Project #4

(Report due is 11:59 PM, 01/21/2024)

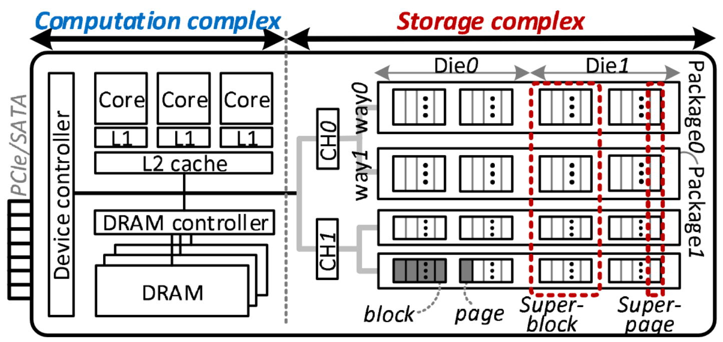
Project description:
In this project, you will explore solid-state drives (SSDs). Specifically, you will learn about the various SSD internal mechanisms (e.g., the flash translation layer (FTL)) through an SSD simulator. Further, you will implement a hybrid-level addressing mapping scheme in the SSD simulator. Whenever you face trouble or problems with this project, please contact TA. All the following information has been verified on Ubuntu 22.04.2 LTS with the Linux Kernel 5.19.0.

Submission:
- Please submit (in a report form, both PDF and Word are acceptable) the deliverables for each part in order and clearly defined. Please give a brief description of the results in your report and DO NOT SUBMIT ANY SCRIPTS.
- Please send your report as an attachment to ca2023fall@163.com. The titles of your email AND attachment should both be Student ID_Name_Proj4 (e.g., 123456789_XX_Proj4).

Late policy:
- You will be given 3 slip days (shared by all projects), which can be used to extend project deadlines, e.g., 1 project extended by 3 days or 2 projects each extended by 1 day.
- Projects are due at 23:59:59, no exceptions; 20% off per day late, 1 second late = 1 hour late = 1 day late.

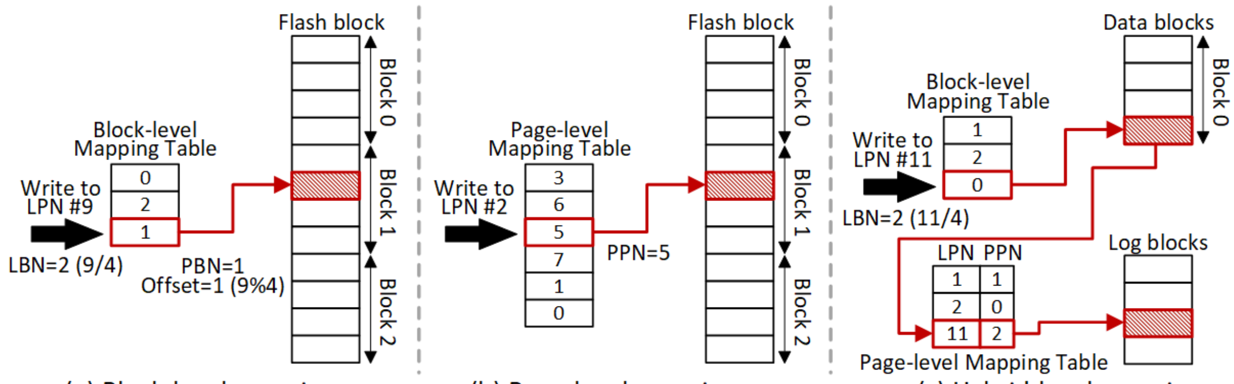
Introduction to the SSD

SSDs have become major storage media in modern computing systems. The figure below illustrates the SSD internal architecture, which consists of a computation complex and a storage complex.



Storage complex. Multiple flash packages, each containing multiple dies, are connected to the interconnection buses, referred to as *channel*. A set of flash packages across different channels can simultaneously operate. To leverage such parallelism, the flash firmware spreads a host request over multiple dies that have the same offset address but exist across different channels.

Each set of flash dies (or packages) is called a *way*. A flash die comprises hundreds to thousands of blocks, each containing hundreds of pages. A group of multiple physical pages and blocks that span over all internal channels or ways is referred to as *superpage* or *superblock*, respectively. This organization poses a high SSD internal parallelism, thus achieving superb I/O throughput.
The flash supports read and write operations at page granularity. However, due to the physical characteristics, the whole flash block must be erased before writing any page in that block, making flash unsupportable for in-place updates. To address the limitation, the SSD constructs an indirection layer called the *flash translation layer* (FTL), which maintains a mapping table to record the mapping between the host logical address (LPN) and its corresponding flash physical address (PPN). For an overwrite, the data is written to a pre-erased free block. Subsequently, FTL updates the mapping of the new flash page and invalidates the stale flash page. Depending on the mapping granularity, the FTL mapping schemes can be divided into three categories: *block-level*, *page-level*, and *hybrid-level mapping*, as illustrated in the following figure.



(a) Block-level mapping. (b) Page-level mapping. (c) Hybrid-level mapping.

Block-level mapping. In block-level mapping, the LPN is divided into a logical block number (LBN) and a page offset. A block-level mapping table uses the LBN to find the physical block (i.e., PBN) that includes the requested page, while the page offset is used to locate the page in the corresponding block. Due to such a fixed mapping, the page offsets of the logical and physical blocks should be identical, therefore, every overwrite to the same logical page incurs a block-level copy operation. That is, all the data in the corresponding block as well as the new data have to be written into another empty block, resulting in poor overwrite performance.
Page-level mapping. The page-level scheme maps the LPN directly to any PPN by maintaining a logical-physical mapping table (i.e., page-level mapping table in Figure (b)). Although this scheme improves performance with flexible address translation, the mapping table occupies huge amounts of DRAM.
Hybrid-level mapping. To achieve a balance between performance and DRAM consumption, several hybrid-level mapping schemes are proposed. Most of them use a log block mechanism to store updates. Specifically, flash blocks are divided into data blocks and log blocks. Data blocks represent ordinary storage space, while log blocks are used for storing overwrites. For translating addresses, these schemes maintain a block mapping table for the data blocks and a page mapping table for the log blocks, which consume less DRAM than page-level mapping. When an overwrite

request arrives, the new data of the request is written to the log block instead of being stored in the original location, thus avoiding the heavy block copy operation in the block-level mapping. The mapping scheme determines the specific association between data blocks and log blocks. **In this project, you will implement a typical hybrid-level mapping scheme (i.e., FAST [1]) in an SSD simulator (cf. Task Part).**

In page-level mapping scheme, when overwrites exhaust all free blocks, garbage collection (GC) is triggered to reclaim the used blocks where the SSD selects victim blocks, moves all valid pages to a new free block, and then erases it. On the other hand, in hybrid-level mapping scheme, the necessity of GC becomes obsolete as it is replaced by the operations of merging data blocks and log blocks, as well as recycling log blocks. More specifically, when log blocks are exhausted, the SSD reclaims these blocks by merging them with the associated data blocks. During such merge process, extensive pages are migrated into new blocks and multiple blocks undergoes erasure, thereby incurring latency spike.

