figure 1.5.

Die interleaved erase

Die interleaved erase operation does not present any particular constraint, apart from the fact that each targeted block must be located in a different die of the same chip.

An example of die interleaved erase operation is presented on Figure 1.5.

1.3.3 Intra chip advanced combined operations

Those commands are a combination of the previously presented advanced commands.

Multi plane cache operations

Multi plane cache read and multi plane cache write operation allow reading / writing sets of pages sequentially in all the planes of the same die. Constraints for the multi plane operations *and* the cache read / write operation apply.

The constraint on a multi plane cache read operation are the following:

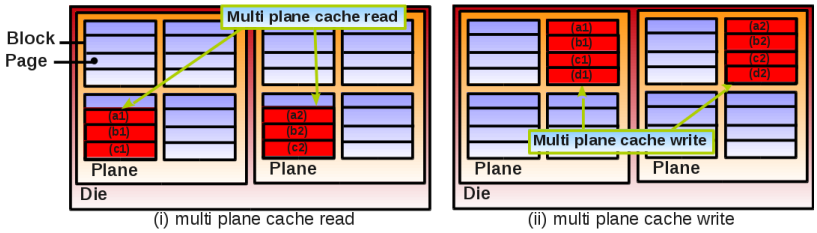

Figure 1.6: Examples of multi plane cache read (left) and multi plane cache write (right) operations

1. Each set of pages read must be in the same location in each plane of the die targeted by the operation ;
2. Each set of pages read must be in a different plane of the same die ;
3. The operation must target all the planes of the same die ;
4. Pages must be read in each plane in a sequential way ;
5. The sets must not exceed the containing block boundary ;

The multi plane cache write operation exhibits the same constraints, with the addition of the standard write constraints: each page written must be free, and writes must occur sequentially in a block.

Examples of multi plane cache read and write operations are presented on Figure 1.6.

Die interleaved cache operation

Die interleaved cache read and write operations allows performing multiple cache read or multiple cache write in different dies of the same chip. Cache read and write operation constraints apply. As compared to the multi plane cache operations, the sets read / written during die interleaved cache read / write can be at different location within the targeted dies, and of different sizes. Moreover, it is not mandatory to target *all* the dies in the chip.

The following constraints apply to the die interleaved cache read operation:

1. Each set of pages read must be in a different die of the same chip ;
2. Pages in a set must be read sequentially ;
3. The sets must not exceed the containing block boundary ;

For the die interleaved cache write command, the standard write constraints apply: pages written must be free, and writes must occur sequentially within a block.

Examples of die interleaved cache read and die interleaved cache write operations are illustrated on figure 1.7.

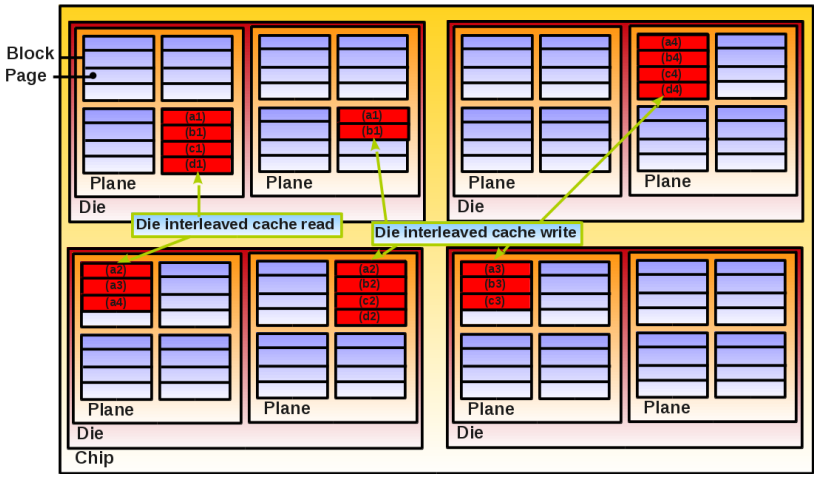

Figure 1.7: Examples of die interleaved cache read and die interleaved cache write operations

Multi plane copy back

This operation allow performing several copy back operation in each plane of the same die. The multi plane constraint apply: the addresses for source and target pages must be the same in each plane of the targeted die. Source and target index within their containing block must be both odd, or both even (copy back constraint).

To summarize the multi plane copy back constraints :

1. Each copy back operation must be performed in a different plane of the same die ;
2. Sources pages for the set of copy back executed must have the same address (*block*, *page*) within each plane ;
3. Target pages must also have the same address (*block*, *page*) within each plane ;
4. Copy back operations are executed in *all* the planes of the targeted die ;
5. The in-block index of target and source pages of each copy back operation must be both odd or both even.
6. Standard write constraints apply on the target page of each copy back operation: it must be free and write in the containing block must occur sequentially.

An example of a multi plane copy back operation is illustrated on Figure 1.8.

Die interleaved copy back

The die interleaved copy back perform several copy back operation in parallel, in different dies of the same chip. As with all the die interleaved operations, it is not

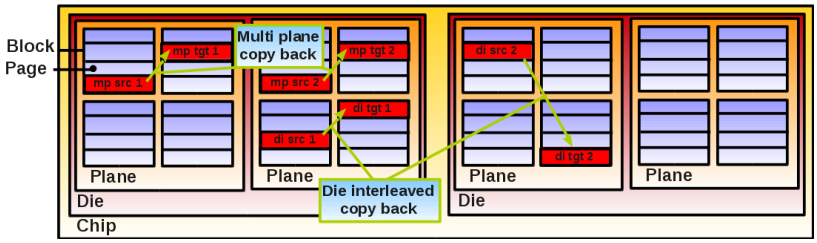

Figure 1.8: Examples of multi plane copy back and die interleaved copy back operations

mandatory to target *all* the dies in each chip. There is also no constraint on the address of the copy back operations (apart from the constraint relative to the copy back operations themselves).

