Architecture and RISC-V ISA Extension Supporting Asynchronous and Flexible Parallel Far Memory Access

Songyue Wang, Luming Wang, Tianyue Lu, Mingyu Chen

ICT, CAS

University of Chinese Academy of Sciences

June 19, 2022 @CARRV'22, co-located with ISCA'22, New York City

中国科学院大学 University of Chinese Academy of Sciences

中国科学院计算技术研究所 INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES

先进计算机系统研究中心 Center for Advanced Computer Systems

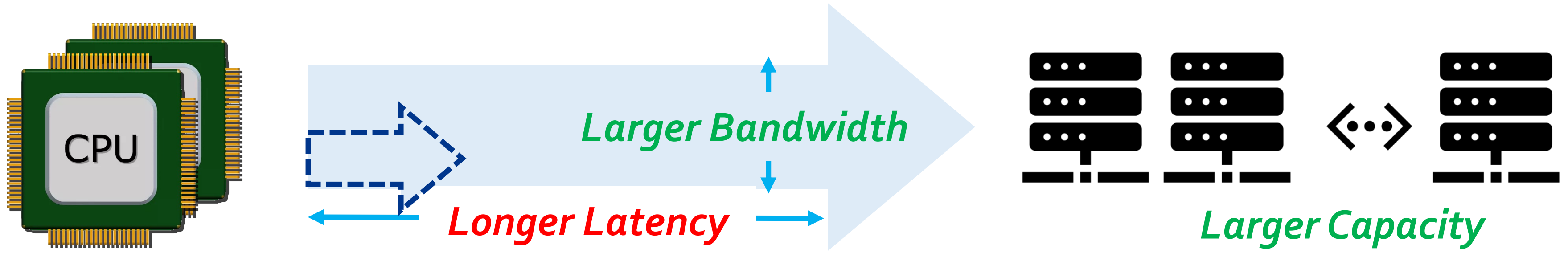
Far Memory Technology (FMT)

New Interconnect Techologies and Protocols

NVM Main Memory:
3D Xpoint/Optane

New Medium besides DRAM

- **Allow one host to access more memory resources**
  - **Large aggregated capacity and bandwidth ☺**
  - **Widely distributed latency ☹ (reach 300ns~5μs)**
- **Leave a challenge for CPU to fully utilize the abundant memory resources!**

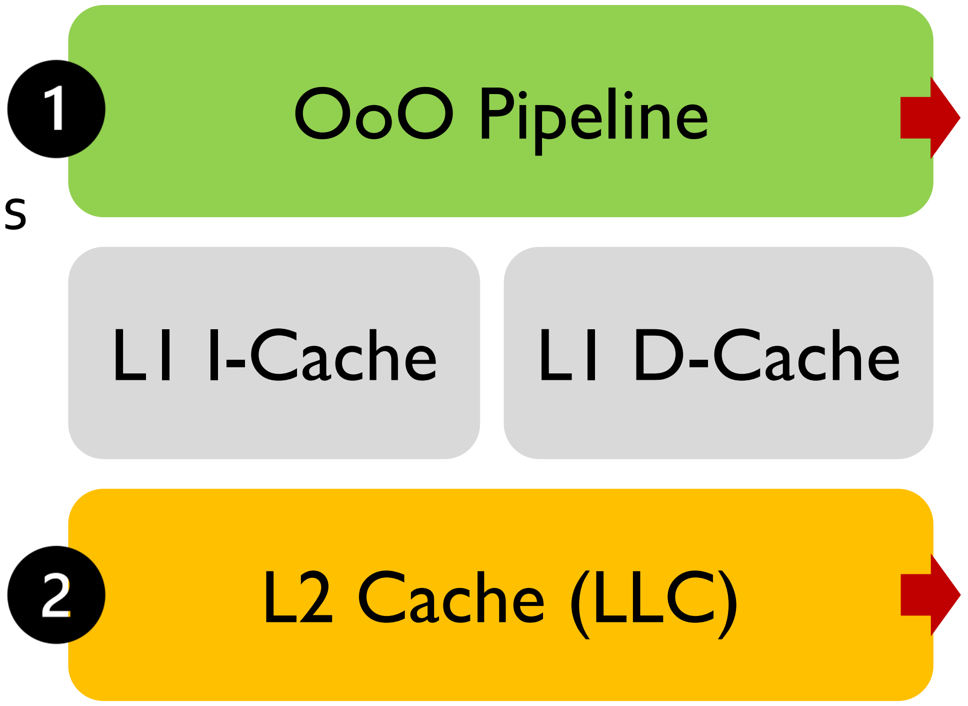


Modern CPU cannot Reach High MLP

- Additional **64ns** latency → **30%+** slowdown!
  - Especially applications hich **benefit from FMT**! (such as Redis, GAPBS and Spark)
- Modern CPUs **implicitly** boost MLP:

OoO Execution
Dynamical scheduling
more load/store instructions

Non-Blocking Cache
Handling several miss
memory requests



ROB & Physical Register Files: **~200** entries

Stall due to synchronous waiting of load/store!

MSHR: only **12** in single Intel Skylake!

Modern CPU cannot Reach High MLP

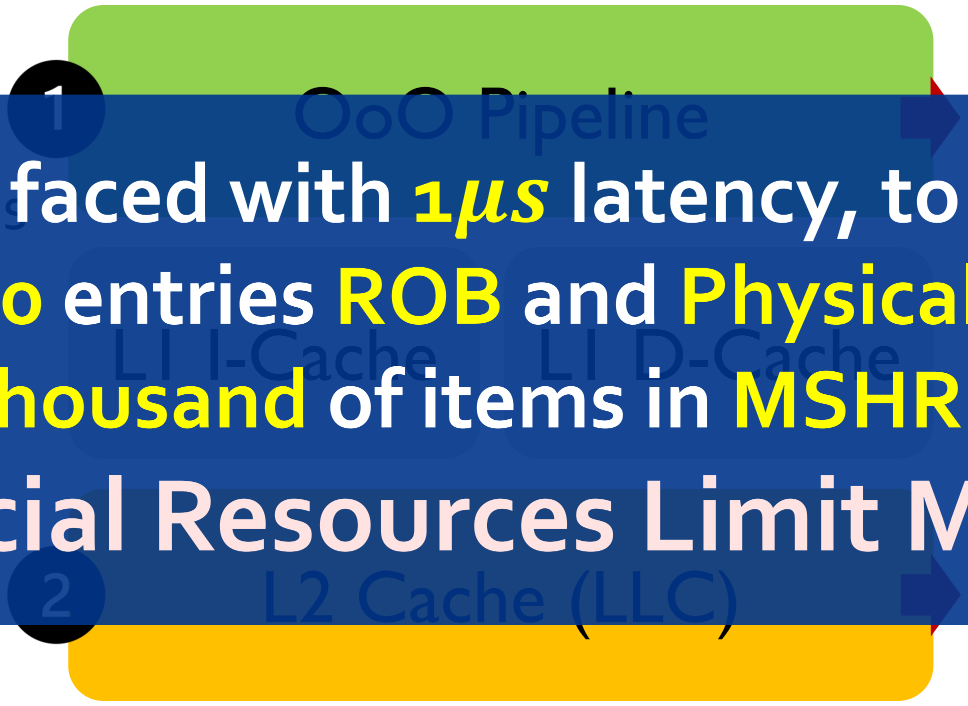
- Additional 64ns latency → 30%+ slowdown!
  - Especially applications hich **benefit from FMT**! (such as Redis, GAPBS and Spark)
- Modern CPUs **implicitly** boost MLP:

OoO Execution

Dynamical scheduling more load/store instructions

Non-Blocking Cache

Handling several miss memory requests



ROB & Physical Register Files: ~200 entries

Stall due to synchronous waiting of load/store!

MSHR: only **12** in single Intel Skylake!

A 2GHz CPU faced with 1μs latency, to avoid stall:
~2000 entries ROB and Physical RF
thousand of items in MSHR

Criticial Resources Limit MLP!