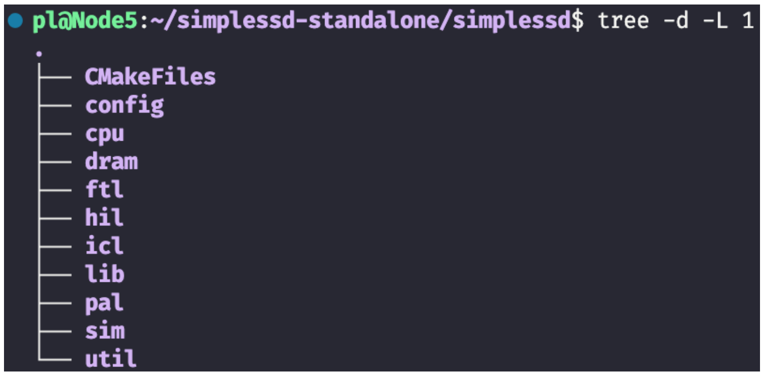
Computation complex. The computation complex consists of embedded ARM cores, internal DRAM, and device controllers. Specifically, multiple cores are allocated to flash firmware that controls all I/O services and address translations within an SSD. In addition, the device controller fetches requests from the host and sends responses back via a high-performance communication protocol (e.g., NVMe). SSDs employ embedded DRAM, which stores the FTL mapping table and I/O queues. To leverage DRAM performance, SSD also buffers the data of requests in DRAM, which hides the long latency imposed by the underlying flash.

For more detailed information about SSDs, you can refer to the paper [2].

Introduction to a Precise SSD Simulator

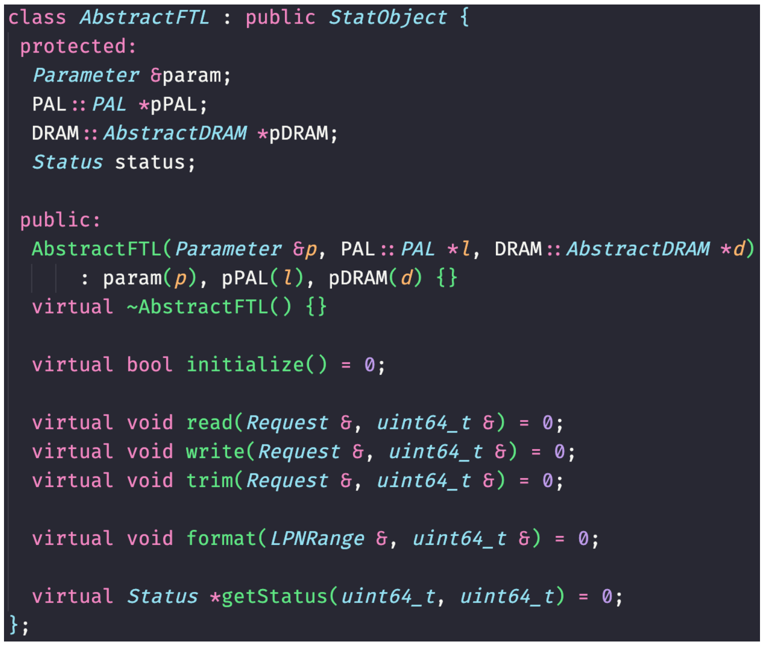
In this part, we will learn about a precise SSD simulator called Amber/SimpleSSD [2]. SimpleSSD models embedded ARM cores, DRAMs, and various flash technologies within an SSD. SimpleSSD also includes the full firmware stack of an SSD, including DRAM cache logic, flash firmware (e.g., FTL and Host Interface Layer (HIL)), and diverse standard protocols (e.g., NVMe). To meet various simulation demands, SimpleSSD supports both standalone (without the host system) and full-system (with the host system) simulation environments. Note that in this project, we will use **the standalone version of SimpleSSD** (i.e., SimpleSSD-Standalone) to simulate an SSD.

The following figure shows the code structure of SimpleSSD, where the code in each subfolder simulates the corresponding component. In this project, we mainly focus on ***ftl*** and ***pal*** components.



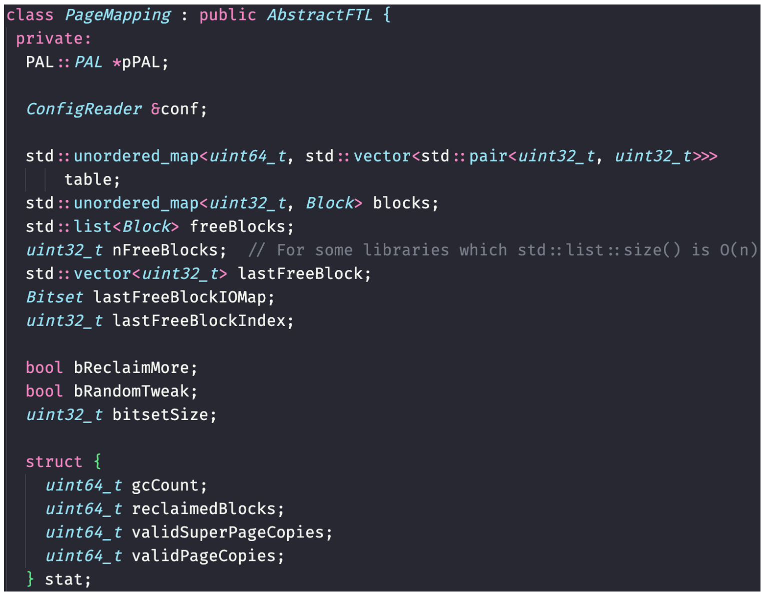
```
pl@Node5:~/simplessd-standalone/simplessd$ tree -d -L 1
.
├── CMakeFiles
├── config
├── cpu
├── dram
├── ftl
├── hil
├── icl
├── lib
├── pal
├── sim
└── util
```

