P&S Modern SSDs

Introduction to MQSim

Rakesh Nadig
Dr. Jisung Park
Prof. Onur Mutlu
ETH Zürich

Spring 2022

8th April 2022

Introduction

- Rakesh Nadig
  - PhD Student @ SAFARI research group since 2021
  - Senior staff engineer @ Samsung Electronics 2014-2021
  - MS in Electrical and Computer Engineering from University of California Irvine
  - Research Area: Memory/Storage Systems | NAND Flash Memory | Near-Storage Processing | Non-Volatile Memory | Machine Learning | Hybrid Memory/Storage Systems
  - rakesh.nadig@safari.ethz.ch

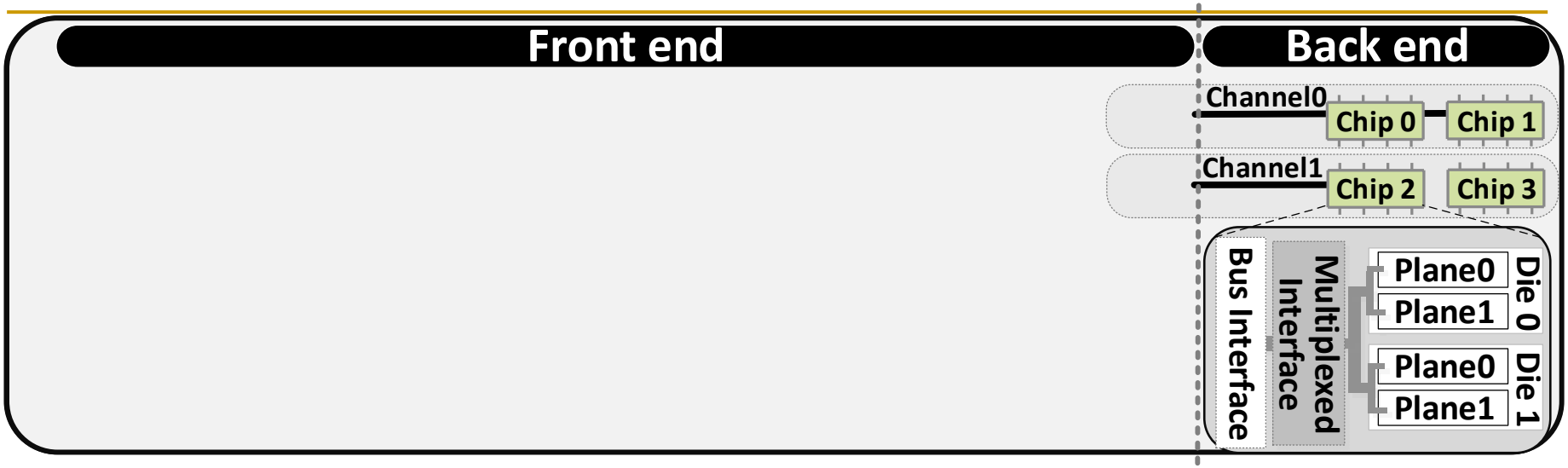
Outline

- Internal Components of a Modern SSD
- Introduction to MQSIM
- Mechanism
- Code structure
- Configuring MQSim

Outline

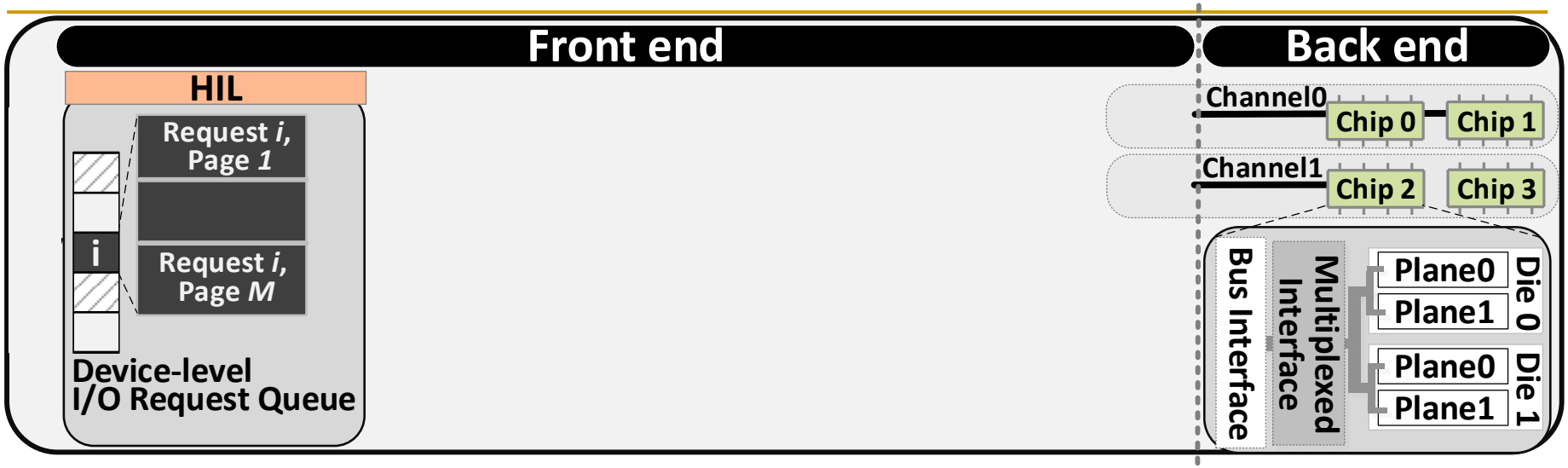
- Internal Components of a Modern SSD
- Introduction to MQSIM
- Mechanism
- Code structure
- Configuring MQSim