The following constraint apply:

1. Each copy back operation must be in different dies of the same chip ;
2. The in-block index of target and source pages of each copy back operation must be both odd or both even.
3. Standard write constraints apply on the target page of each copy back operation: it must be free and write in the containing block must occur sequentially.

An example of a die interleaved copy back operation is illustrated on Figure 1.8.

Die interleaved multi plane read / write / erase operations

These operation are used to launch several multi plane page read / write and multiple multi plane block erase in several dies of the same chip.

Each page read / write / block erase multi plane operation must satisfy the related constraints mentioned above. Multi plane read / write / erase launched in different dies can be at various addresses.

Examples of die interleaved multi plane read / write / erase operations are illustrated on Figure 1.9.

Die interleaved multi plane cache operation

These commands allow launching several multi plane cache read / write operation in different dies of the same chip. Each multi plane cache read / write operation launched in a die must satisfy the constraints of multi plane cache read / write operations. Between different dies, multi plane cache operations can target different addresses and be of various sizes (int terms of number of pages read / written in cache mode).

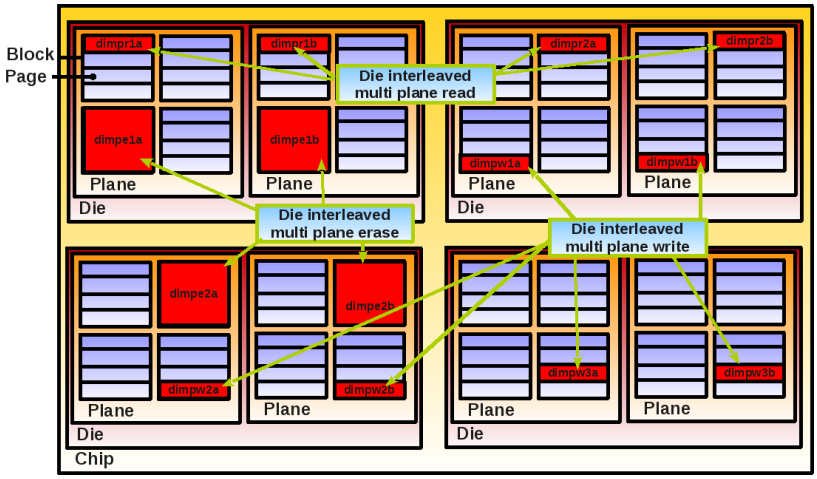

Figure 1.9: Example of die interleaved multi plane read / write / erase operations

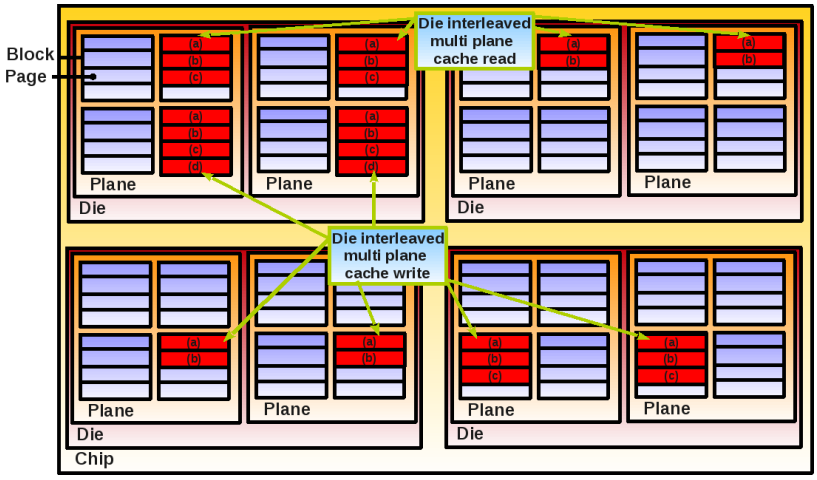

Figure 1.10: Example of die interleaved multi plane cache read and write operations

1.3.4 Multi devices operations

Multi chip

A multi chip operation allows launching in parallel several NAND commands in several chips in the same channel. Chips in a channel share the same I/O bus. Thus, command opcode, address(es), and potential data are sent on the channel in an interleaved way. The commands are executed in parallel, then potential data are dent back on the I/O bus in an interleaved way.

A multi chip operation can be a set of any of the previously presented commands.

Multi channel

Multi channel operations allow launching several NAND commands on multiple channels. This allow complex NAND subsystems to offer true parallelism. A multi channel operation can be a set of any of the previously presented commands, including multi chip operations.

1.4 Performance model

The performance model defines the way OpenFlash computes execution times for the various flash events occurring in the simulated NAND subsystem. Flash events are in fact the reception of commands by the subsystem, so the performance model is used to compute the execution times of all the commands supported by the simulated subsystem.

The performance model is a set of equations or functions. Each one is used to compute the execution time of a flash command, according to some meaningful parameters. A simple performance model for a simple NAND subsystem supporting only legacy operations can be illustrated as follows :

- $T_{LegacyRead} = 25\ \mu s$;
- $T_{LegacyWrite} = 200\ \mu s$;
- $T_{LegacyErase} = 1500\ \mu s$.

With such a performance model we can describe that a legacy page read operation takes 25 μs, a page write takes 200 μs, and a block erase 1500 μs. The same performance model can be represented as a set of C++ function as follows:

```
1  double legacyRead()
2  {
3    return 25;
4  }
5
6  double legacyWrite()
7  {
8    return 200;
9  }
10
11 double legacyErase()
```

```
12 {
13   return 1500;
14 }
```

For the rest of this document we will present some examples of performance model as C++ functions, because (1) it is the programming language in which OpenFlash is written and (2) mathematical equation can easily be represented as functions. We can describe some more complex performance models that do not return constants but compute their return values based on some parameters :

```
1  double legacyRead(int AddressedPageIndex)
2  {
3    if(AddressedPageIndex % 2 == 0)
4      return 25;
5    else
6      return 250;
7  }
8
9  double legacyWrite(int AddressedPageIndex)
10 {
11   if(AddressedPageIndex % 2 == 0)
12     return 200;
13   else
14     return 500;
15 }
16
17 double legacyErase()
18 {
19   return 1500;
20 }
```

Here we can see that the time taken to read or write a page is computed according to the address of the targeted page.

```
1
2  double copyBack(int sourceIndex, int targetIndex)
3  {
4    double res;
5
6    res = legacyRead(sourceIndex) + legacyWrite(targetIndex) - copyBackBenefit;
7
8    return res;
9  }
10
11 double dieInterleavedRead(vector<int> pagesToRead)
12 {
13   for(int i=0; i<(int)pagesToRead.size(); i++)
14 }
```

TODO: A completer. Peut etre ne presenter que des concepts theoriques ici et deplacer tout ce quiest code C++ / implementation dans la partie "using the API".