Key Observation

The bottleneck to boost MLP

- **ISA**
  - ***Load & store*** are **synchronous** and **blocking**
- **Microarchitecture**
  - **Request** and **response** are **coupling**
- **Storage**
  - **Limited MSHR** for handling **status**
  - **Limited Physical RF** for **data storage**
- **Semantic**
  - **Fixed** Cache **access length** and strategy

Our Goal

Our Goal: Supporting Massive Parallel and Flexible Memory Access

- **ISA**
  - *Load & store* are synchronous and blocking

Asynchronous Memory Access ISA Extension for RISC-V (AME)

- **Microarchitecture**
  - Request and response are coupling
- **Storage**
  - Limited MSHR for handling status
  - Limited Physical RF for data storage
- **Semantic**
  - Fixed Cache access length and strategy

Our Goal

Our Goal: Supporting Massive Parallel and Flexible Memory Access

- **ISA**
  - *Load & store* are synchronous and blocking

Asynchronous Memory Access ISA Extension for RISC-V (AME)

- **Microarchitecture**
  - Request and response are coupling
- **Storage**
  - Limited MSHR for handling status
  - Limited Physical RF for data storage

Asynchronous Memory Access Unit (AMU)

- **Semantic**
  - Fixed Cache access length and strategy

Our Goal

Our Goal: Supporting Massive Parallel and Flexible Memory Access

- **ISA**
  - *Load & store* are synchronous and blocking
- **Microarchitecture**
  - Request and response are coupling
- **Storage**
  - Limited MSHR for handling status
  - Limited Physical RF for data storage
- **Semantic**
  - Fixed Cache access length and strategy

Asynchronous Memory Access ISA Extension for RISC-V (AME)

Function Interface

Asynchronous Memory Access Unit (AMU)

Variable Granularity

Outline

Background

Asynchronous **M**emory access ISA **E**xtensions (AME)

Programing Model

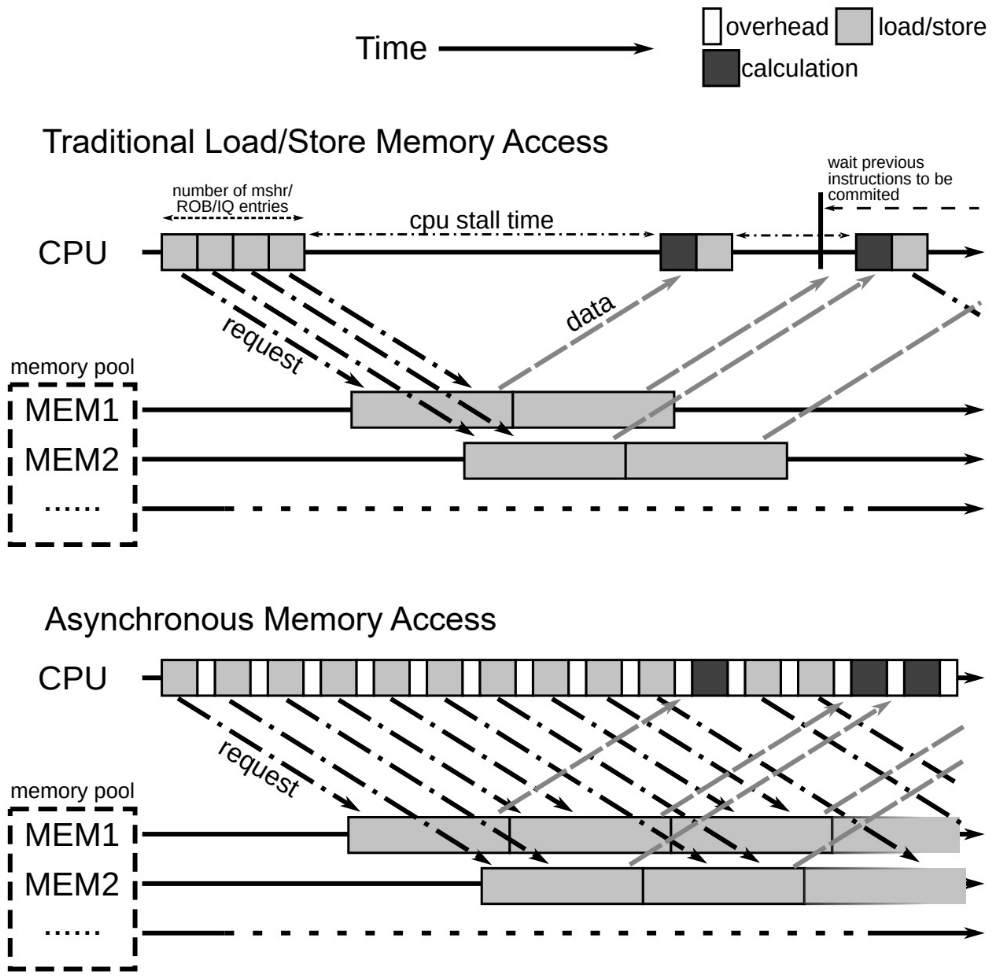
Evaluation

Key Idea 1: Asynchronous memory access ISA (AME)

1 **ISA:** Asynchronous Memory access ISA Extension (AME)

- Core: ***aload*** and ***astore*** instructions

- **Step 1**: Write the request item to ***AMQ***
  - Asynchronous Memory request Queue
- **Step 2**: Commit!



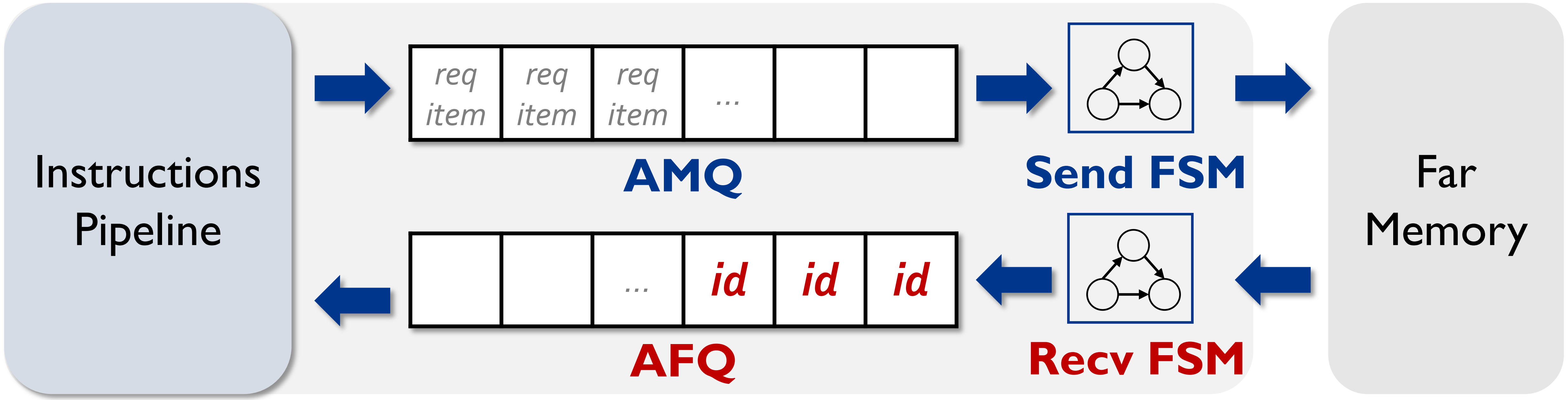
Time
overhead
load/store
calculation
Traditional Load/Store Memory Access
number of mshr/ ROB/IQ entries
cpu stall time
wait previous instructions to be commited
CPU
request
data
memory pool
MEM1
MEM2
......
Asynchronous Memory Access
CPU
request
memory pool
MEM1
MEM2
......