FTL module in SimpleSSD. We start by introducing the FTL implementation in SimpleSSD. As shown in the below figure, *simplessd-standalone/simplessd/ftl/abstract_ftl.h* defines the FTL abstract class (i.e., *AbstractFTL*), which contains several virtual functions that need to be implemented by the FTL mapping scheme. The *initialize* function sets the necessary parameters of the mapping scheme (e.g., the threshold for triggering GC) and warms up the SSD by writing a certain amount of data. The *read* and *write* functions describe how the firmware handles read and write requests from the upper layer based on the FTL mapping scheme, respectively. The *getStatus* function returns some status metrics for the FTL. **Note that your hybrid-level mapping scheme should inherit *AbstractFTL* class and implement all virtual functions required by the abstract class.** For convenience, you **do not** need to implement specific logic for *trim*, getStatus,and *format* functions.



```
class AbstractFTL : public StatObject {
 protected:
  Parameter &param;
  PAL::PAL *pPAL;
  DRAM::AbstractDRAM *pDRAM;
  Status status;

 public:
  AbstractFTL(Parameter &p, PAL::PAL *l, DRAM::AbstractDRAM *d)
      : param(p), pPAL(l), pDRAM(d) {}
  virtual ~AbstractFTL() {}

  virtual bool initialize() = 0;

  virtual void read(Request &, uint64_t &) = 0;
  virtual void write(Request &, uint64_t &) = 0;
  virtual void trim(Request &, uint64_t &) = 0;

  virtual void format(LPNRange &, uint64_t &) = 0;

  virtual Status *getStatus(uint64_t, uint64_t) = 0;
};
```

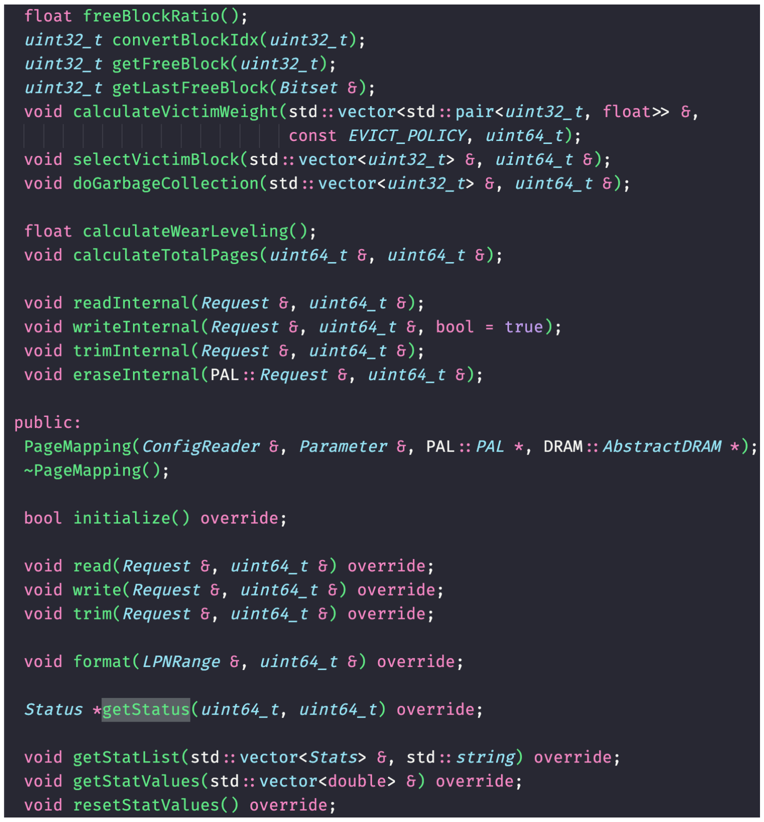
SimpleSSD provides a default superpage-level FTL mapping scheme, which can be found in *simplessd-standalone/simplessd/ftl/page_mapping.hh*:



```
class PageMapping : public AbstractFTL {
 private:
  PAL::PAL *pPAL;

  ConfigReader &conf;

  std::unordered_map<uint64_t, std::vector<std::pair<uint32_t, uint32_t>>>
      table;
  std::unordered_map<uint32_t, Block> blocks;
  std::list<Block> freeBlocks;
  uint32_t nFreeBlocks;  // For some libraries which std::list::size() is O(n)
  std::vector<uint32_t> lastFreeBlock;
  Bitset lastFreeBlockIOMap;
  uint32_t lastFreeBlockIndex;

  bool bReclaimMore;
  bool bRandomTweak;
  uint32_t bitsetSize;

  struct {
    uint64_t gcCount;
    uint64_t reclaimedBlocks;
    uint64_t validSuperPageCopies;
    uint64_t validPageCopies;
  } stat;
```

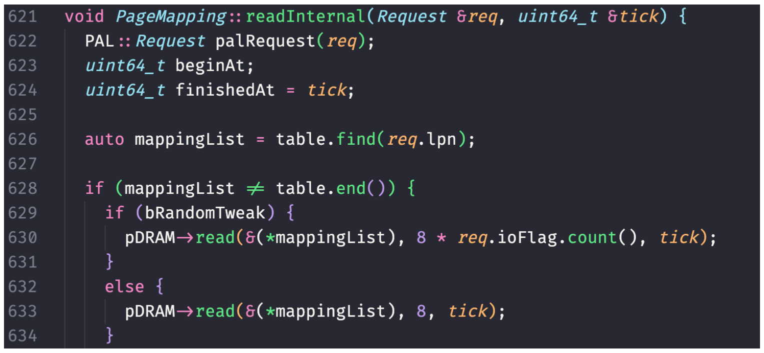
PageMapping inherits *AbstractFTL* and adds some necessary member variables. Specifically, the *pPAL* is a pointer to the underlying NAND flash model (i.e., parallelism abstraction layer (PAL)), which can be used to obtain the latency of flash operations from a precise flash model. We will discuss it shortly. *table* implements the superpage-level mapping from LPNs to PPNs which are described by a flash superpage number and a page offset within the superpage. Note that the superpage-level is slightly different from pure page-level mapping. *blocks* stores flash block status information, such as the page valid information, erase count, and write pointer of blocks. *freeBlocks* is a collection of free blocks, while *nFreeBlocks* stores the number of free blocks, which is used to check whether GC is needed.

The following figure depicts all functions implemented by PageMapping. PageMapping simply implements the public interface required by AbstractFTL (e.g., *read* and *write*) while placing the detailed logic into the internal functions (e.g., *readInternal* and *writeInternal* functions). To perform GC, PageMapping calculates the weight of each block based on wear-leveling, selects the victim blocks, and executes the GC operation through *calculateVictimWeight*, *selectVictimBlock,* and *doGarbageCollection* function, respectively.