Internal Components of a Modern SSD



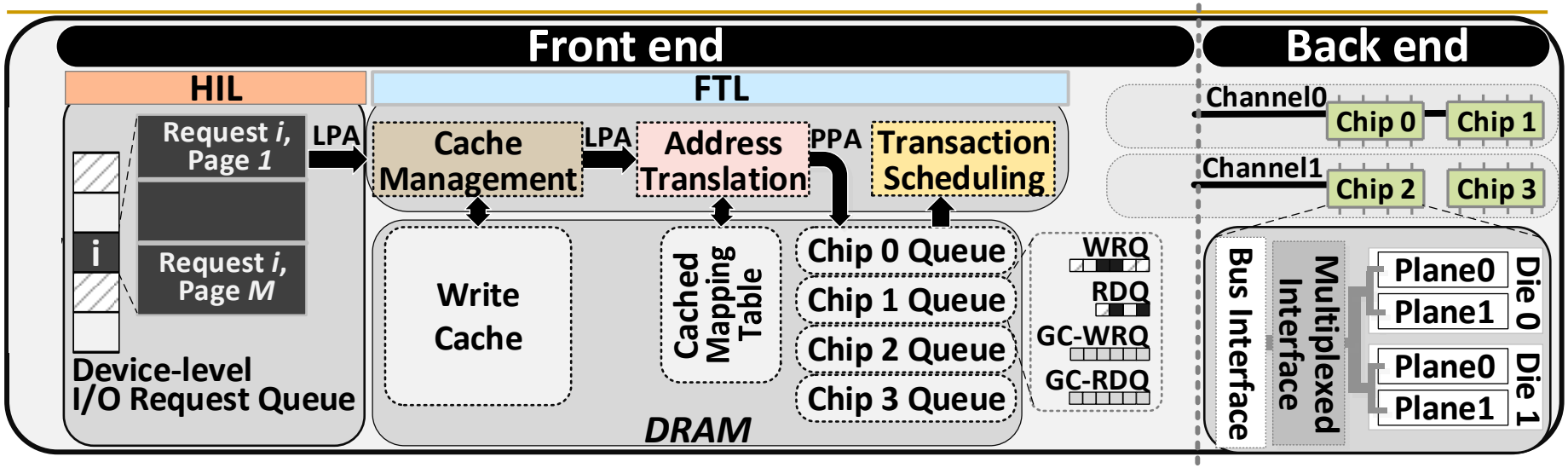
- Back End: data storage
  - Memory chips (e.g., NAND flash memory, PCM, MRAM, 3D XPoint)

Internal Components of a Modern SSD



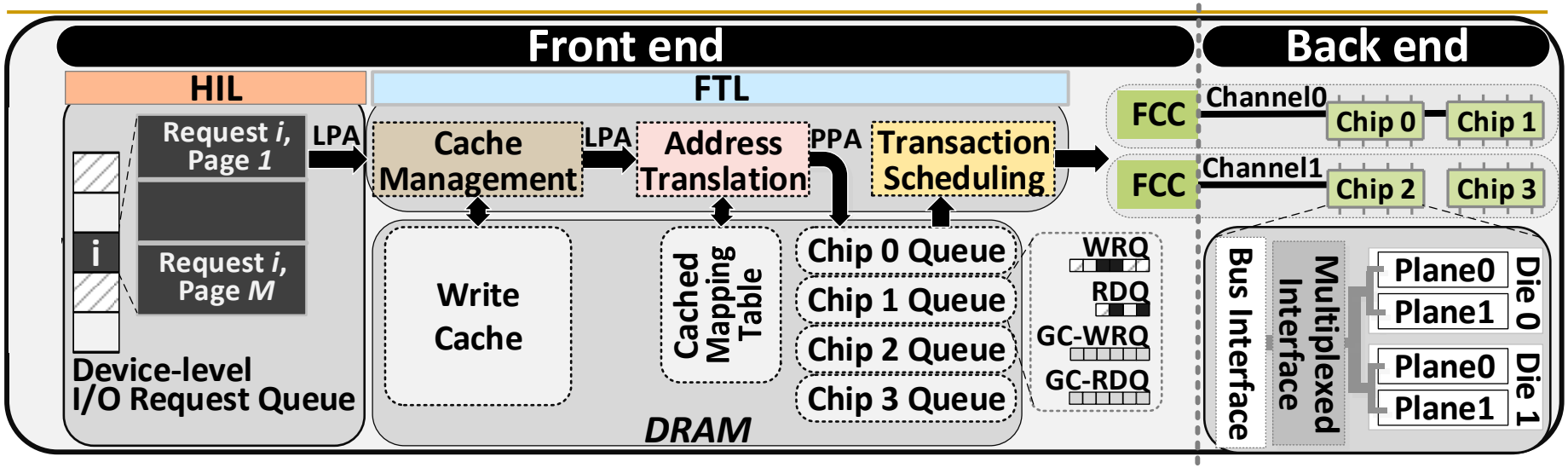
- Back End: data storage
  - Memory chips (e.g., NAND flash memory, PCM, MRAM, 3D XPoint)
- Front End: management and control units
  - **Host–Interface Logic (HIL)**: protocol used to communicate with host

Internal Components of a Modern SSD



- Back End: data storage
  - Memory chips (e.g., NAND flash memory, PCM, MRAM, 3D XPoint)
- Front End: management and control units
  - **Host–Interface Logic (HIL)**: protocol used to communicate with host
  - **Flash Translation Layer (FTL)**: manages resources, processes I/O requests