- **Release the pressure of ROB**
- **Allow issuing massive and concurrent memory request from pipeline**

Key Idea 2: Asynchronous Memory Access Unit (AMU)

2 **Microarchitecture: AMU** (2 queues and 2 FSM)

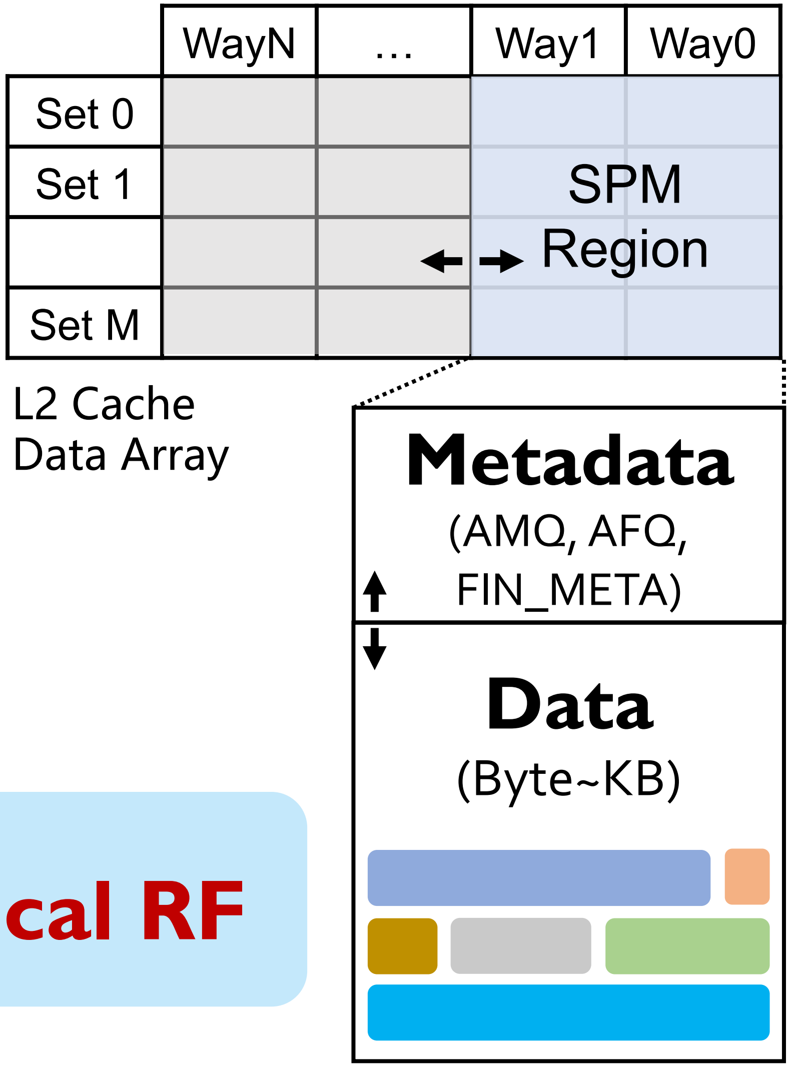
- **An unique ID** for Each request
- Aload/astore commits to **AMQ** (Asynchronous Memory access Request Queue)
- **Send FSM** issues according to items in **AMQ**
- **Recv FSM** pushes response IDs to **AFQ** (Asynchronous Memory access Finish Queue)
- Software retrieves ID from **AFQ**



Key Idea 3: ScratchPad Memory for AMU

3 **Storage:** ScratchPad Memory (SPM)

- Divided from **L2 Cache Data Array (≥128KB)**
- Compatibility: Adjustable by Cache way



- **Metadata Region**
  - AMQ, AFQ and FIN_META
  - Adjustable length

~ MSHR

- **Data Region**
  - For variable granularity memory data

~ Physical RF

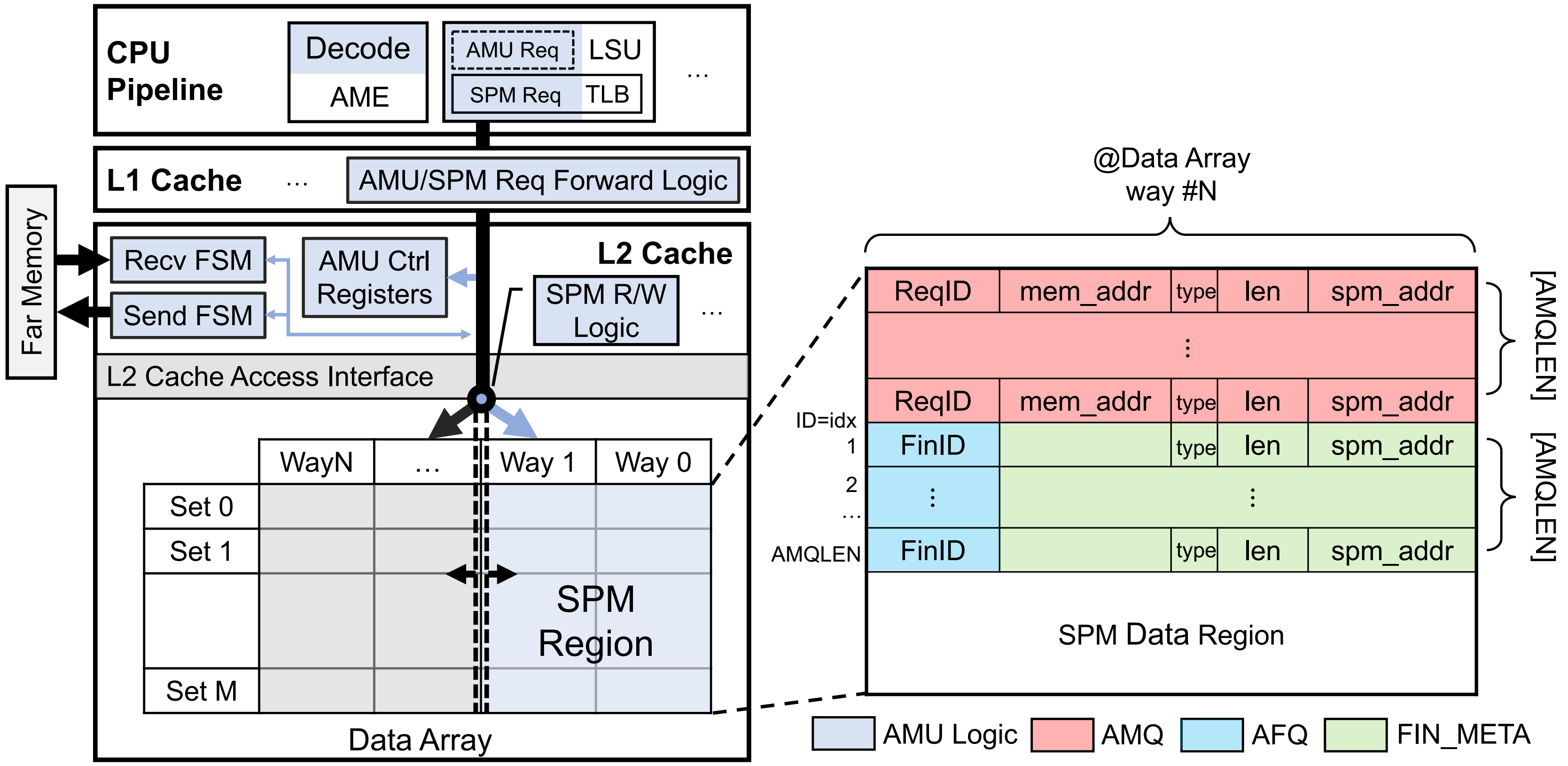
Supporting thousand of outstanding memory access!