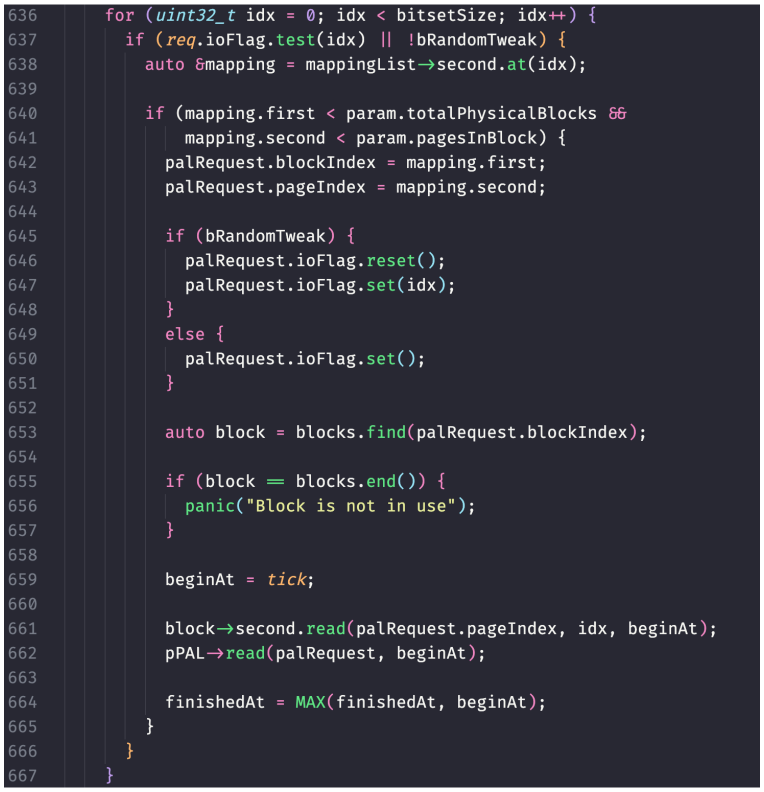
```
 float freeBlockRatio();
 uint32_t convertBlockIdx(uint32_t);
 uint32_t getFreeBlock(uint32_t);
 uint32_t getLastFreeBlock(Bitset &);
 void calculateVictimWeight(std::vector<std::pair<uint32_t, float>> &,
                            const EVICT_POLICY, uint64_t);
 void selectVictimBlock(std::vector<uint32_t> &, uint64_t &);
 void doGarbageCollection(std::vector<uint32_t> &, uint64_t &);

 float calculateWearLeveling();
 void calculateTotalPages(uint64_t &, uint64_t &);

 void readInternal(Request &, uint64_t &);
 void writeInternal(Request &, uint64_t &, bool = true);
 void trimInternal(Request &, uint64_t &);
 void eraseInternal(PAL::Request &, uint64_t &);

public:
 PageMapping(ConfigReader &, Parameter &, PAL::PAL *, DRAM::AbstractDRAM *);
 ~PageMapping();

 bool initialize() override;

 void read(Request &, uint64_t &) override;
 void write(Request &, uint64_t &) override;
 void trim(Request &, uint64_t &) override;

 void format(LPNRange &, uint64_t &) override;

 Status *getStatus(uint64_t, uint64_t) override;

 void getStatList(std::vector<Stats> &, std::string) override;
 void getStatValues(std::vector<double> &) override;
 void resetStatValues() override;
```

Read opertion in FTL module. We will use the read operation as an example to introduce how PageMapping operates and how it interacts with the upper and lower layers in SimpleSSD. As shown in the following figure, PageMapping first looks up the mapping table (i.e., *table*) to translate the LPN to the PPN which is stored in *mappingList*.



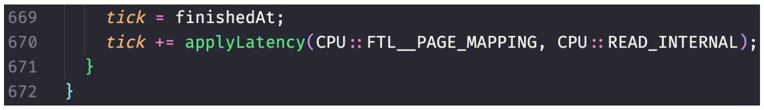
```
void PageMapping::readInternal(Request &req, uint64_t &tick) {
  PAL::Request palRequest(req);
  uint64_t beginAt;
  uint64_t finishedAt = tick;

  auto mappingList = table.find(req.lpn);

  if (mappingList != table.end()) {
    if (bRandomTweak) {
      pDRAM->read(&(*mappingList), 8 * req.ioFlag.count(), tick);
    }
    else {
      pDRAM->read(&(*mappingList), 8, tick);
    }
```

PageMapping then locates the data required by the upper layer request in the superpage and subsequently constructs the underlying layer request (i.e., *palRequest*) based on the address of the requested data. Further, PageMapping sends the *palRequest* to the NAND flash model via the *pPAL* pointer and obtains the flash read latency in the *beginAt* variable.



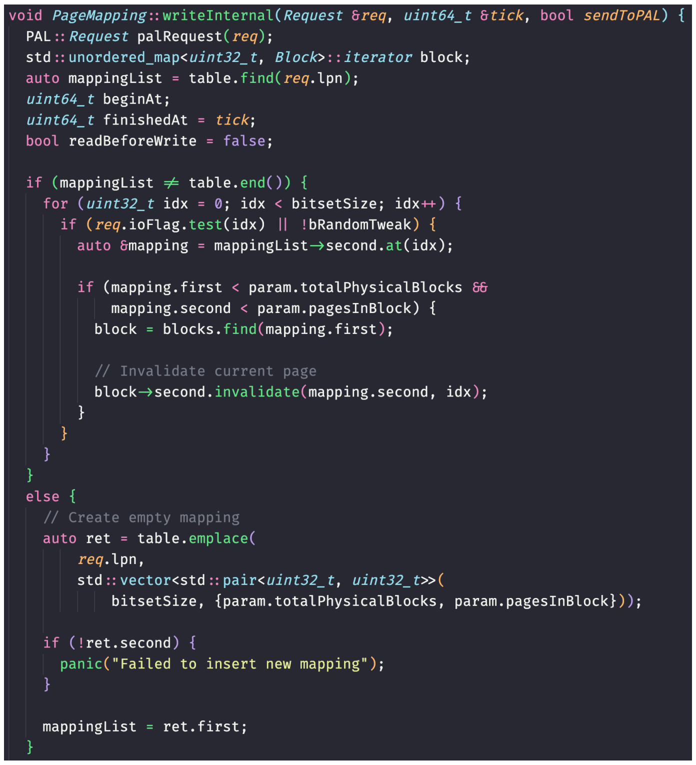
```
    for (uint32_t idx = 0; idx < bitsetSize; idx++) {
      if (req.ioFlag.test(idx) || !bRandomTweak) {
        auto &mapping = mappingList->second.at(idx);

        if (mapping.first < param.totalPhysicalBlocks &&
            mapping.second < param.pagesInBlock) {
          palRequest.blockIndex = mapping.first;
          palRequest.pageIndex = mapping.second;

          if (bRandomTweak) {
            palRequest.ioFlag.reset();
            palRequest.ioFlag.set(idx);
          }
          else {
            palRequest.ioFlag.set();
          }

          auto block = blocks.find(palRequest.blockIndex);

          if (block == blocks.end()) {
            panic("Block is not in use");
          }

          beginAt = tick;

          block->second.read(palRequest.pageIndex, idx, beginAt);
          pPAL->read(palRequest, beginAt);

          finishedAt = MAX(finishedAt, beginAt);
        }
      }
    }
```

PageMapping updates the request latency by taking the maximum value of the returned read latency. PageMapping also adds the firmware latency to the request latency. But in this project, you can neglect the firmware latncy. Finally, the request latency is returned to the upper layer.