1.5 Power consumption model

TODO

CHAPTER

2 Using the Flash Subsystem API

In this chapter we explain how to use the NAND subsystem API to describe and simulate a NAND flash based storage subsystem.

2.1 Basics

2.1.1 The Address object

To address a specific object (page, block, plane, etc.) of the simulated flash subsystem, the Address object is used. Addressing is essential of course when sending commands, but also for other functions of the simulator such as gathering statistics on a given subset of the subsystem after a simulation. In the flash subsystem, an address is from a theoretical point of view a tuple of 6 elements represented as follows:

$$(a, b, c, d, e, f)$$

The members of this tuple are:

- a is the channel index in the whole flash subsystem ;
- b is the chip index in the channel ;
- c is the die index in the chip ;
- d is the plane index in the die ;
- e is the block index in the plane ;
- f is the page index in the block.

For example, consider a flash system with 2 channels, 4 chips per channel, 2 dies per chip, 2 planes per die, 2048 blocks per plane and 64 pages per block. If we want to address, for instance during a legacy read operation, the seventh page (page 6) in the 100th block (block 99) of the second plane (plane 1) of the first die (die 0) of the third chip (chip 2) of the first channel (channel 0) of such a subsystem, the tuple locating this exact page will be: $(0,2,0,1,99,6)$.

Note that for certain operations, the lowest levels members of the address are not significant. For example, if one want to erase the block containing the page

of our previous example, the page member of the address is insignificant because we target the whole block and not a particular page in this block. Therefore, any number can be passed for the page member of the address, OpenFlash will not perform any address range check on that member.

At the code level, an address object is simply created with the Address class:

```
Address a1(0,2,0,1,99,6);
Address *a2 = new Address(0,2,0,1,99,6);
/* do some things with a1 and *a2 */
delete a2;
```

After instantiating, one can retrieve the members of the address using getters functions, and modify them with setters:

```
int channel = a1.getChannel(); // Channel is 0
a1.setDie(1);                  // Die member of a1 is now equal to 1
```

Comparison functions (== and !=) are also available:

```
Address a3(1,2,0,1,99,6);
Address a4(0,1,0,1,1501,60);

bool res = (a3 == a4); // res is false
bool res = (a3 != a4); // res is true
```

For more information on the Address object, see the technical documentation here: *http://stockage.univ-brest.fr/ polivier/sample_doc/classAddress.html*

2.1.2 The Error Manager

That entity is used to centralize the error management in OpenFlash. Each simulation program realized with OpenFlash must have one instance of the error manager. It is declared as `extern` in one of OpenFlash headers (common.h), and must be defined as a global variable in the main file of the simulation program. The way the error manager is defined is explained hereafter.

The error manager is called from various points in the OpenFlash sources. When an error or a problem arises during a simulation, the error manager is called an the simulation may be stopped according to the gravity of the error. OpenFlash error manager defines two types of problems:

1. *Warnings* are not fatal to the functioning of the simulation, and the simulation may continue after a warning. The role of the warning is to indicate that something unanticipated may have happen. It will in practice take the form of a warning message on the standard / error output ;
2. *Errors* are serious and indicate that something went wrong during the simulation. In practice the simulation should not continue after an error.

Example of errors are writing in a page already containing data, out-of-range addressing during commands sending, etc. An example of warning is the fact that a page write operation does not occur sequentially in a block. Note that for now, most of the problems that may occur in OpenFlash are *errors*.

At the current stage, the error manager takes two parameters:

1. The fact that a simulation should stop on a *warning* ;
2. The fact that a simulation should stop on an *error*.

It is recommended to set the error manager not to stop the simulation on a warning, and to stop the simulation on an error. Setting that a simulation should *not* stop on an error is reserved for unit testing / debugging, and should never be set this way during a real simulation.

In practice, the simulator *stops* the simulation by executing the following instruction: `assert(0)`. This allows keeping the program context in some debuggers such as gdb, for debugging purposes.

As stated earlier, the error manager object is declared as '`extern ErrorManager em;`' in one of OpenFlash headers. It must be defined at global scope in the main file of the simulation program, as follows:

```
1 ErrorManager em(false, true, true);
```

The constructor of the ErrorManager object takes 3 parameters which are all booleans:

- The first parameter indicates if the simulation should be stopped on an error ;
- The second parameter indicates if the simulation should be stopped on a warning ;
- The last parameter indicates if a warning should be throwed when a non sequential write occurs within a block.

The error manager will be used by the flash subsystem layer of OpenFlash, but the user may also use it in his own code. To throw a warning or an error, proceed as follows:

```
1 em.warning("This is a warning");
2 em.error("This is an error");
```

As the error manager is declared at global scope, one should be able to call it from anywhere in the program. The error / warning message will be printed on the error output and the program will continue or be stopped according to the parameters of the error manager.

For more information concerning the ErrorManager class see the related technical documentation at the following URL: *http://stockage.univ-brest.fr/ polivier/sample_doc/classErrorManager.html*.

2.1.3 Basic simulation template & compiling

Defining and executing a simulation require to create one or several C++ files, one of these files containing a `main` function. These files must include the OpenFlash sources which present some dependencies, in particular with the libconfig library. The libconfig library is used by OpenFlash to provide configuration file support, its use is explained in a following section of this document.