AME: Programming Interface of AMU

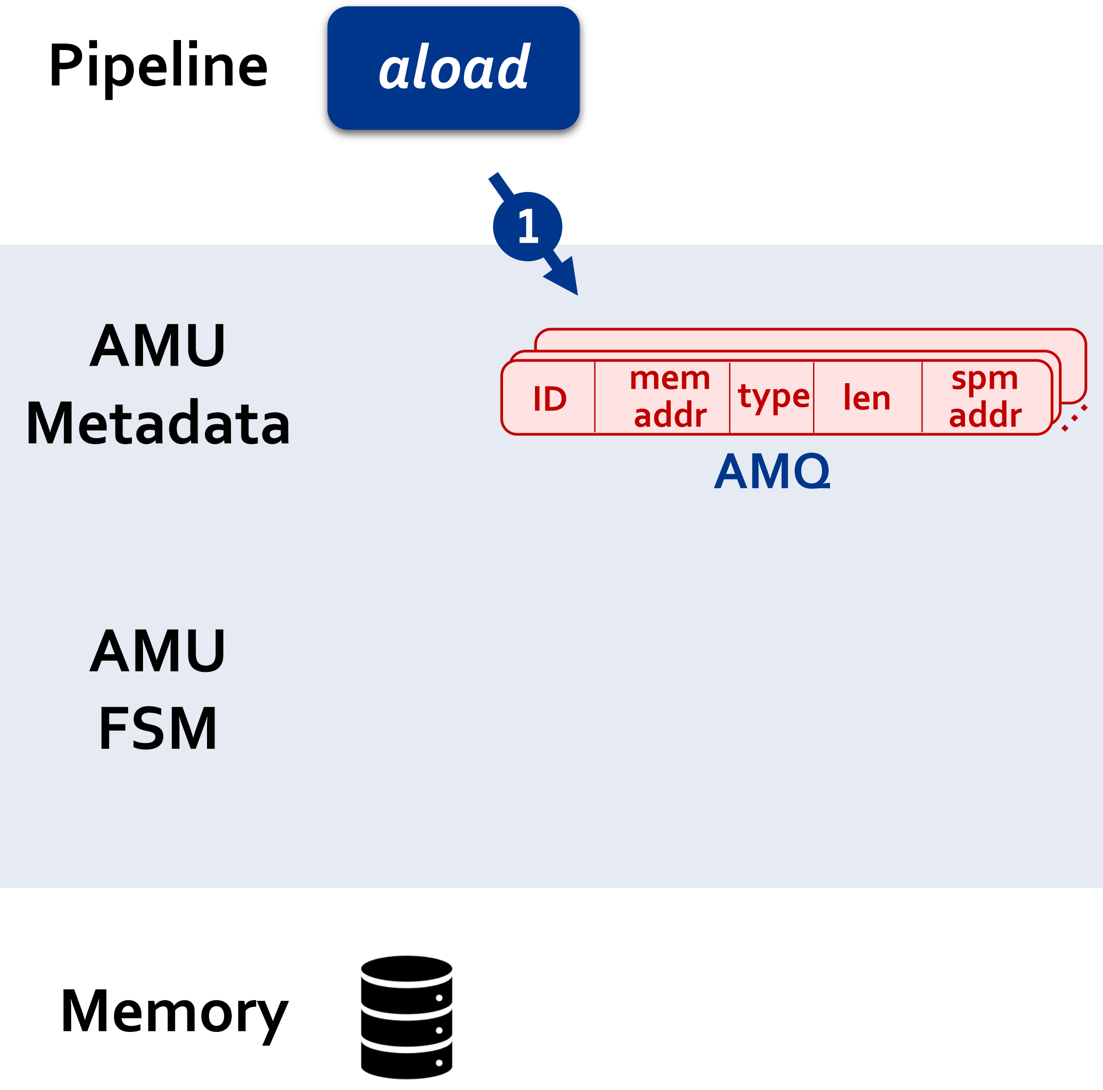
- AME Core Instructions
  - `aload  id, spm_addr, mem_addr` : move Memory → SPM
  - `astore id, spm_addr, mem_addr` : move SPM → Memory
  - `asetid id` : Set ID for the next `aload/astore`
  - `getfin id` : Get a **finished ID** from **AFQ**
- AMU Control Registers
  - `SPMWAY` : Which ways of L2 Cache are continuously occupied by SPM.
  - `RWLEN` : Granularity of accessing memory (8-byte alignment)
  - `AMQLEN` : Length of AMQ (max concurrency of memory access via AMU)

Microarchitecture of AMU

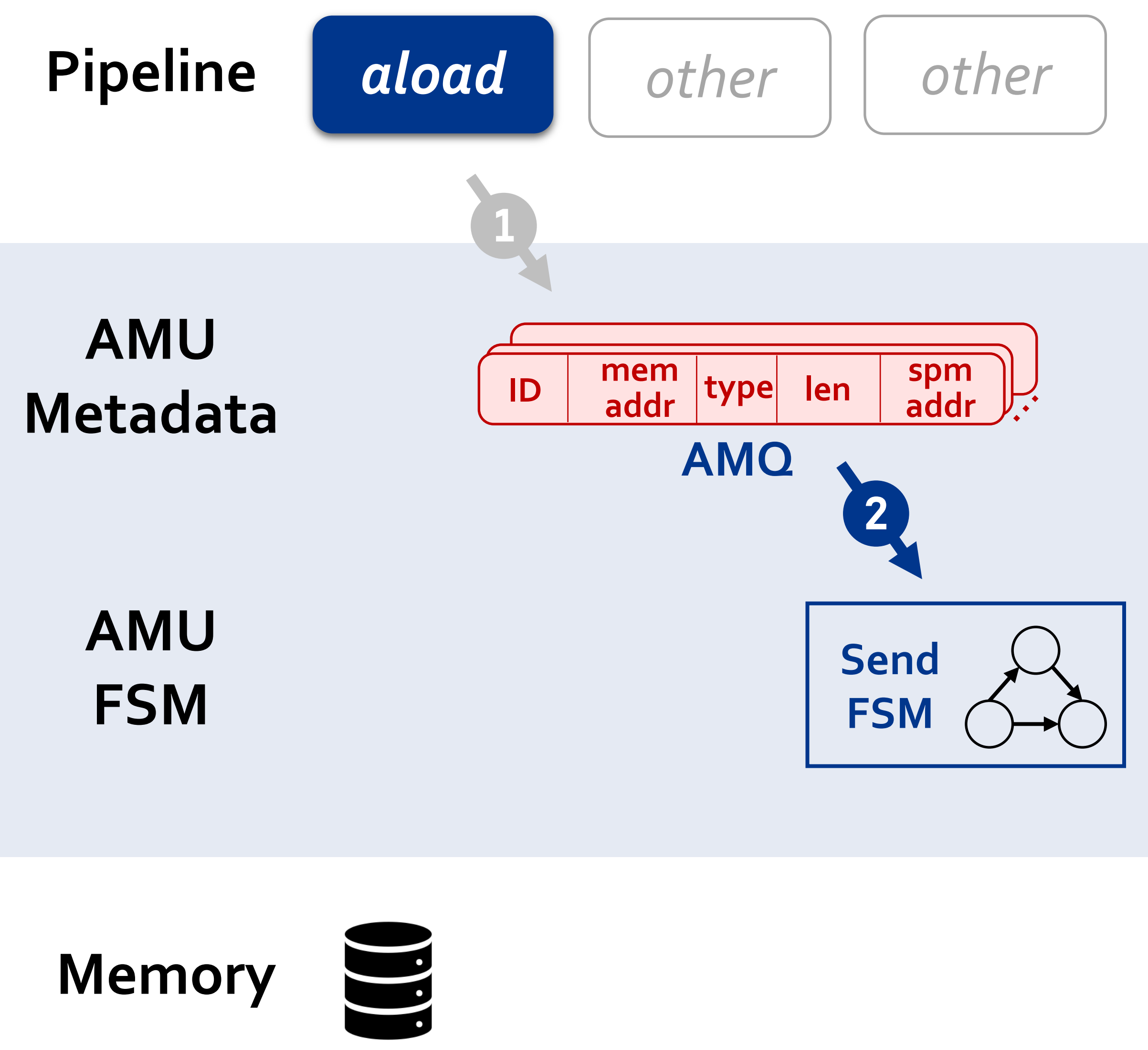
- Pipeline
- Memory Access Path
- L2 Cache
  - Send/Recv FSM
  - AMQ/AFQ/FIN_META