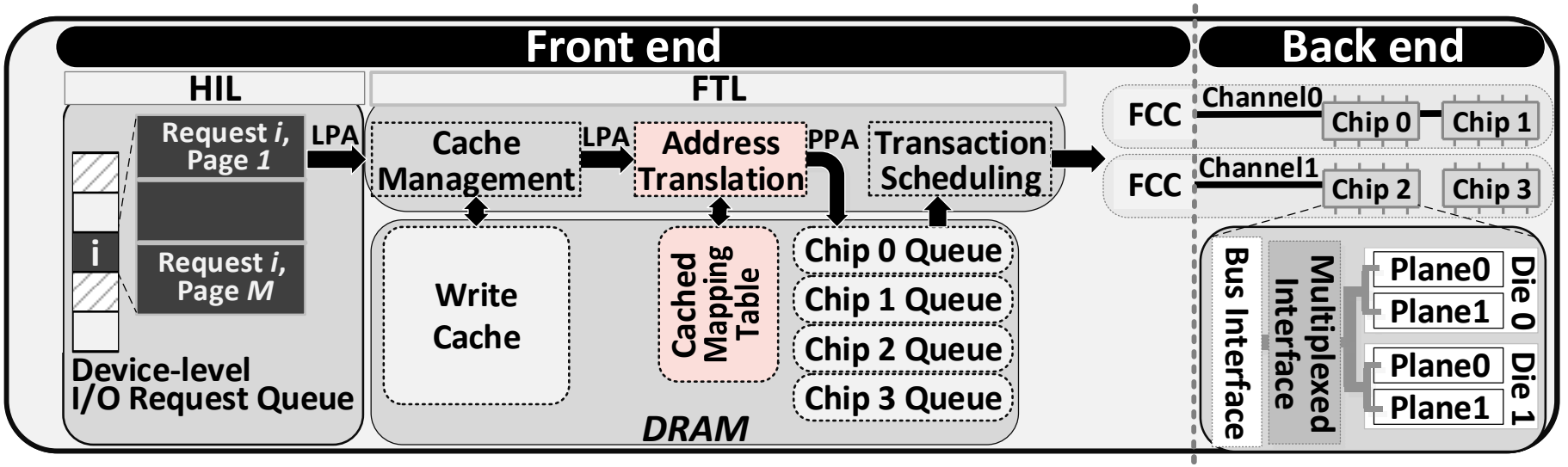
Internal Components of a Modern SSD



- Back End: data storage
  - Memory chips (e.g., NAND flash memory, PCM, MRAM, 3D XPoint)
- Front End: management and control units
  - **Host–Interface Logic (HIL)**: protocol used to communicate with host
  - **Flash Translation Layer (FTL)**: manages resources, processes I/O requests
  - **Flash Channel Controllers (FCCs)**: sends commands to, transfers data with memory chips in back end

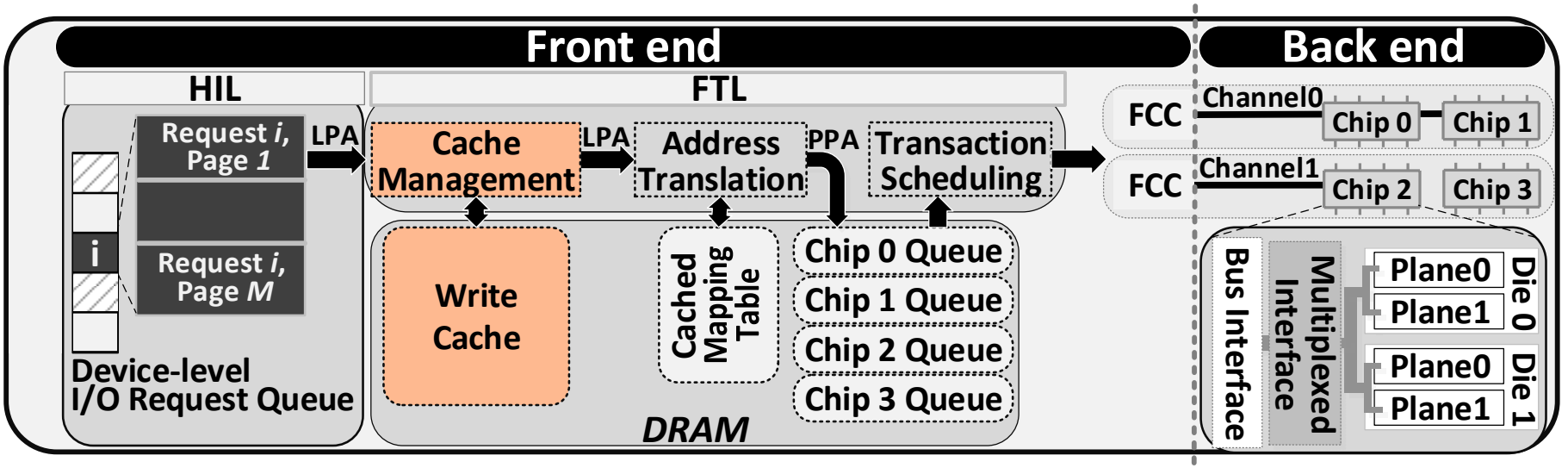
FTL: Managing the SSD's Resources

- Flash writes can take place only to pages that are erased
  - Perform **out-of-place updates** (i.e., write data to a different, free page), mark **old page as invalid**
  - **Update logical-to-physical mapping** (makes use of ***cached mapping table***)
  - Some time later: **garbage collection** reclaims invalid physical pages *off the critical path of latency*



FTL: Managing the SSD's Resources

- Flash writes can take place only to pages that are erased
  - Perform **out-of-place updates** (i.e., write data to a different, free page),
    mark **old page as invalid**
  - **Update logical-to-physical mapping** (makes use of ***cached mapping table***)
  - Some time later: **garbage collection** reclaims invalid physical pages *off the critical path of latency*
- Write cache decreases resource contention, reduces latency

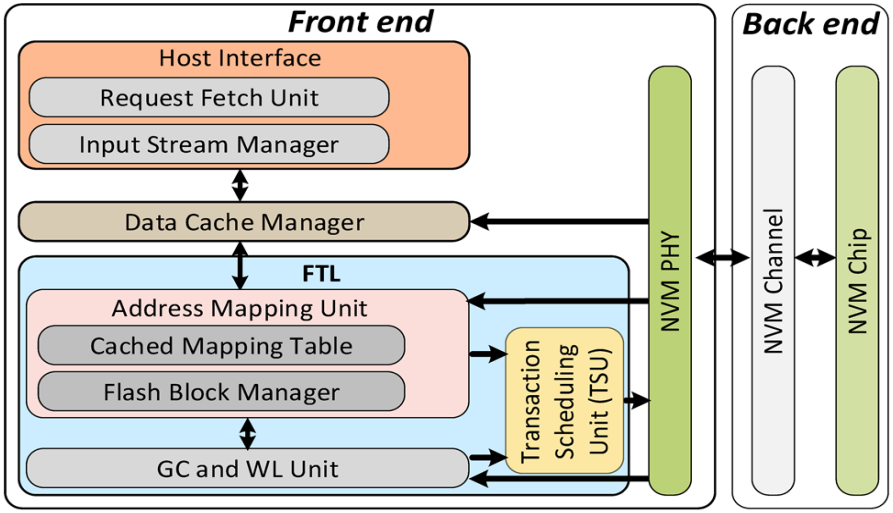


Outline

- Internal Components of a Modern SSD
- Introduction to MQSIM
- Mechanism
- Code structure
- Configuring MQSim