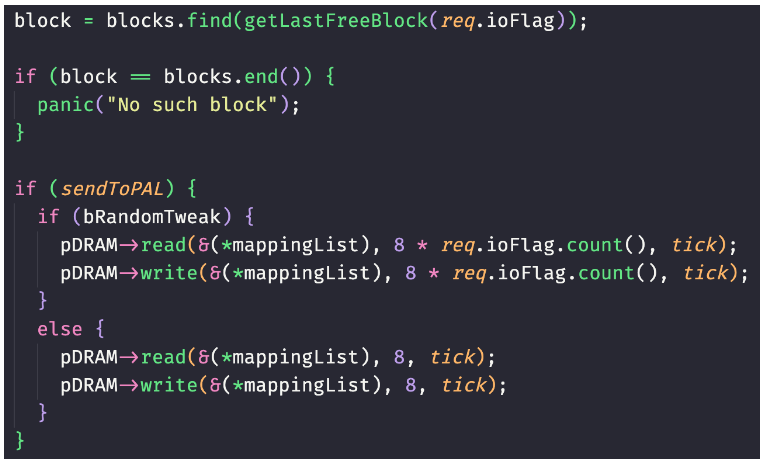
```
    tick = finishedAt;
    tick += applyLatency(CPU::FTL__PAGE_MAPPING, CPU::READ_INTERNAL);
  }
}
```

Write opertion in FTL module. The subsequent section outlines write operations performed by PageMapping. Specifically, PageMapping employs *Block* class to manage the flash blocks currently in use within the SSD. As we describe before, this class encompasses various information, including the validity status of individual pages and LPN associated with each page. Moreover, *Block* class offers multiple functions that emulate real flash operations, such as writing a page to a flash block (*write* function) and invalidating a page within the flash block (*invalidate* function). **It is crucial to carefully examine its code as FAST frequently queries the flash block information.** Notice that since SimpleSSD supports different levels of superpage mapping, *mappingList* actually points to a virtual superpage comprising a *bitsetSize* number of flash pages, while *idx* denotes the offset of the requested page within the superpage. However, for the sake of simplicity, we configure the superpage mapping to the most basic form, where **a superpage consists of only one flash page**. When PageMapping starts processing a write request, it first invalidates the obsolete page. This process involves finding the previous mapping from *table*, subsequently locating the corresponding *block*, and finally invoking the *invalidate* function.



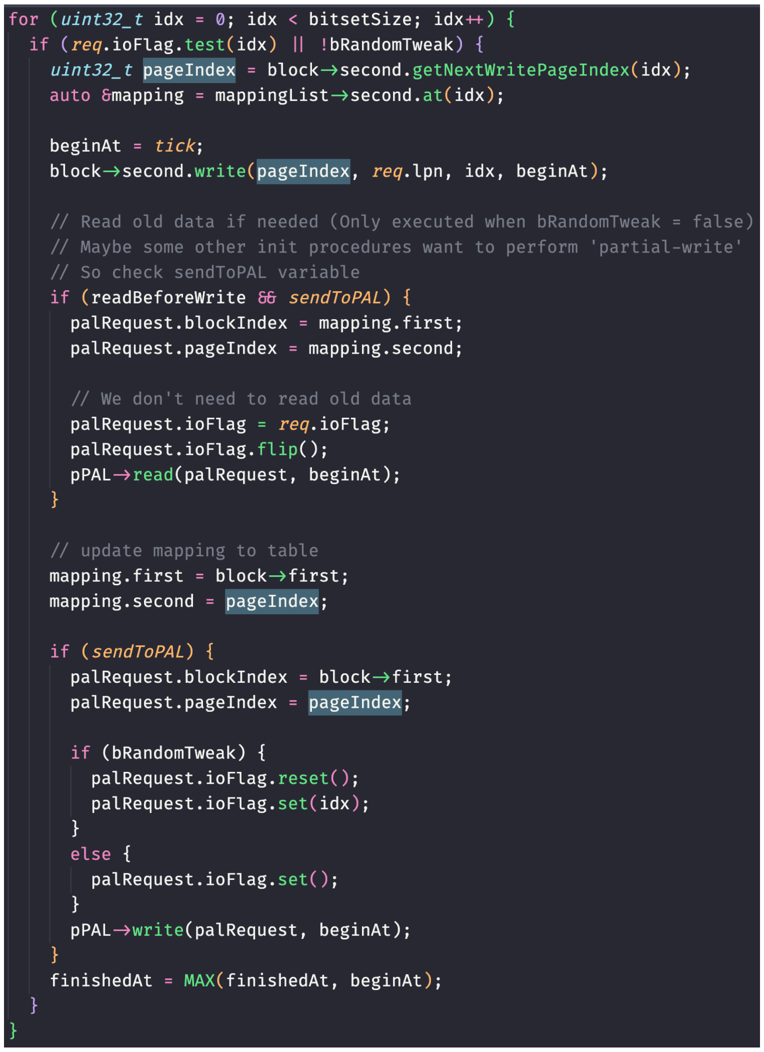
```cpp
void PageMapping::writeInternal(Request &req, uint64_t &tick, bool sendToPAL) {
  PAL::Request palRequest(req);
  std::unordered_map<uint32_t, Block>::iterator block;
  auto mappingList = table.find(req.lpn);
  uint64_t beginAt;
  uint64_t finishedAt = tick;
  bool readBeforeWrite = false;

  if (mappingList != table.end()) {
    for (uint32_t idx = 0; idx < bitsetSize; idx++) {
      if (req.ioFlag.test(idx) || !bRandomTweak) {
        auto &mapping = mappingList->second.at(idx);

        if (mapping.first < param.totalPhysicalBlocks &&
            mapping.second < param.pagesInBlock) {
          block = blocks.find(mapping.first);

          // Invalidate current page
          block->second.invalidate(mapping.second, idx);
        }
      }
    }
  }
  else {
    // Create empty mapping
    auto ret = table.emplace(
        req.lpn,
        std::vector<std::pair<uint32_t, uint32_t>>(
            bitsetSize, {param.totalPhysicalBlocks, param.pagesInBlock}));

    if (!ret.second) {
      panic("Failed to insert new mapping");
    }

    mappingList = ret.first;
  }
```