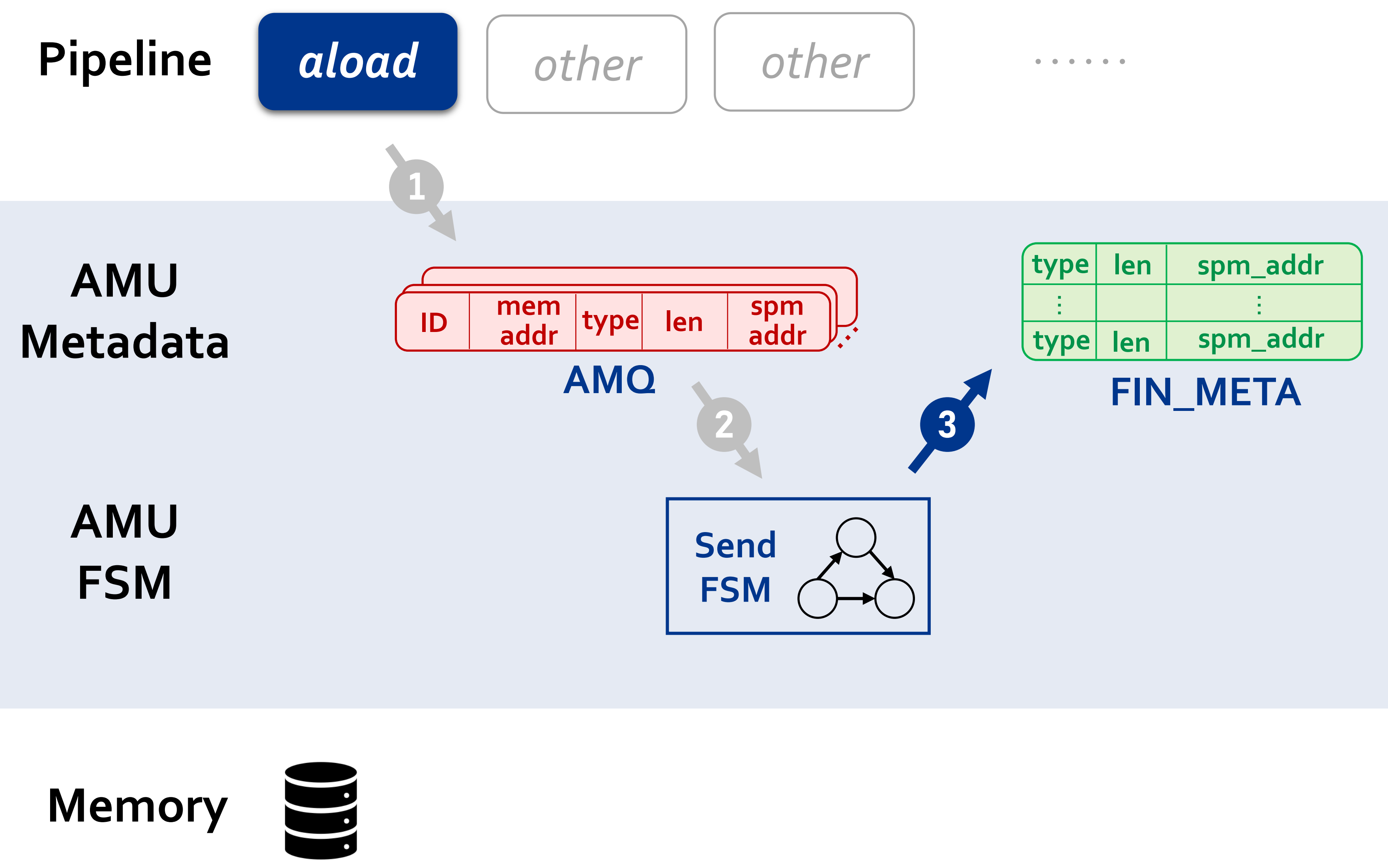
Process of an asynchronous memory access



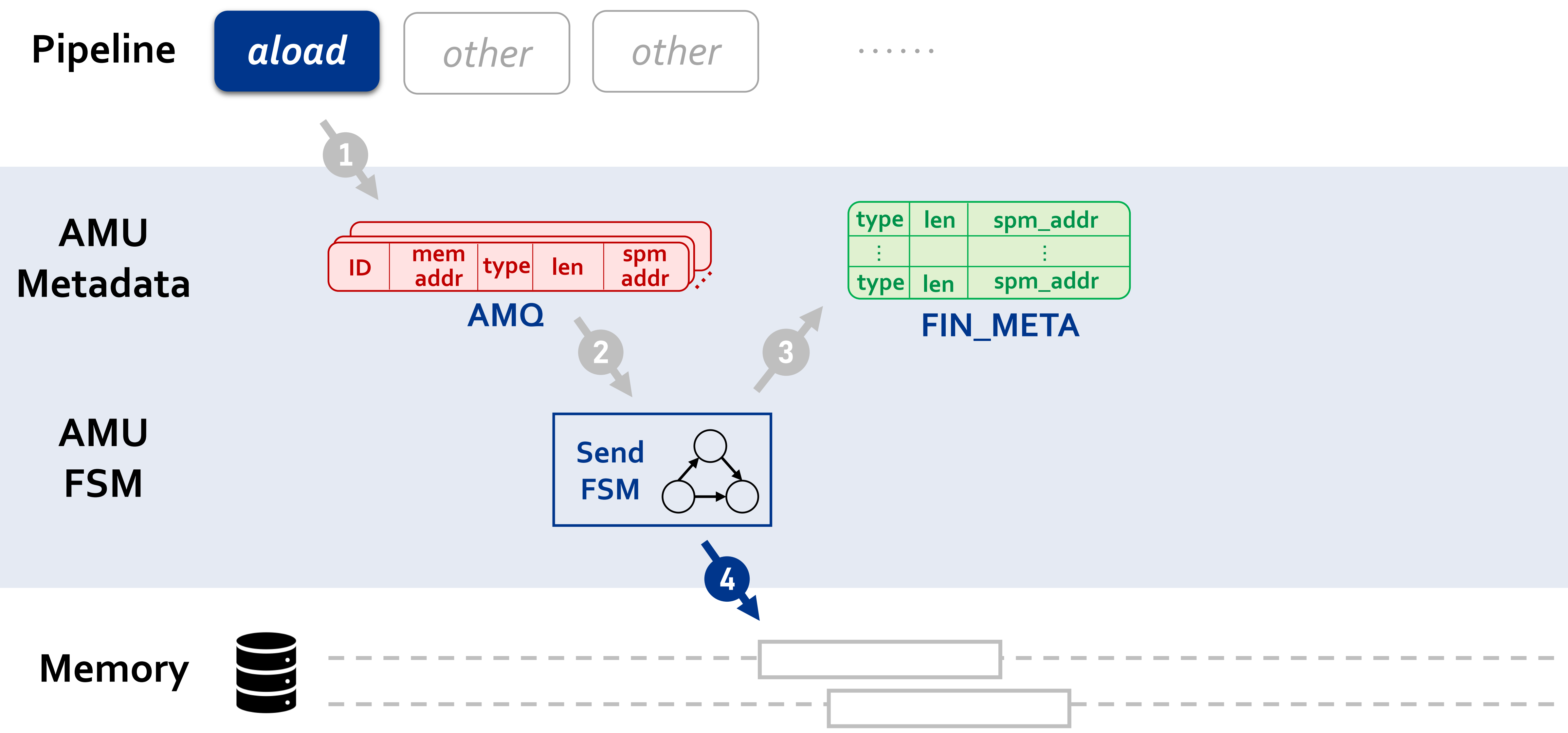
Process of an asynchronous memory access