MQSim

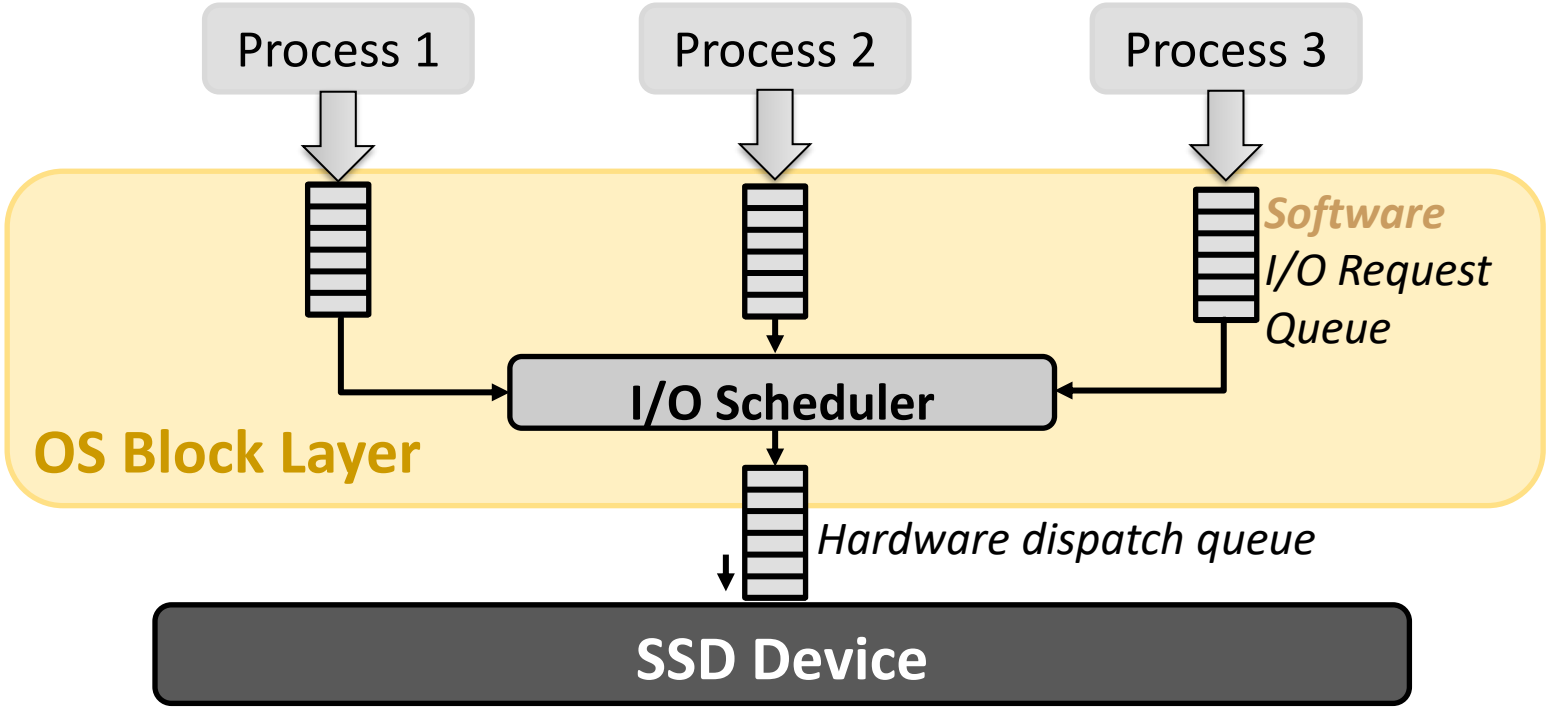
- Accurately models conventional SATA-based SSDs and modern multi-queue SSDs
  - Multi-queue protocols
  - Supports steady-state behavior with preconditioning
  - Models end-to-end I/O request latency



- Flexible design
  - Modular components
  - Ability to support **emerging non-volatile memory (NVM) technologies**
- Open-source release: http://github.com/CMU-SAFARI/MQSim
  - Written in C++
  - MIT License