Here we present how to build a simulation from a single main C++ file. The file must be present at the root of OpenFlash source (with the others OpenFlash source files). This main file must include the `FlashSystem.hpp` header, a `main` function, and the definition of the error manager as explained earlier. A basic template for a simulation program should then be as follows:

```
1  #include <iostream> // some standard includes ...
2  #include <cstdlib>  // ... for practical reasons
3
4  #include "FlashSystem.hpp" // OpenFlash flash subsystem header
5
6  ErrorManager em(false, true, true); // error manager
7
8  int main(int argc, char **argv)   // Main function
9  {
10   return EXIT_SUCCESS;
11 }
```

Consider this file is named MySimulation.cpp. To be able to compile it edit the OpenFlash Makefile as follows:

```
15 # Executable targets (cpp with main functions):
16 UNITTESTS=UnitTests.cpp
17 EXAMPLE=Example.cpp
18 MYSIMULATION=MySimulation.cpp    # Add this line ...
19 # Full list
20 FULLSRCS=$(SRCS) $(UNITTESTS) $(MYSIMULATION) # ... and the variable here
21 FULLHEADERS=$(SRCS:.cpp=.hpp) $(UNITTESTS:.cpp=.hpp)
22
23 # Then add at the end of the following line a target with the name of your
       choice:
24 all: libconfig UnitTest++ UnitTests Example MySimulation
```

Then add the following target:

```
50 # My simulation:
51 MySimulation: $(SRCS:.cpp=.o) $(MYSIMULATION:.cpp=.o)
52    $(CXX) $(CXXFLAGS) $(INCLUDES) $^ -o $@ $(LDFLAGS)
```

Typing `make` in a shell at the root of OpenFlash sources should then trigger the compiling of your (empty) simulation file, and all its dependencies. You should observe the creation of a `MySimulation` executable at the end of the process. Note that for now OpenFlash makefile performs static compiling with the libraries. We are aware that static compiling is not considered as a good programming practice. However, the current goal is to distribute OpenFlash in a self sufficient archive, and we want OpenFlash not to be subject to bugs due to library code changes.

2.2 Defining the structure and functions of the simulated flash subsystem

The simulated flash subsystem is represented by the C++ class FlashSystem, which is the core of the OpenFlash flash subsystem layer. You can find plenty of information about the FlashSystem class in this document. This information can be com-

pleted by the technical documentation concerning the FlashSystem class. It can be found at the following URL: *http://stockage.univ-brest.fr/ polivier/sample_doc/classFlashSystem.html*.

The structural parameters for the simulated flash subsystem are defined when instantiating the FlashSystem object. They are:

1. The number of pages per block ;
2. The number of blocks per plane ;
3. The number of planes per die ;
4. The number of dies per chip ;
5. The number of chips per channel ;
6. The total number of channels in the flash subsystem.

These values are passed as parameters, in reverse order, to the FlashSystem constructor. The following example creates a FlashSystem object with 2 channels, 4 chips per channels, 2 dies per chip, 2 planes per die, 2048 blocks per plane and 64 pages per block:

```
FlashSystem f1(2,4,2,2,2048,64);
```

To describe a FlashSystem with just one chip having the same structure as above:

```
FlashSystem f2(1,1,2,2,2048,64);
```

Once the FlashSystem object is instantiated, one can get the structural parameters using "getters" functions, such as:

```
int pagesPerBlock = f1.getPagesPerBlockNum(); // Number of pages per block, here
    it is 64
int channels = f1.getChannelNum(); // Number of channels, here 2

int totalPageNum = f1.getPageNum(); // Total number of pages in the FlashSystem,
    here it is 2*4*2*2*2048*64 = 4194304
```

When a FlashSystem object is instantiated, it supports by default all the flash commands (described in the previous chapter). One can deactivate the support of some commands as follows:

```
// Indicate that the flash subsystem does not support die interleaved multi
    plane commands:

f1.setSupportedCommands(DIE_INTERLEAVED_MULTI_PLANE_READ, false);
f1.setSupportedCommands(DIE_INTERLEAVED_MULTI_PLANE_WRITE, false);
f1.setSupportedCommands(DIE_INTERLEAVED_MULTI_PLANE_ERASE, false);
f1.setSupportedCommands(DIE_INTERLEAVED_MULTI_PLANE_CACHE_READ, false);
f1.setSupportedCommands(DIE_INTERLEAVED_MULTI_PLANE_CACHE_WRITE, false);

// One can also reactivate the support for some commands:
f1.setSupportedCommands(DIE_INTERLEAVED_MULTI_PLANE_CACHE_WRITE, true);
```

The name of the command in upper case is define by the OpenFlash flash command model which will be depicted in details in the next section. Deactivating the support of some commands will result in an error thrown if one of these command is issued to the simulated system during the simulation.

2.3 Sending commands to the flash subsystem

There are two ways to send a command to the flash subsystem during the simulation:

Method 1 By using one of the member function of the FlashSystem object, according to the command one want to send. For example, the FlashSystem class provides a FlashSystem::readPage() method to perform a legacy page read operation. There is one function for each of the supported commands.

Method 2 By using the FlashSystem::receiveCommand() method, taking a *FlashCommand* object as parameter. This object describes the command sent to the flash subsystem.

In fact, the method 1 above is just an encapsulation which internally creates a *FlashCommand* object according to the operation and call the `FlashSystem::receiveCommand()` method. The FlashCommand class implements the OpenFlash flash command model and is used to describe a given flash operation ready to be sent to the flash subsystem. The OpenFlash flash command model is detailed in the next sections.

All commands except multi chips and multi channels commands can be sent to the FlashSystem object through method 1. It is easier because one does not have to bother with instantiating a FlashCommand object. Nevertheless, multi chips and multi channels commands consists both in a set commands sent on multiple chips / multi ple channels. These commands can be of *any* type[1]. Therefore OpenFlash uses arrays of generic types to describe the sets of commands constituting multi chips / channels commands. This generic type is the FlashCommand type.