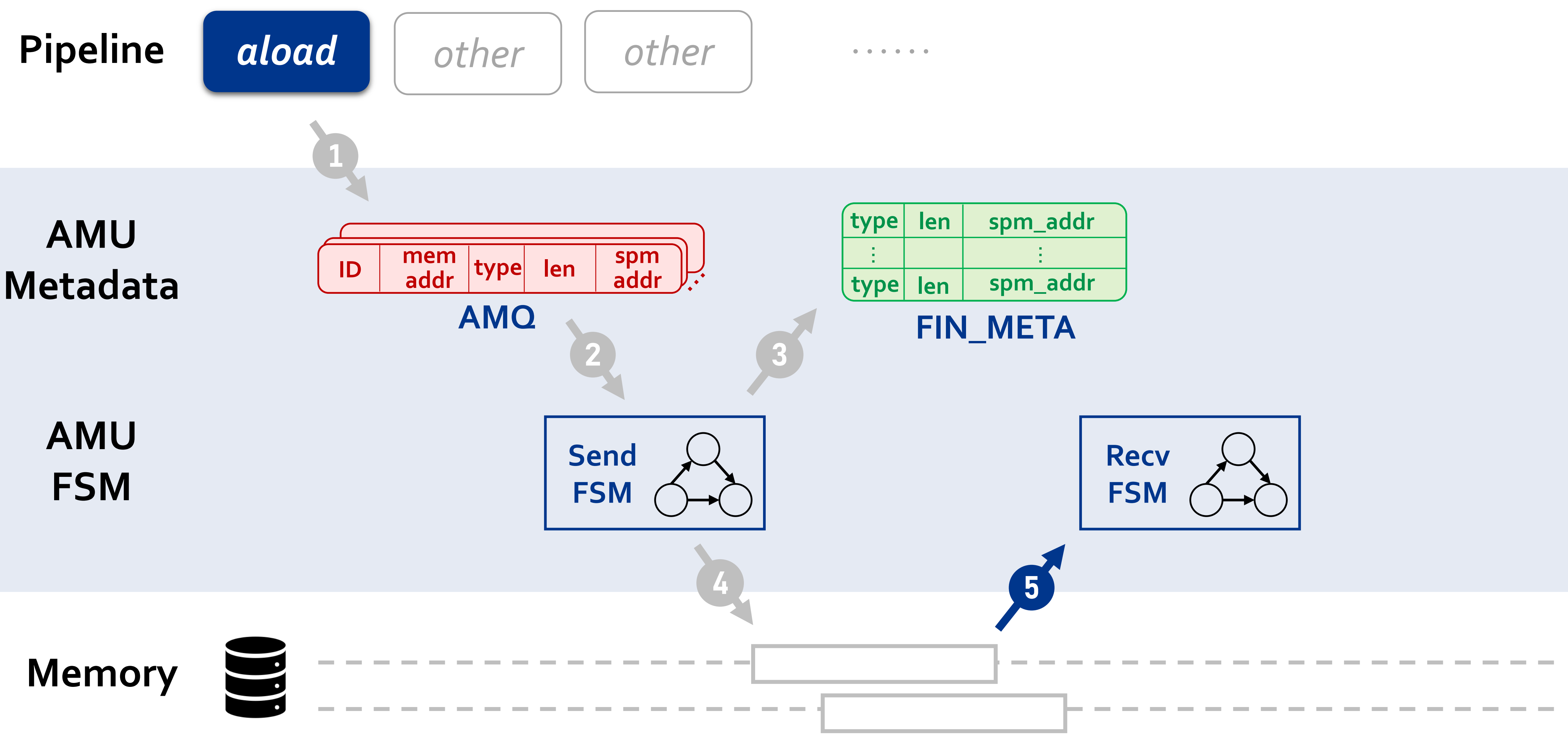
Process of an asynchronous memory access



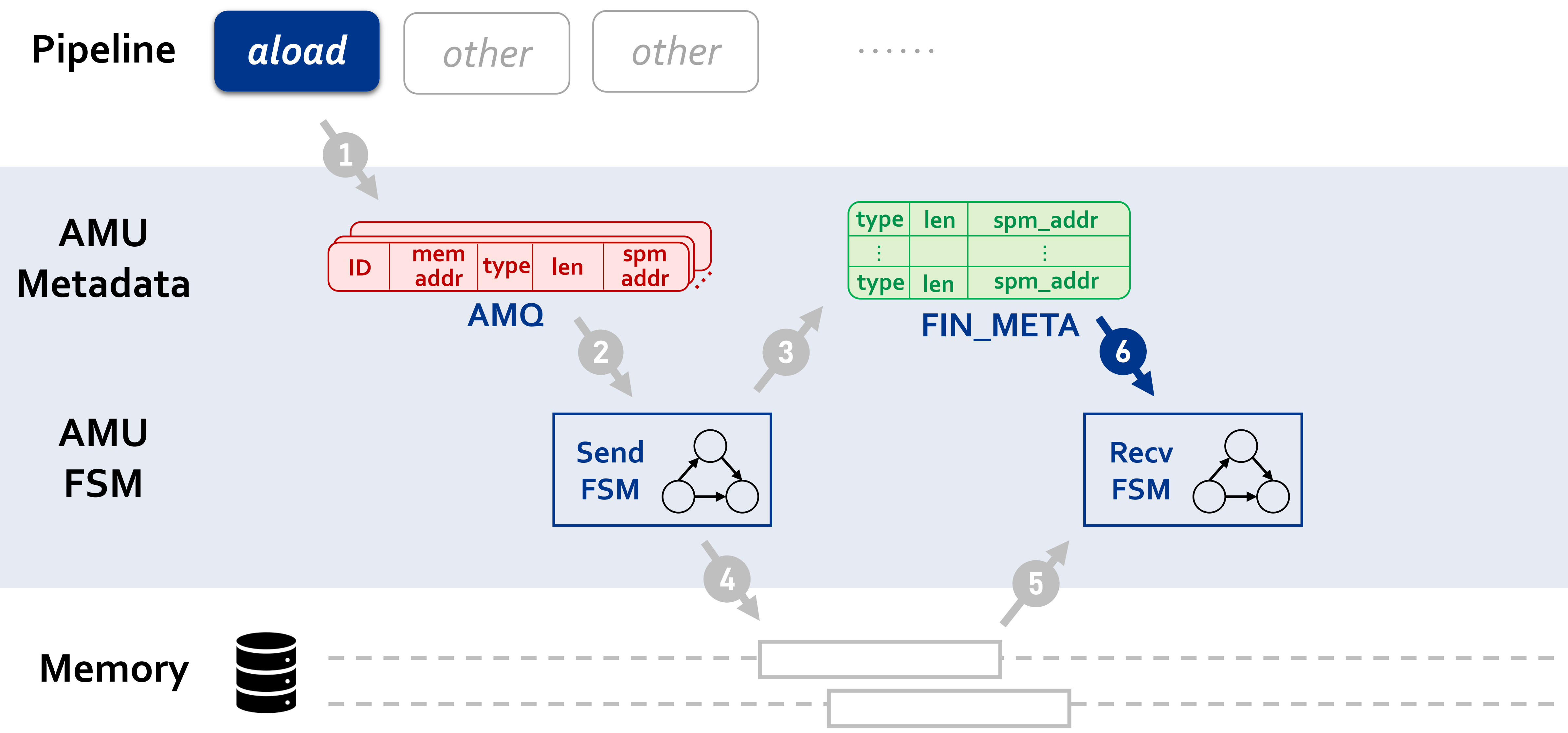
Process of an asynchronous memory access