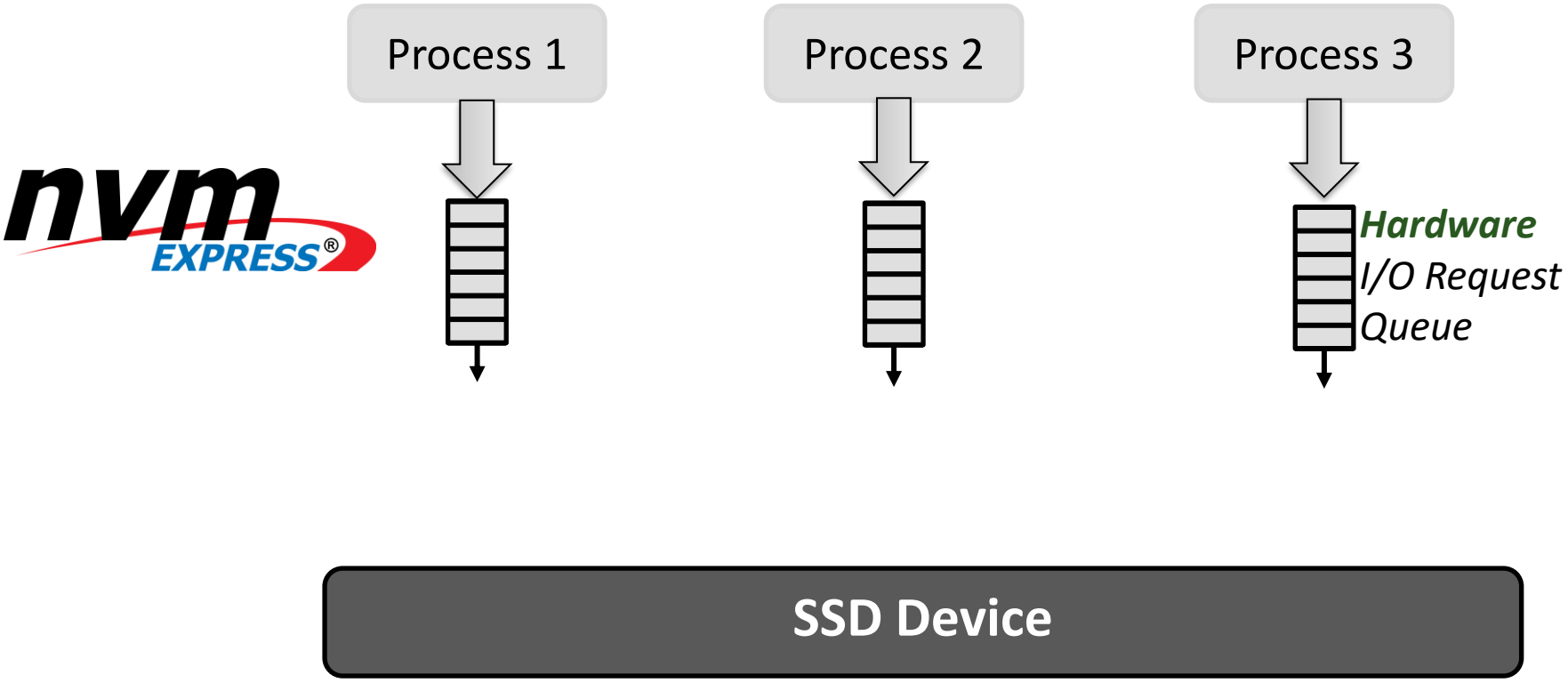
Support for Multi-Queue Protocols

- Conventional host interface (e.g., SATA)
  - Designed for magnetic hard disk drives: only **thousands of IOPS** per device
  - **OS handles scheduling, fairness** control for I/O requests

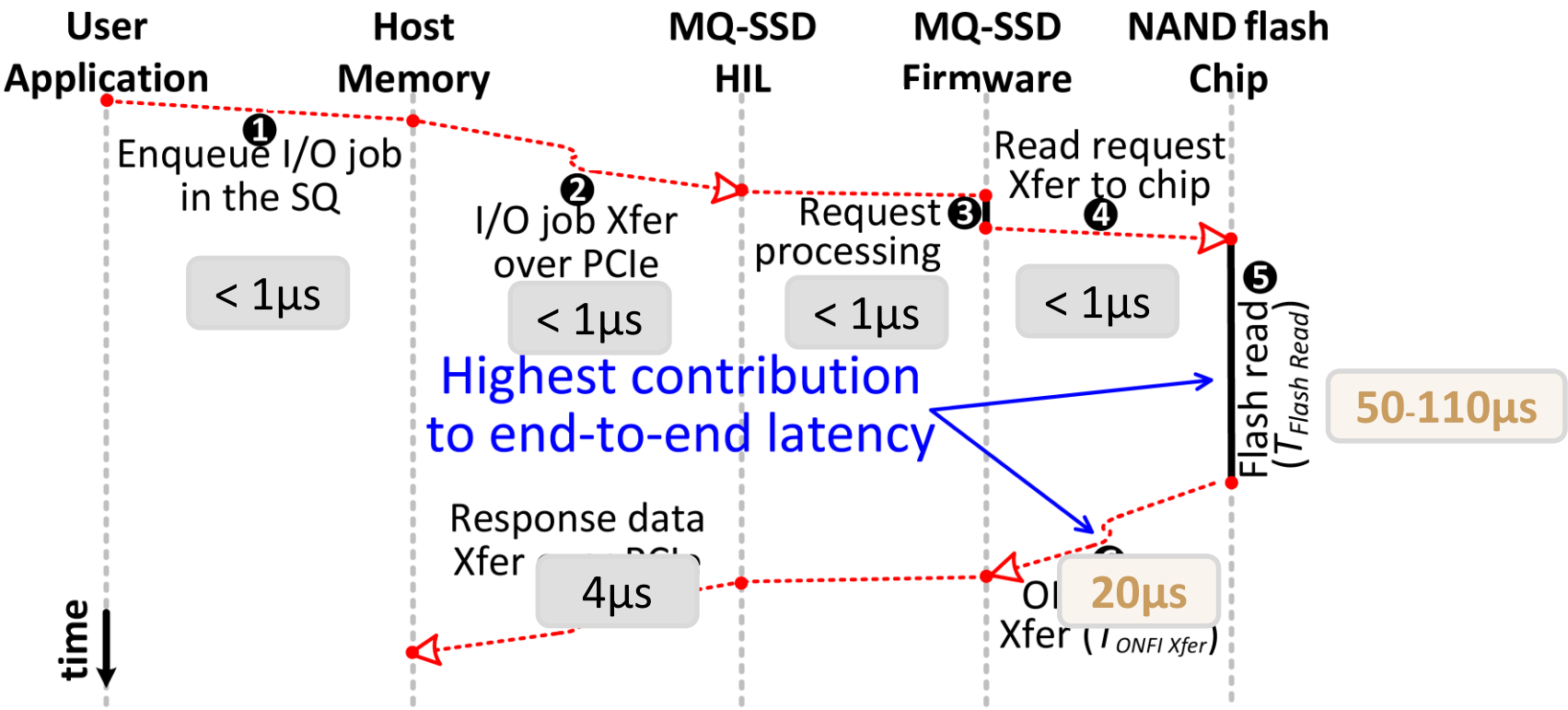


Support for Multi-Queue Protocols

- Modern host interface (e.g., NVMe)
  - Takes advantage of SSD throughput: enables **millions of IOPS** per device
  - Avoids OS intervention: **SSD must perform scheduling, ensure fairness**



Complete Model of Request Latency



SSD Back End Model

- Three major latency components of the SSD back end
  - **Address and command transfer** to the memory chip
  - Flash memory **read/write execution** for different NAND technologies (1-bit, 2-bit and 3-bit per cell)
  - **Data transfer** to/from memory chips
- **Die- and plane-level parallelism** and advanced command execution
- Decouples the sizes of read and write operations
  - **Page-sized write** operations
  - **Sub-page read** operations

SSD Front End Model

- Host-Interface Model
  - **NVMe multi-queue (MQ)** and **SATA native command queue** models for a modern SSD
  - **Different priority classes** for **host side request queues** as per the NVMe standard specification
- Data Cache Manager
  - **DRAM-based cache** with Least Recently Used (LRU) replacement policy to cache recently accessed data
- FTL Components
  - Address translation unit
  - Garbage collection and wear-leveling unit
  - Transaction scheduling unit
  - Support for multi-flow request processing

Outline

- Internal Components of a Modern SSD
- Introduction to MQSIM
- Mechanism
- Code structure
- Configuring MQSim