TODO Explain the flash command model.

In the following sections we explain with examples how to send all types of commands using method 1 (member functions). Next, after having depicted the OpenFlash flash command model, when explain how to use method 2 (`FlashSystem::receiveCommand()` function).

2.3.1 Method 1 : FlashSystem member functions

In this section we describe all the `FlashSystem` object member functions used to send flash commands through method 1. We give for each function its prototype, explanations on the parameters and return values. We also list all the error that may lead the command to fail. Note that although all these member functions return -1 in case of error, if the error manager is configured to stop the program in case of error the program will be killed before the function can return.

[1] Note that a multi chip command cannot contain a multi channel / multi chip command, and that a multi channel command cannot contain itself a multi channel command

Sending legacy operations

Legacy operations are page read, page write, and block erase operations. Sending such operations through method 1 consist in calling the following member functions:

- `FlashSystem::readPage(Address a)`
  - Parameter `a` is the address of the page to read ;
  - This function returns 0 on success, 1 if the page read is empty (which can be considered as a success or a failure according to the situation), and -1 on error. For now the only error that can occur when sending a legacy read command is the fact that one of the members of the address is out of range in the flash system: for example targeting a page in the block 4000 in a flash system with 2048 blocks per plane.
- `FlashSystem::writePage(Address a)`
  - Parameter `a` is the address of the page to read ;
  - This function returns 0 on success, -1 on error. An error can be due to various situations:
    - Out of range addressing ;
    - The targeted page already contains data.
- `FlashSystem::eraseBlock(Address a)`
  - The parameter `a` is the address of the block to erase (the page member of this address is not significant) ;
  - This function returns 0 on success and -1 on error. An error is due to out of range addressing.

Below are some examples of sending legacy operations through method 1.

```
1 int res;
2 FlashSystem f(2,4,2,2,2048,64);       /* declare simulated system */
3
4 res = f.readPage(Address(1,3,0,0,33,60));          /* read a page */
5 res = f.writePage(Address(1,3,0,0,0,0));           /* write a page */
6 res = f.eraseBlock(Address(0,2,1,1,0,0));          /* erase a block */
```

Sending cache and copy back operations

These operation are sent through the following methods:

- `FlashSystem::cacheRead(Address startPage, int nbPagesToRead)`
  - `startPage` is an address targeting the first page to read ;
  - `nbPagesToRead` is the number of pages to read sequentially, starting from `startPage` ;
  - This function returns 0 on success, a positive integer equal to the number of empty pages read if any. It returns -1 on error. An error can be due to several situations:
    - Out of range addressing for `firstPage` ;
    - The set of pages to read crosses the containing block boundaries ;
    - `nbPagesToRead` is equal to 0.

- `FlashSystem::cacheWrite(Address startPage, int nbPagesToWrite)`
  - This function works the same way as `cacheRead`. The return value is a bit different: it is 0 on success and -1 on error. Errors can be due to the same situations as with `cacheRead`. Another cause of error for `cacheWrite` is the fact that one of the written pages already contains data.
- `FlashSystem::copyBack(Address source, Address target)`
  - `source` and `target` are the addresses of the source page (read) and the target page (written) of the copy back operation.
  - This function returns 0 on success, -1 on error. An error is due to the following situations:
    - Out of range addressing for `source` and / or `target` ;
    - `source` and `target` are not in the same plane ;
    - `source` and `target` have the same address ;
    - The page index in the containing block for `source` and `target` are not both odd or both even.

Below is an example of cache read and write operations, as well as a copy back operation example.

```
1  int res;
2  FlashSystem f(2,4,2,2,2048,64);       /* declare simulated system */
3
4  /* read the entire block 0 :*/
5  res = f.cacheRead(Address(0,0,0,0,0,0), 64);
6
7  /* write the first half of block 3 :*/
8  res = f.cacheWrite(Address(0,0,0,0,3,0), 32);
9
10 /* Copy back from the first page of block 0 to the page 32 of block 3 :*/
11 res = f.copyBack(Address(0,0,0,0,0,0), Address(0,0,0,0,3,32));
```

Sending multi plane operations

As presented earlier, multi plane operation have one constraint saying that the read / write / erase operation must be performed on *all* the planes of the same die. Moreover all these sub-operations must be performed at the same location within the targeted planes. Therefore, to describe a multi-plane operation in OpenFlash, one need only to specify the address of *one* page (for multi plane read & write) or *one* block (multi plane erase) and the tool will automatically apply this operation on all the planes of the die containing the targeted page / block.

Sending multi plane operation is done through the following functions:

- `FlashSystem::multiPlaneRead(Address a)`
  - `a` is the address of one page to read, the multi plane operation will automatically be applied on all the planes of the targeted die ;
  - This function return 0 on success, a positive integer representing the number of empty pages read if any, and -1 on error. An error may be due to out of range addressing.

- `FlashSystem::multiPlaneWrite(Address a)`
  - Works in a similar way to the `multiPlaneRead` function ;
  - Returns 0 on success, -1 on error. An error may be due to out of range addressing, or to the fact that one of the written pages already contains data.
- `FlashSystem::multiPlaneErase(Address a)`
  - Returns 0 on success, -1 on error (out of range addressing).

Below are some examples of multi plane read, write and erase operations.

```
1  int res;
2  FlashSystem f(2,4,2,2,2048,64);       /* declare simulated system */
3
4  /* multi plane read in (1,3,1,0,124,12) and (1,3,1,1,124,12) */
5  res = f.multiPlaneRead(Address(1,3,1,1,124,12));
6
7  /* multi plane write in (0,2,0,0,12,4) and (0,2,0,1,12,4) */
8  res = f.multiPlaneWrite(Address(0,2,0,0,12,4));
9
10 /* multi plane erase in (0,0,0,0,122,X) and (0,0,0,1,122,X) */
11 res = f.multiPlaneErase(Address(0,0,0,0,122,0));
```

Note that a warning is thrown when sending a multi plane operation in a flash subsystem with only one plane per die.