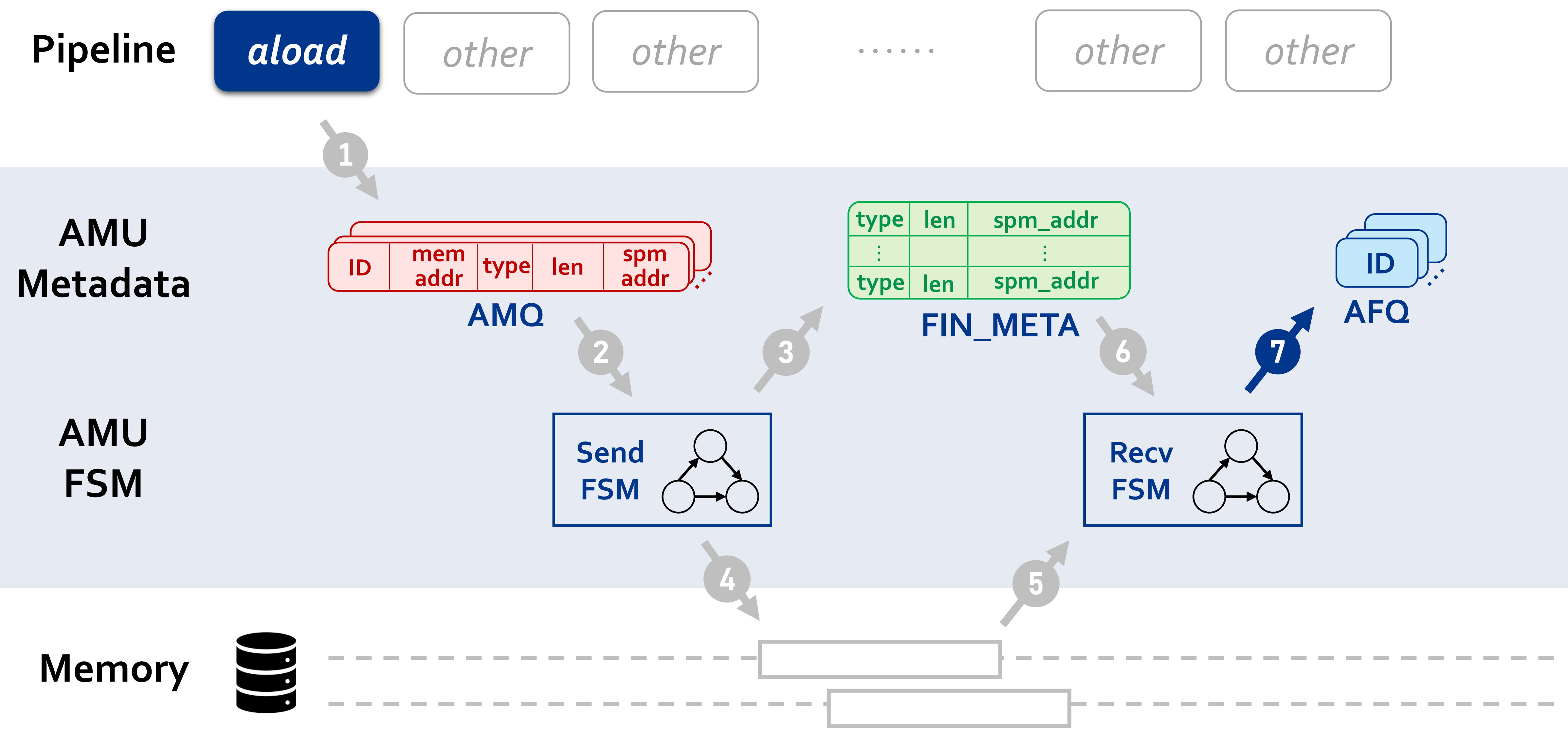
Process of an asynchronous memory access



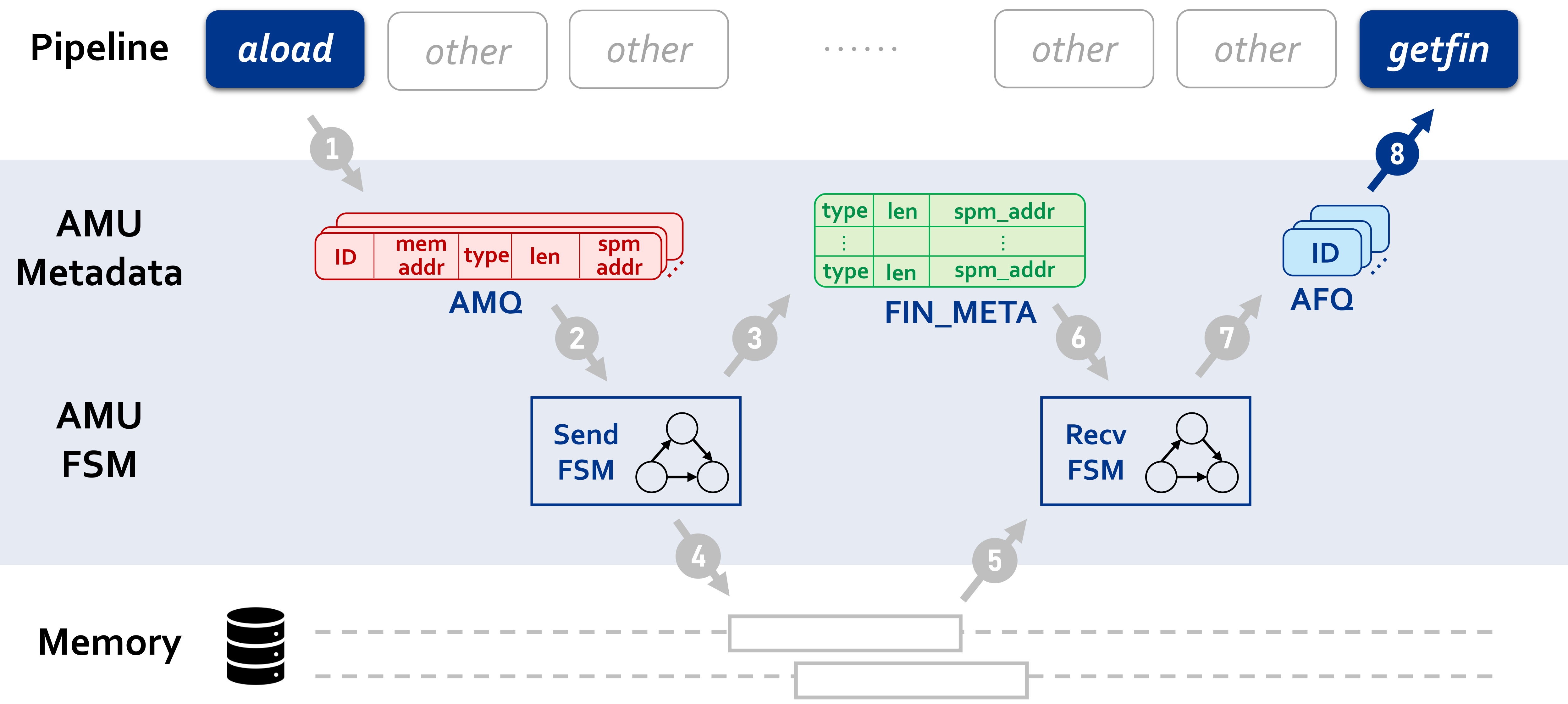
Process of an asynchronous memory access



Process of an asynchronous memory access



Process of an asynchronous memory access



Outline

Basic Paradigm

Step 1: Config AMU

- Max parallelism (length of AMQ)
- Access granularity

```
#define MAX_PARALLELISM 256
int *mem_to_access; // memory to be accessed

// AMU Configuration
acfgwr(MAX_PARALLELISM, AMQLEN);
acfgwr(sizeof(int), RWLEN);

int *spm_data_area = (int *)alloc_spm_addr(sizeof(int));
int id = 1;  // alloc an ID
// issue an aload request
aload(id, spm_data_area, &far_mem_to_access);
// process other
while(id != getfin()) { /* process other */ }
// access SPM via standard load/store
printf("%d\n", *spm_space);
```

Basic Paradigm

Step 1: Config AMU
- Max parallelism (length of AMQ)
- Access granularity

Step 2: Issue an aload/astore request
- Allocate SPM space
- Allocate an ID

```
#define MAX_PARALLELISM 256
int *mem_to_access; // memory to be accessed

// AMU Configuration
acfgwr(MAX_PARALLELISM, AMQLEN);
acfgwr(sizeof(int), RWLEN);

int *spm_data_area = (int *)alloc_spm_addr(sizeof(int));
int id = 1;  // alloc an ID
// issue an aload request
aload(id, spm_data_area, &far_mem_to_access);
// process other
while(id != getfin()) { /* process other */ }
// access SPM via standard load/store
printf("%d\n", *spm_space);
```

Basic Paradigm

Step 1: Config AMU

- Max parallelism (length of AMQ)
- Access granularity

Step 2: Issue an aload/astore request

- Allocate SPM space
- Allocate an ID

Step 3: Wait for finish

- Use *getfin* for checking

```
#define MAX_PARALLELISM 256
int *mem_to_access; // memory to be accessed

// AMU Configuration
acfgwr(MAX_PARALLELISM, AMQLEN);
acfgwr(sizeof(int), RWLEN);

int *spm_data_area = (int *)alloc_spm_addr(sizeof(int));
int id = 1;  // alloc an ID
// issue an aload request
aload(id, spm_data_area, &far_mem_to_access);
// process other
while(id != getfin()) { /* process other */ }
// access SPM via standard load/store
printf("%d\n", *spm_space);
```