Mechanism

- MQSim is a **discrete-event** based simulator
- Each operation performed by MQSim is an event (Command/Address transfer, Read data out etc.)
- A **red-black (RB)** tree maintains the events in the order of their completion times
- Each node of the RB tree has a **key** which indicates the **timestamp** to finish and, the **value** is the **callback** function
- For example, at time 5, a data transfer event starts, and the event will execute for another 5 time-units. The simulator will insert a node <5 + 5, callback> into the tree
- At each simulation cycle, the simulator will find the node with the smallest time (value), set the timer to that timestamp, execute the callback function
- MQSim supports Real disk traces (e.g., MSR Cambridge Block I/O Traces, YCSB etc.) and Synthetic workloads (generated by the simulation engine)

Initialization

- **MQSim components** (FTL, SSD Device, TSU etc.) are represented as **objects** and maintained in a **map**
- The objects in the map are iterated through and corresponding functions are executed
- Each object maintains their own version of **setup_triggers**, **start_simulation** etc.

```
for(std::unordered_map<sim_object_id_type, Sim_Object*>::iterator obj = _ObjectList.begin();
    obj != _ObjectList.end();
    ++obj) {
    if (!obj->second->IsTriggersSetUp()) {
        obj->second->Setup_triggers();
    }
}

for (std::unordered_map<sim_object_id_type, Sim_Object*>::iterator obj = _ObjectList.begin();
    obj != _ObjectList.end();
    ++obj) {
    obj->second->Validate_simulation_config();
}

for (std::unordered_map<sim_object_id_type, Sim_Object*>::iterator obj = _ObjectList.begin();
    obj != _ObjectList.end();
    ++obj) {
    obj->second->Start_simulation();
}
```

Simulation Engine

- Engine.cpp/start_simulation

```
while (true) {
    if (_EventList->Count == 0 || stop) {
        break;
    }

    EventTreeNode* minNode = _EventList->Get_min_node();
    ev = minNode->FirstSimEvent;

    _sim_time = ev->Fire_time;

    while (ev != NULL) {
        if(!ev->Ignore) {
            ev->Target_sim_object->Execute_simulator_event(ev);
        }
        Sim_Event* consumed_event = ev;
        ev = ev->Next_event;
        delete consumed_event;
    }
    _EventList->Remove(minNode);
}
```

The callback function is Execute_simulator_event

Mechanism of MQSim

- A simulator event is registered using the Register_sim_event function

```
Sim_Event* Engine::Register_sim_event(sim_time_type fireTime, Sim_Object* targetObject, void* parameters, int type)
{
        Sim_Event* ev = new Sim_Event(fireTime, targetObject, parameters, type);
        DEBUG("RegisterEvent " << fireTime << " " << targetObject)
        _EventList->Insert_sim_event(ev);
        return ev;
}
```

- Example: An event is registered for sending the Read command and address

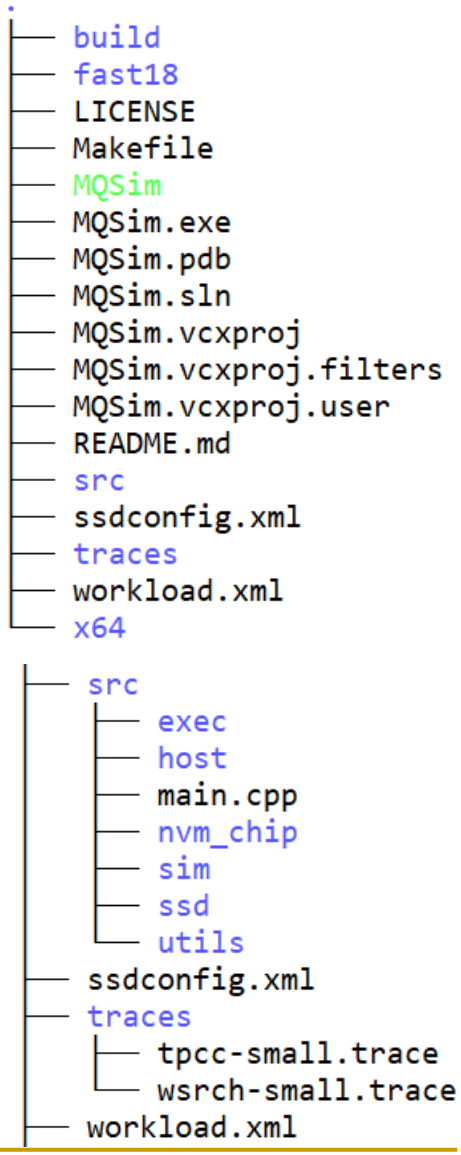
```
if (chipBKE->OngoingDieCMDTransfers.size() == 0) {
    targetChip->StartCMDXfer();
    chipBKE->Status = ChipStatus::CMD_IN;
    chipBKE->Last_transfer_finish_time = Simulator->Time() + suspendTime + target_channel->ReadCommandTime[transaction_list.size()];
    Simulator->Register_sim_event(Simulator->Time() + suspendTime + target_channel->ReadCommandTime[transaction_list.size()], this,
        dieBKE, (int)NVDDR2_SimEventType::READ_CMD_ADDR_TRANSFERRED);
} else {
    dieBKE->DieInterleavedTime = suspendTime + target_channel->ReadCommandTime[transaction_list.size()];
    chipBKE->Last_transfer_finish_time += suspendTime + target_channel->ReadCommandTime[transaction_list.size()];
}
```

Outline

- Internal Components of a Modern SSD
- Introduction to MQSIM
- Mechanism
- Code structure
- Configuring MQSim

Code Structure

- **ssdconfig.xml** – configuration file that contains the preferred SSD configuration
- **workload.xml** – workload definition file
- traces – folder in which the trace files have to be placed
- src/sim – contains the files related to the simulation engine
- src/nvm_chip – code related to NVM chip. Contains files for die, plane, block and page
- src/ssd – contains files related to Flash Translation Layer and Transaction Scheduling Unit
- src/host – files related to trace and synthetic workloads, PCIe etc.