Sending die interleaved operations

Die interleaved operations constraints are lighter than multi plane ones:

1. Die interleaved operations can concern only a subset of all the dies in a chip ;
2. Targeted pages / blocks of one die interleaved operation may be in different locations within the concerned die.

Therefore, to describe a die interleaved operation in OpenFlash, one must use arrays. OpenFlash makes an intensive use of C++ standard `vector` type. The parameters of OpenFlash functions used to send die interleaved operations takes vectors of addresses as parameters. Each member of the vector is one page to read / write, or one block to erase. Each member must be in a different die of the same chip.

Sending die interleaved operations is done through the use of the following functions:

- `FlashSystem::dieInterleavedRead(vector<Address> addresses)`
  - `addresses` is an array of Address objects, each one targeting a page to read in a different die of the same chip ;
  - This function returns 0 on success, a positive integer equal to the number of empty pages read if any, and -1 on error. An error may be due to the following situations:
    - Out of range addressing for one of the members of `addresses` ;

    - The members of `addresses` are not in different dies of the same chip.
- `FlashSystem::dieInterleavedWrite(vector<Address> addresses)`
  - Works the same way as the die interleaved read function ;
  - Returns 0 on success, -1 on error. Errors may occur in the same situations as with the die interleaved read function. An error is also returned when one of the written pages already contains data.
- `FlashSystem::dieInterleavedErase(vector<Address> addresses)`
  - Works the same way as the die interleaved read / write functions ;
  - Return 0 on success, -1 on error. An error is due to out of range addressing.

Below are some examples of die interleaved read / write / erase operations.

```
1  int res;
2  FlashSystem f(2,4,2,2,2048,64);       /* declare simulated system */
3
4  /* die interleaved read on (1,2,0,1,339,12) and (1,2,1,1,339,12) :*/
5  vector<Address> addresses_read;
6  addresses_read.push_back(Address(1,2,0,1,339,12));
7  addresses_read.push_back(Address(1,2,1,1,339,12));
8  res = f.dieInterleavedRead(addresses_read);
9
10 /* die interleaved write on (0,1,0,1,145,60) and (0,1,1,1,229,12) :*/
11 vector<Address> addresses_write;
12 addresses_write.push_back(Address(0,1,0,1,145,60));
13 addresses_write.push_back(Address(0,1,1,1,229,12));
14 res = f.dieInterleavedWrite(addresses_write);
15
16 /* die interleaved erase on (1,0,0,0,222,X) and (1,0,1,1,444,X) :*/
17 vector<Address> addresses_erase;
18 addresses_erase.push_back(Address(1,0,0,0,222,0));
19 addresses_erase.push_back(Address(1,0,1,1,444,X));
20 res = f.dieInterleavedErase(addresses_erase);
```

Note that a warning is thrown when sending a die interleaved operation in a flash subsystem containing only one die per chip.

Sending multi plane cache & copy back operations

Multi plane cache operations are described with one start page and a number of page to read / write. The described cache operation is automatically applied on all the planes of the targeted die.

Multi plane copy back operations works the same way: they are described with only one couple of addresses for the source and target pages of the operation. The copy back is the performed automatically in all the planes of the targeted die.

The functions used to send multi plane cache and copy back operations are:

- `FlashSystem::multiPlaneCacheRead(Address startPage, int nbPagesToRead)`
  - `startPage` is the address of the first page to read in *one* of the targeted planes ;

  - `nbPagesToRead` is the number of pages to read in each copy back of the multi plane copy back operation
  - The operation will automatically be performed in all the planes of the die targeted by `startPage` ;
  - This operation returns 0 on success, a positive integer equal to the number of empty pages read if any, and -1 on error. An error can occur in various situations:
    * `startPage` is out of range ;
    * `startPage` + `nbPagesToRead` crosses the containg block boundaries ;
    * `nbPagesToRead` is 0.
- `FlashSystem::multiPlaneCacheWrite(Address firstPage, int nbPagesToWrite)`
  - Works in a similar way as the multi plane cache read function ;
  - Returns 0 on success, -1 on error. In addition to the errrors previously evocated in the multi plane cache read function, the multi plane cache write function may generate an error if one of the pages written is already containing data.
- `FlashSystem::multiPlaneCopyBack(Address source, Address target)`
  - `source` and `target` are the source and target page of a copy back oepration in *one* plane. The same copy back operation will automatically be performed in all the planes of the target die.
  - Returns 0 on success, -1 on error. An error can signify that:
    * `source` or `target` is out of range ;
    * Page index within their blocks for `source` and `taret` are not both odd or both even.
    * `source` and `target` are requal or not in the same plane.

Below are some examples of multi plane copy back and cache oeprations.

```
1  int res;
2  FlashSystem f(2,4,2,2,2048,64);       /* declare simulated system */
3
4  /* Multi plane cache read on the entirety of the first block of planes
5   * (0,2,1,0,X,X) and (0,2,1,1,X,X) */
6  res = f.multiPlaneCacheRead(Address(0,2,1,0,0,0), 64);
7
8  /* Multi plane cache write on the first half of the first block of
9   * planes (0,1,0,0,X,X) and (0,1,0,1,X,X) */
10 res = f.multiPlaneCacheWrite(Address(0,1,0,0,0,0), 32);
11
12 /* Multi plane copy back from (1,1,0,0,122,12) to (1,1,0,0,444,14) and
13  * from (1,1,0,1,122,12) to (1,1,0,0,444,14) */
14 res = f.multiPlaneCopyBack(Address(1,1,0,0,122,12), Address(1,1,0,0,444,14));
```

Note that sending these operation may throw a warning when the simulated flash subsystem only contains one plane per die.