The FTL proceeds by allocating a free flash page for the upcoming write via *getLastFreeBlock* function. Notice that to optimize performance, SimpleSSD customizes the allocation policy to suit the flash parallelism. But you can simplify this allication process in your implementation. This can be achieved by maintaining a linked list of free blocks, where each write utilizes the block at the head, and any reclaimed free block is inserted at the end (i.e., the FIFO policy)



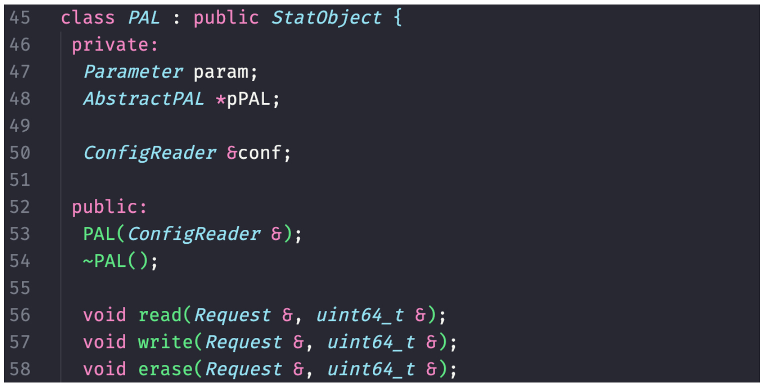
```cpp
block = blocks.find(getLastFreeBlock(req.ioFlag));

if (block == blocks.end()) {
  panic("No such block");
}

if (sendToPAL) {
  if (bRandomTweak) {
    pDRAM->read(&(*mappingList), 8 * req.ioFlag.count(), tick);
    pDRAM->write(&(*mappingList), 8 * req.ioFlag.count(), tick);
  }
  else {
    pDRAM->read(&(*mappingList), 8, tick);
    pDRAM->write(&(*mappingList), 8, tick);
  }
}
```

Finally, similar to a read operation, the FTL emulates a write operation in the block by validing the flash page and recording the corresponding LPN via *write* function. Subsequently, a *palRequest* is dispatched to the underlying *PAL* to accquire the write latency. It is worth noting that *sendToPAL* is utlized to distinguish whether the write request originates from the *warmup* phase or the emulation phase. The *warmup* phase is specifically designed to replicate the real SSD state after a prolonged period of operation, and it constructs write requests to change the SSD internal states (e.g., FTL mappings). To rapidly reach the steady state, write requests in the warmup phase does not need to be forwarded to the *PAL*, instead, they solely necessitate modification to metadata within the FTL. Similarly, in your implementation, it is imperative to support the warmup operation to emulate internal states in real SSDs.



```
for (uint32_t idx = 0; idx < bitsetSize; idx++) {
  if (req.ioFlag.test(idx) || !bRandomTweak) {
    uint32_t pageIndex = block->second.getNextWritePageIndex(idx);
    auto &mapping = mappingList->second.at(idx);

    beginAt = tick;
    block->second.write(pageIndex, req.lpn, idx, beginAt);

    // Read old data if needed (Only executed when bRandomTweak = false)
    // Maybe some other init procedures want to perform 'partial-write'
    // So check sendToPAL variable
    if (readBeforeWrite && sendToPAL) {
      palRequest.blockIndex = mapping.first;
      palRequest.pageIndex = mapping.second;

      // We don't need to read old data
      palRequest.ioFlag = req.ioFlag;
      palRequest.ioFlag.flip();
      pPAL->read(palRequest, beginAt);
    }

    // update mapping to table
    mapping.first = block->first;
    mapping.second = pageIndex;

    if (sendToPAL) {
      palRequest.blockIndex = block->first;
      palRequest.pageIndex = pageIndex;

      if (bRandomTweak) {
        palRequest.ioFlag.reset();
        palRequest.ioFlag.set(idx);
      }
      else {
        palRequest.ioFlag.set();
      }
      pPAL->write(palRequest, beginAt);
    }
    finishedAt = MAX(finishedAt, beginAt);
  }
}
```

PAL module in SimpleSSD. We move on to the *PAL* in SimpleSSD, which models the NAND flash. The following figure depicts functions provided by PAL.



```
class PAL : public StatObject {
 private:
  Parameter param;
  AbstractPAL *pPAL;

  ConfigReader &conf;

 public:
  PAL(ConfigReader &);
  ~PAL();

  void read(Request &, uint64_t &);
  void write(Request &, uint64_t &);
  void erase(Request &, uint64_t &);
```

PAL implements *read*, *write*, and *erase* functions that receive the requests from the FTL scheme and return the latency of the corresponding flash operations. Note that you **do not** need to pay much attention to the details of the PAL. You can view the PAL as a black-box model, that is, you provide reasonable parameters for these PAL functions and get the latency simulated by the black-box model.

Configuration settings in SimpleSSD. Finally, we describe the configuration settings in SimpleSSD. SimpleSSD provides the default *simulation configuration file* and *SimpleSSD configuration file*, which are located in *simplessd-standalone/config/sample.cfg* and *simplessd-standalone/simplessd/config/sample.cfg*, respectively.

The simulation configuration file (i.e., simplessd-standalone/config/sample.cfg) sets the key parameters of the simulation environment. Specifically, *LogFile*, *DebugLogFile*, and *LatencyLogFile* set the output paths of the statistic log, debug log, and latency log, respectively. These log files help you observe the behavior of the SSD. For example, *DebugLogFile* records the log output during simulation, which can track the processing path of each request, while *LatencyLogFile* holds statistics for each request in the format of (request ID, address offset, request length, latency). Additionally, SimpleSSD provides an I/O generator and a trace replayer to generate workloads for evaluations, and which one to use is decided by *Mode* in this file. You can use them to evaluate the performance of your FTL scheme implementation. The parameters in the *generator section* of the configuration file determine the characteristics of the generated microbenchmark, such as the total I/O size, the I/O type, and the I/O size of a single I/O request. The parameters in the *trace section* determine how to replay a macro benchmark, such as *File* to specify the trace file and *TimingMode* to specify how the workload is replayed (i.e., synchronous or asynchronous replay). The configuration file also provides a regular expression parameter to parse I/O requests from the trace file, such as the size and address range of requests. For more parameters, please refer to the simulation configuration file.

The SimpleSSD configuration file sets the key parameters of the simulated SSD, including the embedded cores, NVMe, FTL, ICL, and PAL. This file provides a detailed explanation for each parameter, and you can refer to this file to learn how to set the parameters.