Code Structure

- Code flow
  - Host -> Data Cache -> Address Translation Unit -> TSU -> Flash Controller -> NVM_Chip
- Some important functions:
  - Data Cache Manager: **process_new_user_requests**()
  - Address Mapping Unit: **translate_lpa_to_ppa_and_dispatch**()
  - Transaction Scheduling Unit: **Schedule**()
  - Flash Controller: **send_command_to_chip**()

Outline

- Internal Components of a Modern SSD
- Introduction to MQSIM
- Mechanism
- Code structure
- Configuring MQSim

SSD Configuration File (ssdconfig.xml)

- Host parameters
  - **PCIe_Lane_Bandwidth:** the PCIe bandwidth per lane in GB/s
  - **PCIe_Lane_Count:** the number of PCIe lanes
- Device Parameters
  - **HostInterface_Type:** the type of host interface. Range = {NVME, SATA}
  - **IO_Queue_Depth:** the length of the host-side I/O queue.
  - **Data_Cache_Capacity:** the size of the DRAM data cache in bytes
  - **Ideal_Mapping_Table:** if mapping is ideal, all the mapping entries are found in the DRAM and there is no need to read mapping entries from flash
  - **Transaction_Scheduling_Policy:** the transaction scheduling policy that is used in the SSD back end. Range = {OUT_OF_ORDER as defined in the Sprinkler (Jung et.al., HPCA 2014), PRIORITY_OUT_OF_ORDER which implements OUT_OF_ORDER and NVMe priorities}

SSD Configuration File (ssdconfig.xml)

- Device Parameters
  - **Flash_Channel_Count:** the number of flash channels in the SSD back end
  - **Flash_Channel_Width:** the width of each flash channel in byte
  - **Channel_Transfer_Rate:** the transfer rate of flash channels in the SSD back end in MT/s
  - **Chip_No_Per_Channel:** the number of flash chips attached to each channel in the SSD back end
- NAND Parameters
  - **Flash_Technology:** Range = {SLC, MLC, TLC}.
  - Page read latency for LSB, CSB and MSB pages (in nanoseconds)
  - Page program latency (in nanoseconds)
  - **Block_Erase_Latency:** erase latency in nanoseconds
  - **Block_PE_Cycles_Limit:** the PE limit of each flash block

SSD Configuration File (ssdconfig.xml)

- NAND Parameters
  - **Die_No_Per_Chip:** the number of dies in each flash chip
  - **Plane_No_Per_Die:** the number of planes in each die
  - **Block_No_Per_Plane:** the number of flash blocks in each plane
  - **Page_No_Per_Block:** the number of physical pages in each flash block
  - **Page_Capacity:** the size of each physical flash page in bytes
  - **Page_Metadata_Capacity:** the size of the metadata area of each physical flash page in bytes

Trace-based workload configuration

```
<IO_Scenario>
    <IO_Flow_Parameter_Set_Trace_Based>
        <Priority_Class>HIGH</Priority_Class>
        <Device_Level_Data_Caching_Mode>WRITE_CACHE</Device_Level_Data_Caching_Mode>
        <Channel_IDs>0,1,2,3,4,5,6,7</Channel_IDs>
        <Chip_IDs>0,1,2,3</Chip_IDs>
        <Die_IDs>0,1</Die_IDs>
        <Plane_IDs>0,1</Plane_IDs>
        <Initial_Occupancy_Percentage>70</Initial_Occupancy_Percentage>
        <File_Path>traces/tpcc-small.trace</File_Path>
        <Percentage_To_Be_Executed>100</Percentage_To_Be_Executed>
        <Relay_Count>1</Relay_Count>
        <Time_Unit>NANOSECOND</Time_Unit>
    </IO_Flow_Parameter_Set_Trace_Based>
</IO_Scenario>
```

Synthetic workload configuration

```
<IO_Flow_Parameter_Set_Synthetic>
    <Priority_Class>HIGH</Priority_Class>
    <Device_Level_Data_Caching_Mode>WRITE_CACHE</Device_Level_Data_Caching_Mode>
    <Channel_IDs>0,1,2,3,4,5,6,7</Channel_IDs>
    <Chip_IDs>0,1,2,3</Chip_IDs>
    <Die_IDs>0,1</Die_IDs>
    <Plane_IDs>0,1</Plane_IDs>
    <Initial_Occupancy_Percentage>75</Initial_Occupancy_Percentage>
    <Working_Set_Percentage>50</Working_Set_Percentage>
    <Synthetic_Generator_Type>QUEUE_DEPTH</Synthetic_Generator_Type>
    <Read_Percentage>100</Read_Percentage>
    <Address_Distribution>RANDOM_UNIFORM</Address_Distribution>
    <Percentage_of_Hot_Region>0</Percentage_of_Hot_Region>
    <Generated_Aligned_Addresses>true</Generated_Aligned_Addresses>
    <Address_Alignment_Unit>16</Address_Alignment_Unit>
    <Request_Size_Distribution>FIXED</Request_Size_Distribution>
    <Average_Request_Size>8</Average_Request_Size>
    <Variance_Request_Size>0</Variance_Request_Size>
    <Seed>6533</Seed>
    <Average_No_of_Reqs_in_Queue>16</Average_No_of_Reqs_in_Queue>
    <Intensity>32768</Intensity>
    <Stop_Time>10000000000</Stop_Time>
    <Total_Requests_To_Generate>0</Total_Requests_To_Generate>
</IO_Flow_Parameter_Set_Synthetic>
```

MQSim Output File

```
<Host>
    <Host.IO_Flow>
        <Name>Host.IO_Flow.Trace.traces/tpcc-small.trace</Name>
        <Request_Count>6999</Request_Count>
        <Read_Request_Count>4381</Read_Request_Count>
        <Write_Request_Count>2618</Write_Request_Count>
        <IOPS>6999.000000</IOPS>
        <IOPS_Read>4381.000000</IOPS_Read>
        <IOPS_Write>2618.000000</IOPS_Write>
        <Bytes_Transferred>59718656.000000</Bytes_Transferred>
        <Bytes_Transferred_Read>36315136.000000</Bytes_Transferred_Read>
        <Bytes_Transferred_Write>23403520.000000</Bytes_Transferred_Write>
        <Bandwidth>59718656.000000</Bandwidth>
        <Bandwidth_Read>36315136.000000</Bandwidth_Read>
        <Bandwidth_Write>23403520.000000</Bandwidth_Write>
        <Device_Response_Time>3458</Device_Response_Time>
        <Min_Device_Response_Time>5</Min_Device_Response_Time>
        <Max_Device_Response_Time>18316</Max_Device_Response_Time>
        <End_to_End_Request_Delay>3458</End_to_End_Request_Delay>
        <Min_End_to_End_Request_Delay>5</Min_End_to_End_Request_Delay>
        <Max_End_to_End_Request_Delay>18316</Max_End_to_End_Request_Delay>
    </Host.IO_Flow>
</Host>
```