Sending die interleaved cache & copy back operations

Die interleaved cache operation must be described by 2 arrays. The first array (named `firstPages`) is a list of addresses for the first pages to read / write. The second array (names `nbPagesToRead / write`) is a list of integer representing the number of pages to read or write. Therefore, each couple `firstPages`[i], `nbPagesToRead/Write`[i] represent a cache operation that will be applied in one die. Each address in array A must the targets a different die in the same chip. `firstPages`[i] + `nbPageToRead/Write`[i] must not cross block boundaries. Each member of the couple must target the same die. There must not be 2 couples which target the same die.

Die interleaved copy back operations are also described by 2 arrays: `sources` and `targets`. Each couple `sources[i]`, `targets[i]` represent a copy back operation to perform in a die. Each member of the couple must target the same die. There must not be 2 couples which target the same die.

OpenFlash functions used to send die interleaved cache and copy back commands are the followings:

- `FlashSystem::dieInterleavedCacheRead(vector<Address> startPages, vector<int> nbPagesToRead)`
  - Each cache read operation composing the die interleaved cache read operation is represented by a couple `startPages[i]`, `nbPagesToRead[i]`. Each cache read operation must satisfy the related constraints ;
  - Returns 0 on success, a positive integer equal to the number of empty pages read if any, -1 on error. An error can signify that:
    - The two vector parameters do not have the same sizes ;
    - Targeted addresses are not in the same chip, or each cache operation is not in different dies of the same chip ;
    - Out of range addressing.
- `FlashSystem::dieInterleavedCacheWrite(vector<Address> startPages, vector<int> nbPagesToWrite)`
  - Works in a similar way than the die interleaved cache read function ;
  - Return 0 on success, -1 on error. In addition to the error causes evocated for the die interleaved read function, the die interleaved write function fails if one of the written pages already contain data.
- `FlashSystem::dieInterleavedCopyBack(vector<Address> sources, vector<Address> targets)`
  - Each copy back operation composing the die interleaved copy back operation is represented by a couple `sources[i]`, `targets[i]`. Each couple must satisfy the copy back constraints (in particular the source and target member of each couple must be in the same plane).
  - Returns 0 on success, -1 on error. Errors can be:
    - The two vector parameters do not have the same size ;
    - One of the copy back operations have source and targets that are not in the same plane ;
    - One of the copy back operations have source and targets index within

conataining block that are not both odd or both even ;
* Each couple `sources[i]`, `targets[i]` is not loacted in a different die of the same chip.

Below are some examples of die interleaved cache and copy back operations.

```
int res;
FlashSystem f(2,4,2,2,2048,64);       /* declare simulated system */

/* die interleaved cache read for the first half of block (0,1,0,1,2044,X)
 * and the second half of block (0,1,1,0,1054,X) */
vector<Address> startPagesRead;
vector<int> nbPagesToRead;
startPagesRead.push_back(Address(0,1,0,1,2044,0)); nbPagesToRead.push_back(32);
startPagesRead.push_back(Address(0,1,1,0,1054,32)); nbPagesToRead.push_back(32);
res = f.dieInterleavedCacheRead(startPagesRead, nbPagesToRead);

/* die interleaved cache write the whole block (1,0,0,0,154,X) and the
 * second half of block (1,0,1,1,12,X) */
vector<Address> startPagesWrite;
vector<int> nbPagesToWrite;
startPagesWrite.push_back(Address(1,0,0,0,154,0)); nbPagesToWrite.push_back(64);
startPagesWrite.push_back(Address(1,0,1,1,12,32)); nbPagesToWrite.push_back(32);
res = f.dieInterleavedCacheWrite(startPagesWrite, nbPagesToWrite);

/* Die interleaved copy back from (0,3,0,1,144,12) to (0,3,0,1,120,60)
 * and from (0,3,1,0,12,51) to (0,3,1,0,255,53) */
vector<Address> sources; vector<Address> targets;
sources.push_back(Address(0,3,0,1,144,12));
targets.push_back(Address(0,3,0,1,120,60));
sources.push_back(Address(0,3,1,0,12,51));
targets.push_back(Address(0,3,1,0,255,53));
res = f.dieInterleavedCopyBack(sources, targets);
```

Sending die interleaved multi plane operations

Die interleaved multi plane read operations are characterized by one array. Each member of this array is a multi plane read operation composing the whole die interleaved multi plane read operation. Each member must then target a different die in the same chip. The multi plane read operation will be applied automatically in all the planes of the targetted die.

The die interleaved multi plane write / erase operation work in a similar way than the die interleaved multi plane read. These operation are sent with the following functions:

- `FlashSystem::DieInterleavedMultiPlaneRead (vector<Address> addresses)`
  - Takes as parameter a vector of addresses, each one in a different die of the same chip. The multi plane read operation represented by each member of that vector will automatically applied in all the planes of the targetted die.

  - Returns 0 on success, a positive interger equal to the number of empty pages read if any, -1 on error. An arror may be due to the following situations:
    * Out of range addressing ;
    * Each member of `addresses` does not target a different die of the same chip ;
    * The size of `addresses` is superior to the number of dies per chip.
    *
- `FlashSystem::DieInterleavedMultiPlaneWrite (vector<Address> addresses)`
  - Works the same way as the die interleaved multi plane read function.
  - Returns 0 on success, -1 on error. In addition to the errors mentionned for the die interleaved multi plane read function, the write funtion fails when one of the written pages already contains data.
- `FlashSystem::DieInterleavedMultiPlaneErase (vector<Address> addresses)`
  - Works the same way as the die interleaved multi plane read / write functions.