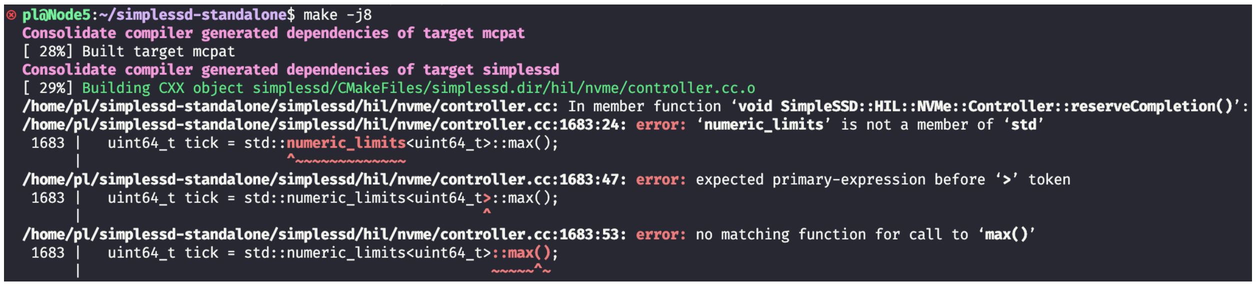
SimpleSSD Setup

The SimpleSSD document shows detailed procedures to compile and run SimpleSSD. In this part, we only show additional steps to successfully run SimpleSSD on our platform. Our platform is Ubuntu 22.04.2 LTS with the Linux Kernel 5.19.0.
The first thing you need to do is download and build SimpleSSD-Standalone by running the following commands:

```
git clone git@github.com:simplessd/simplessd-standalone
cd simplessd-standalone
git submodule update --init –recursive
cmake -DDEBUG_BUILD=off .
make -j 8
```

When building SimpleSSD, you may encounter some compile error issues. For your convenience, we provide solutions for some common issues.

Issue 1.

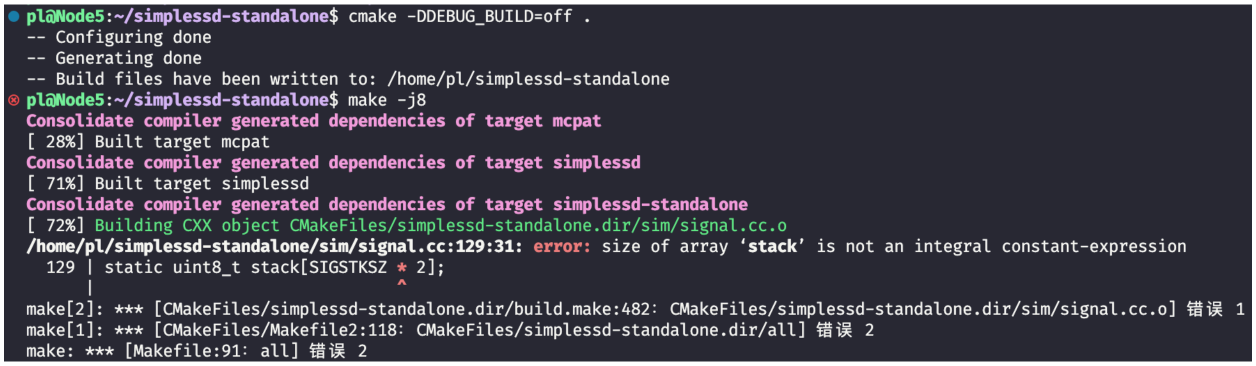


```
⊗ pl@Node5:~/simplessd-standalone$ make -j8
  Consolidate compiler generated dependencies of target mcpat
  [ 28%] Built target mcpat
  Consolidate compiler generated dependencies of target simplessd
  [ 29%] Building CXX object simplessd/CMakeFiles/simplessd.dir/hil/nvme/controller.cc.o
  /home/pl/simplessd-standalone/simplessd/hil/nvme/controller.cc: In member function ‘void SimpleSSD::HIL::NVMe::Controller::reserveCompletion()’:
  /home/pl/simplessd-standalone/simplessd/hil/nvme/controller.cc:1683:24: error: ‘numeric_limits’ is not a member of ‘std’
   1683 |   uint64_t tick = std::numeric_limits<uint64_t>::max();
        |                        ^~~~~~~~~~~~~~
  /home/pl/simplessd-standalone/simplessd/hil/nvme/controller.cc:1683:47: error: expected primary-expression before ‘>’ token
   1683 |   uint64_t tick = std::numeric_limits<uint64_t>::max();
        |                                               ^
  /home/pl/simplessd-standalone/simplessd/hil/nvme/controller.cc:1683:53: error: no matching function for call to ‘max()’
   1683 |   uint64_t tick = std::numeric_limits<uint64_t>::max();
        |                                                ~~~~~^~
```

You can fix this error by updating simplessd-standalone/simplessd/hil/nvme/controller.cc based on the following patch.

```
diff --git a/hil/nvme/controller.cc b/hil/nvme/controller.cc
--- a/hil/nvme/controller.cc
+++ b/hil/nvme/controller.cc
@@ -20,6 +20,7 @@
 #include "hil/nvme/controller.hh"
 #include <algorithm>
+#include <limits>
 #include <cmath>
```

Issue 2.



```
● pl@Node5:~/simplessd-standalone$ cmake -DDEBUG_BUILD=off .
  -- Configuring done
  -- Generating done
  -- Build files have been written to: /home/pl/simplessd-standalone
⊗ pl@Node5:~/simplessd-standalone$ make -j8
  Consolidate compiler generated dependencies of target mcpat
  [ 28%] Built target mcpat
  Consolidate compiler generated dependencies of target simplessd
  [ 71%] Built target simplessd
  Consolidate compiler generated dependencies of target simplessd-standalone
  [ 72%] Building CXX object CMakeFiles/simplessd-standalone.dir/sim/signal.cc.o
  /home/pl/simplessd-standalone/sim/signal.cc:129:31: error: size of array ‘stack’ is not an integral constant-expression
    129 | static uint8_t stack[SIGSTKSZ * 2];
        |                               ^
  make[2]: *** [CMakeFiles/simplessd-standalone.dir/build.make:482: CMakeFiles/simplessd-standalone.dir/sim/signal.cc.o] 错误 1
  make[1]: *** [CMakeFiles/Makefile2:118: CMakeFiles/simplessd-standalone.dir/all] 错误 2
  make: *** [Makefile:91: all] 错误 2
```

You can fix this error by updating simplessd-standalone/sim/signal.cc based on the following patch.

```
diff --git a/sim/signal.cc b/sim/signal.cc
--- a/sim/signal.cc
+++ b/sim/signal.cc
@@ -126,7 +126,7 @@ LONG WINAPI exceptionHandler(LPEXCEPTION_POINTERS pExceptionInfo)
 {
 #define FRAMECOUNT 32
-static uint8_t stack[SIGSTKSZ * 2];
+static uint8_t stack[65536 * 2];
```