MQSim usage

- Linux
  - $ make
  - $ ./MQSim -i <SSD Configuration File> -w <Workload Definition File>
- Windows
  - Open the MQSim.sln solution file in MS Visual Studio 2017 or later.
  - Set the Solution Configuration to Release (it is set to Debug by default).
  - Compile the solution.
  - Run the generated executable file (e.g., MQSim.exe) either in command line mode or by clicking the MS Visual Studio run button. Please specify the paths to the files containing the 1) SSD configurations, and 2) workload definitions.
  - $ MQSim.exe -i <SSD Configuration File> -w <Workload Definition File>

P&S Modern SSDs

Introduction to MQSIM

Rakesh Nadig
Dr. Jisung Park
Prof. Onur Mutlu
ETH Zürich

Spring 2022

8th April 2022

SAFARI

BACKUP SLIDES

High-Performance Steady-State Model

- SSDs should be evaluated in steady state
  - *Fresh, out-of-the-box* (FOB) device **unlikely to perform garbage collection**
  - **Write cache not warmed up** for an FOB device
- Many previous SSD studies incorrectly simulate FOB devices
- Difficult to reach steady state in most simulators
  - **Very slow** (e.g., SSDSim execution time increases by up to 80x)
  - **Widely-used traces aren't large enough for proper warm-up**

Exec/

```
yunxin@copper:~/.../src$ tree exec/ -L 1
exec/
├── Device_Parameter_Set.cpp
├── Device_Parameter_Set.h
├── Execution_Parameter_Set.cpp
├── Execution_Parameter_Set.h
├── Flash_Parameter_Set.cpp
├── Flash_Parameter_Set.h
├── Host_Parameter_Set.cpp
├── Host_Parameter_Set.h
├── Host_System.cpp
├── Host_System.h
├── IO_Flow_Parameter_Set.cpp
├── IO_Flow_Parameter_Set.h
├── Parameter_Set_Base.h
├── SSD_Device.cpp
└── SSD_Device.h
```

- exec/: most of the files here are easy to read and understand, treat it as a reference book, not something you need to totally understand at a first glance.

Sim/

```
yunxin@copper:~/.../src$ tree sim/ -L 1
sim/
├── Engine.cpp
├── Engine.h
├── EventTree.cpp
├── EventTree.h
├── Sim_Defs.h
├── Sim_Event.h
├── Sim_Object.h
└── Sim_Reporter.h
```

- sim/: most of the files here are also easy to read and understand, some simulator related definitions could also appear here in Sim_Defs.h.
- The most important feature of this subdirectory is implementing the RB tree mentioned above.

Host/

```
yunxin@copper:~/.../src$ ls host/
ASCII_Trace_Definition.h  PCIe_Link.cpp
Host_Defs.h               PCIe_Link.h
Host_IO_Request.h         PCIe_Message.h
IO_Flow_Base.cpp          PCIe_Root_Complex.cpp
IO_Flow_Base.h            PCIe_Root_Complex.h
IO_Flow_Synthetic.cpp     PCIe_Switch.cpp
IO_Flow_Synthetic.h       PCIe_Switch.h
IO_Flow_Trace_Based.cpp   SATA_HBA.cpp
IO_Flow_Trace_Based.h     SATA_HBA.h
```

- host/: Definitions of host related objects.
- IO_Flow_* is used to generate request, can be ignored as of now.
- Note the order: Host -> PCIe_Root_Complex -> PCIe_Link -> PCIe_Switch

SSD/

- **ssd/**: This is the place where we simulate a SSD(at the device level, not the flash chip level).
- The most important subfolder you will want to explore
- What does a SSD compose of?

```
SSD_Device(Device_Parameter_Set* parameters, std::vector<IO_Flow_Parameter_Set*>* io_flows);
~SSD_Device();
bool Preconditioning_required;
NVM::NVM_Type Memory_Type;
SSD_Components::Host_Interface_Base *Host_interface;
SSD_Components::Data_Cache_Manager_Base *Cache_manager;
SSD_Components::NVM_Firmware* Firmware;
SSD_Components::NVM_PHY_Base* PHY;
std::vector<SSD_Components::NVM_Channel_Base*> Channels;
void Report_results_in_XML(std::string name_prefix, Utils::XmlWriter& xmlwriter);
unsigned int Get_no_of_LHAs_in_an_NVM_write_unit();
```

SSD/

- What does a SSD compose of?
  - Host_Interface: interact between the host.
  - Cache_Manager: manager the cache
  - Firmware: FTL(SSD controller)
    - TSU: Transaction Scheduling Unit
    - Flash_Block_Manager
    - Address_Mapping_Unit
    - GC_and_WL_unit: unit for garbage collection and wear leveling
  - PHY: channel controller
  - Actual Channels

NVM_CHIP/

- Data Structures for hierarchical nvm_chip such as page, block, die, plane…
- Most are easily to read and understand.

```
yunxin@copper:~/.../src$ tree nvm_chip/
nvm_chip/
├── flash_memory
│   ├── Block.cpp
│   ├── Block.h
│   ├── Die.cpp
│   ├── Die.h
│   ├── Flash_Chip.cpp
│   ├── Flash_Chip.h
│   ├── Flash_Command.h
│   ├── FlashTypes.h
│   ├── Page.h
│   ├── Physical_Page_Address.cpp
│   ├── Physical_Page_Address.h
│   ├── Plane.cpp
│   └── Plane.h
├── NVM_Chip.h
├── NVM_Memory_Address.h
└── NVM_Types.h
```

In summary

- If you want to find definitions, go to exec/ or sim/Sim_Defs.h or Ssd/Ssd_Defs.h or Host/Host_Defs.h
- SSD is composed of the following:
  - FTL
    - TSU, AMU, GC_WL, FBM
  - Cache_Manager
  - Channels
  - Channel Controller
  - Host_Interface

Important workload definitions

- Parameter involved in defining an IO flow
- Trace Based:
  - Channel ID: The ID of channels that the trace used
- Synthetic Workload:
  - Synthetic Generator type: generate the IO flow based on the bandwidth on based on the IO queue depth
  - Address distribution: The distribution of the address the request goes to.
  - Request_Size_Distribution: The distribution of the request size it generated
  - Bandwidth: Average bandwidth of the IO requests generated(for the bandwidth mode)
  - Average IO queue depth: Average number of request in the host-side queues(for the IO queue depth mode)