Below are some examples of die interleaved multi plane read / write / erase functions.

```
1  int res;
2  FlashSystem f(2,4,2,2,2048,64);       /* declare simulated system */
3
4  /* die interleaved multi plane read targetting the following addresses :
5   * (1,2,0,0,144,62) ; (1,2,0,1,144,62) ; (1,2,1,0,12,11) ; (1,2,1,1,12,11)
6   */
7  vector<Address> addr_read;
8  addr_read.push_back(Address(1,2,0,0,144,62));
9  addr_read.push_back(Address(1,2,1,0,12,11));
10 res = f.dieInterleavedMultiPlaneRead(addr_read);
11
12 /* die interleaved multi plane write targetting the following addresses :
13  * (0,1,0,0,12,0) ; (0,1,0,1,12,0) ; (0,1,1,0,4,0) ; (0,1,1,1,4,0)
14  */
15 vector<Address> addr_write;
16 addr_write.push_back(Address(0,1,0,0,12,0));
17 addr_write.push_back(Address(0,1,1,0,4,0));
18 res = f.dieInterleavedMultiPlaneWrite(addr_write);
19
20 /* die interleaved multi plane erase targetting the following addresses :
21  * (0,3,0,0,255,X) ; (0,3,0,1,255,X) ; (0,3,1,0,4,X) ; (0,3,1,1,4,X)
22  */
23 vector<Address> addr_erase;
24 addr_erase.push_back(Address(0,3,0,0,255,0));
25 addr_erase.push_back(Address(0,3,1,0,4,X));
26 res = f.dieInterleavedMultiPlaneErase(addr_erase);
```

The die interleaved multi plane cache read and write operation are characterized by two arrays. Consider the die interleaved multi plane cache read operation. The first array `startPages[]` is a list of first pages to read. The second array

`nbPagesToRead` is a list of number of pages to read. Each couple `startPage[i]`, `nbPagesToRead[i]` describe then a multi plane cache read operation (see section concerning multi plane cache operations). The entirety of the couples describe the die interleaved multi plane cache read operation.

Die interleaved multi plane cache write works the same way. To send such types of operations, the following functions are used:

- `FlashSystem::dieInterleavedMultiPlaneCacheRead (vector<Address> firstPages, vector<int> nbPagesToRead)`
  - `firstPages[i]` and `nbPagesToRead[i]` describe the multi plane cache read operations launched in the dies ;
  - This function returns 0 on success, a positive integer equal to the number of empty pages read if any, and -1 on error. An error may signify that:
    - The size of `firstPages` and the size of `nbPagesToRead` are different, or superior to the number of dies per chip ;
    - Out of range addressing ;
    - One couple `startPages[i]`, `nbPagesToRead[i]` do not satisfy multi plane cache read constraints ;
    - Each multi plane cache read operation do not target a different die int the same chip.
- `FlashSystem::dieInterleavedMultiPlaneCacheWrite (vector<Address> firstPages, vector<int> nbPagesToWrite)`
  - This function works in a similar way than the die interleaved multi plane cache read function, with the addition that it may fail when one of the written page already contains data.

Below are some examples of die interleaved multi plane cache read / write operations.

```
1  int res;
2  FlashSystem f(2,4,2,2,2048,64);       /* declare simulated system */
3
4  /* Die interleaved multi plane cache read on the entire block 0 of
5   * (1,2,0,0,X,X), (1,2,0,1,X,X) and the entire block 1 of
6   * (1,2,1,0,X,X), (1,2,1,1,X,X) */
7  vector<Address> startPagesR;
8  vector<int> nbPagesToRead;
9  startPagesR.push_back(Address(1,2,0,0,0,0));
10 nbPagesToRead.push_back(64);
11 startPagesR.push_back(Address(1,2,1,0,0,0));
12 nbPagesToRead.push_back(64);
13 res = f.dieInterleavedMultiPlaneCacheRead(startPagesR, nbPagesToRead);
14
15 /* Die interleaved multi plane cache read on the first half of block 12 of
16  * (1,2,0,0,X,X), (1,2,0,1,X,X) and the entire block 1 of
17  * (1,2,1,0,X,X), (1,2,1,1,X,X) */
18 vector<Address> startPagesW;
19 vector<int> nbPagesToWrite;
20 startPagesW.push_back(Address(1,2,0,0,0,0));
21 nbPagesToWrite.push_back(32);
```

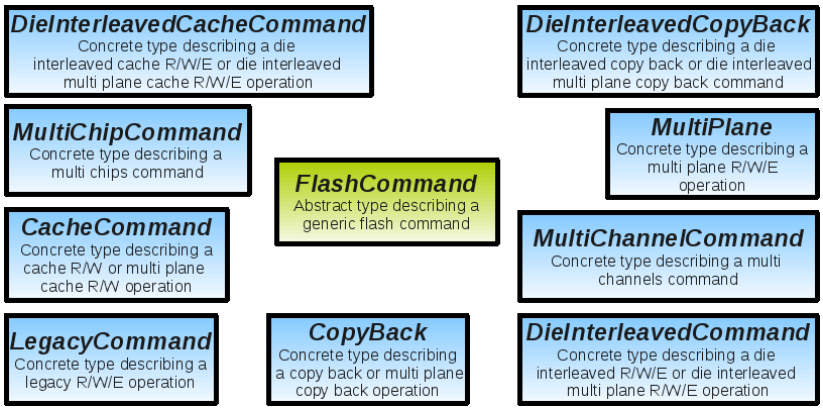

Figure 2.1: The different types (C++ classes) used by OpenFlash to describe a flash command

```
22 startPagesW.push_back(Address(1,2,1,0,0,0));
23 nbPagesToWrite.push_back(64);
24 res = f.dieInterleavedMultiPlaneCacheWrite(startPagesW, nbPagesToWrite);
```

Note that all the die interleaved multi plane functions throw a warning when launched on a simulated subsystem with only one plane per die or one die per chip.

Sending multi chips / channels commands

As explained earlier, sending multi chips or multi channels commands involves instantiating a *FlashCommand* object.

2.3.2 Method 2 : FlashSystem::receiveCommand

OpenFlash flash command model

2.4 Describing the performance and power consumption behavior

TODO.

2.5 Using the configuration file

OpenFlash uses *libconfig*[2] to provide support for a configuration file. It is a text file containing blocks of key values pairs that can be used to define the parameters of the whole simulated systems, and the parameters for the simulation itself (for example error management parameters).

[2] *http://www.hyperrealm.com/libconfig/*

2.5.1 The Param object

2.6 Gathering simulation results

TODO.