After successfully building SimpleSSD, you can use the following command to run SimpleSSD. SimpleSSD runs the workload according to the settings in the *Simulation configuration file* and *SimpleSSD configuration file*, while outputting the simulation results and logs to the path specified by the *Output directory*.

```
# Usage: simplessd-standalone <Simulation configuration file> <SimpleSSD configuration file> <Output directory>
./simplessd-standalone ./config/sample.cfg ./simplessd/config/sample.cfg .
```

Task: Read the paper, Implement, and Evaluate FAST

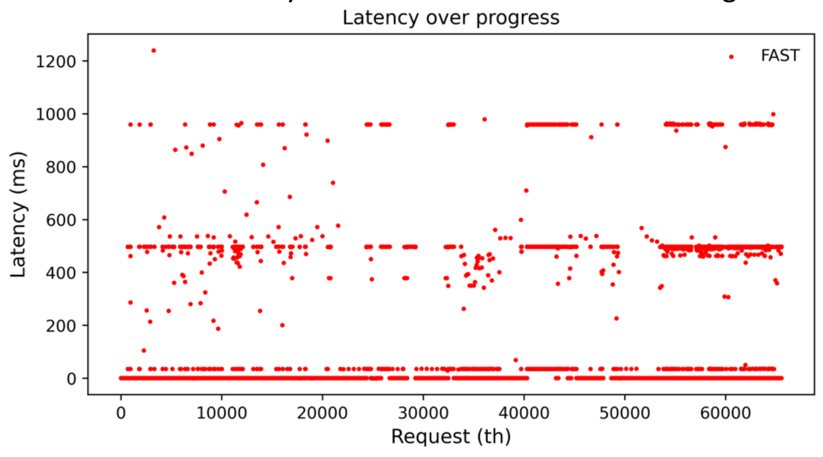
The task in this project is to read the paper [1], implement the FAST hybrid-level mapping scheme in SimpleSSD, and evaluate it.

- Implement FAST on **SimpleSSD-Standalone**. For the sake of simplicity, you do not need to implement the O-FAST optimization in the paper.

- For your convenience, we provide configuration files and macrobenchmark, which can be found in the project #4 package.
- Please ensure that the memory consumption of your implementation meets the requirements of FAST.

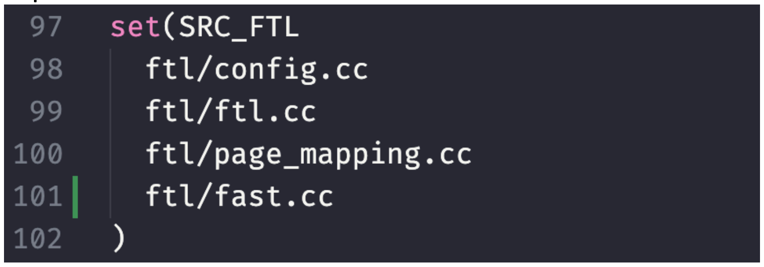
Project deliverables:

1. Write a concise description of your implementation which includes key modifications. This should include the key data structures used in the reimplementation as well as an illustration of the key workflow. Further, you should **analyze** the amount of DRAM utilized by the address mapping table in your implementation, expressed as a percentage of the SSD size.
2. Please commit your implementation as a git patch. Please ensure your patch can accurately reproduce your evaluation results after applying it directly to the default SimpleSSD.
3. Compare the performance of your implementation with PageMapping in SimpleSSD. Specifically, evaluate the I/O latency, the bandwidth and the latency for each request by running microbenchmark and macrobenchmark.
   a) For the microbenchmark, you need to evaluate various I/O patterns, including sequential read, sequential write, random read, random write and sequential read/write miexed, and random read/write mixed. The total I/O size should be set at 128MB, with each I/O request measuring 4KB.
   b) For the macrobenchmark, we provide 5 traces obtained from the FIU traces. These workload exhibits complicated I/O patterns that can check the correctness of your implementation. In addition, workload labels each request as sequential or random. Based on this information and combined with latency data, the correctness of your implementation can be checked. For example, sequential writes should benefit from the SW log block design.
   c) In all benchmark evaluations, the I/O queue depth is set to 1, implying that each request must wait for the completion of the preceding request before being issued. This simplified situation aids in validating the correctness of your implementation. Additionally, you need to warm up the SSD before evaluating, as failure to do so yield inaccurate latency. In the provided configuaration files, the warm up ratio is set to 50%.
   d) Note that all evaluation results should be present as **figures**. For easier understanding, we show an example figure depicting the latency for each request. Similar latency spikes are expected to be observed in your evaluation results when merges are triggered.



Implementation Guide

1. **Implementing FAST on SimpleSSD.** You can refer to PageMapping to implement your hybrid-level mapping. For example, you can refer to its functions and member variables, while the merging operations can be modeled after its GC module. Specifically, you can create a class similar to PageMapping, that is, inherit the FTL abstract class (i.e., AbstractFTL), implement the logic of the hybrid-level mapping, and implement the interface required by AbstractFTL. A slight difference between your implementation and the paper is acceptable. Note that to compile your implementation correctly, you need to include your filename in *CMakeLists*.txt located in the *simplessd* folder, which ensures that your implementation is properly integrated into SimpleSSD.



```
set(SRC_FTL
  ftl/config.cc
  ftl/ftl.cc
  ftl/page_mapping.cc
  ftl/fast.cc
)
```

2. **SimpleSSD, simulation and FAST configuration.** To ensure accurate results, we carefully configure SimlpeSSD and simulation environment. Specifically, we disable the DRAM cache in SimpleSSD to prevent it from hiding the effects of your mapping scheme. Moreover, we set up the simulation to the asynchronous I/O replay mode. Regarding the FAST configuration, since GC are no longer necessary in FAST, you can keep the OverProvision blocks reserved in the default SimpleSSD as free blocks, which can be allocated when block merges are triggered. Additionally, we allocate **a SW log block** and **six RW log block** from these blocks, which quickly triggers block merges under the our workloads.
3. ***Block* class modification.** By default, SimpleSSD restricts each flash block to be written only sequentially. But in FAST random writes can also occur within data blocks. Therefore, we suggest removing this restriction.
4. **Debug log file.** Note that the debug log file can be very large (e.g., 2 GB), therefore you can turn off the debug log output when running the final evaluation.

Appendix A

[1] Lee, S. W., Park, D. J., Chung, T. S., Lee, D. H., Park, S., & Song, H. J. (2007). A log buffer-based flash translation layer using fully-associative sector translation. ACM Transactions on Embedded Computing Systems (TECS), 6(3), 18-es.

[2] Gouk, D., Kwon, M., Zhang, J., Koh, S., Choi, W., Kim, N. S., ... & Jung, M. (2018, October). Amber: Enabling precise full-system simulation with detailed modeling of all SSD resources. In 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) (pp. 469-481). IEEE.