Basic Paradigm

Step 1: Config AMU

- Max parallelism (length of AMQ)
- Access granularity

Step 2: Issue an aload/astore request

- Allocate SPM space
- Allocate an ID

Step 3: Wait for finish

- Use *getfin* for checking

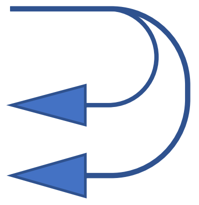
Step 4: Access

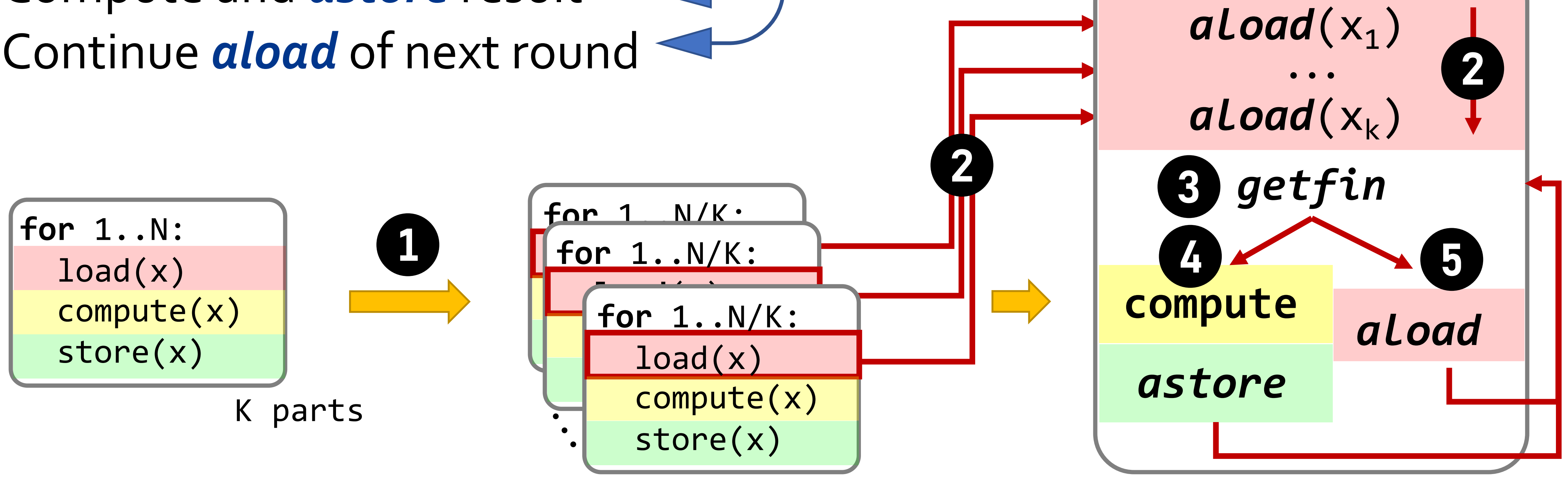
- Via standard load/store.

```
#define MAX_PARALLELISM 256
int *mem_to_access; // memory to be accessed

// AMU Configuration
acfgwr(MAX_PARALLELISM, AMQLEN);
acfgwr(sizeof(int), RWLEN);

int *spm_data_area = (int *)alloc_spm_addr(sizeof(int));
int id = 1;  // alloc an ID
// issue an aload request
aload(id, spm_data_area, &far_mem_to_access);
// process other
while(id != getfin()) { /* process other */ }
// access SPM via standard load/store
printf("%d\n", *spm_space);
```

Issue Massive Requests with AME

- Suitable for **data parallelized process**
  1. Divide into K parts
  2. Issue ***aload*** of all parts in batch
  3. ***Getfin*** to retrieve an finish ID
  4. Compute and ***astore*** result
  5. Continue ***aload*** of next round





CAP: Coroutines for AME Programming

- Idea: AME ~ Linux async I/O
- Based on **C++20 Coroutines**
  - **co_await** for async event
- Features
  - ID & SPM management
  - Lock
  - Scheduler (driven by *getfin*)
- **Efficiency**
  - Similar to **multi-thread programming**
  - Reduce **80% lines of** code
  - No care about scheduler (and *getfin*)!

```
for (int i = 0; i < n; ++i)
  L[i] ^= i;
```

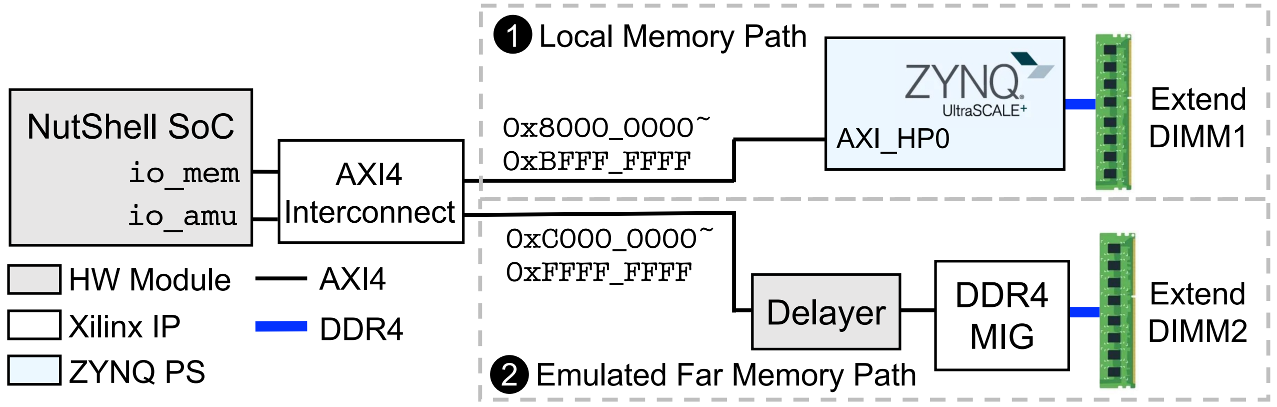
⬇

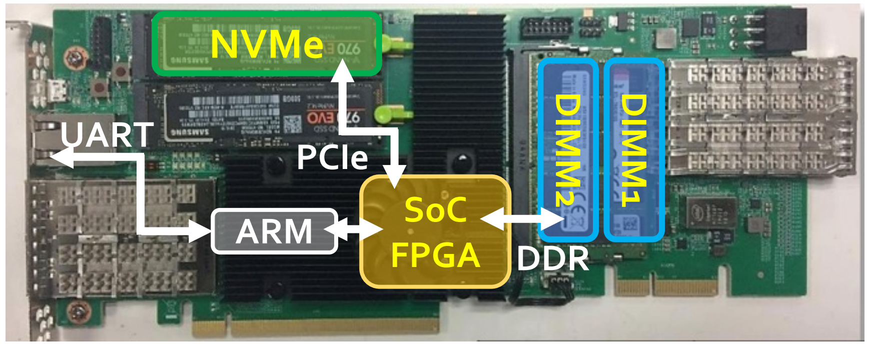
```
template<typename Scheduler>
coro::task<void> update (int idx, int *L,
        int eachNUPDATE, Scheduler &sched){

  int *spm_addr = sched.alloc_spm_addr();

  for (int i = idx; i < idx + eachNUPDATE; ++i) {
    co_await aload_coro(spm_addr, &L[i], sched);
    *spm_addr ^= i;
    co_await astore_coro(spm_addr, &L[i], sched);
  }

  sched.release_spm(spm_addr);
}
```

Outline

FPGA-based Prototype System

- On Xilinx UltraScale+ ZU19EG MPSoC ⟸ **by software on ARM core**
  - ❶ **Local Memory Path**
  - ❷ **Emulated far memory path** with **adjustable access latency**
    - **Real-time MLP observation**
- **NutShell** integrated with AMU
  - **In-order pipelined RISC-V64IMACSU core**: *open-source chip by university (ICT, taped-out)*
  - Boot **Debian 11** on FPGA-based Prototype System **Compatibility of AMU**





Evaluation

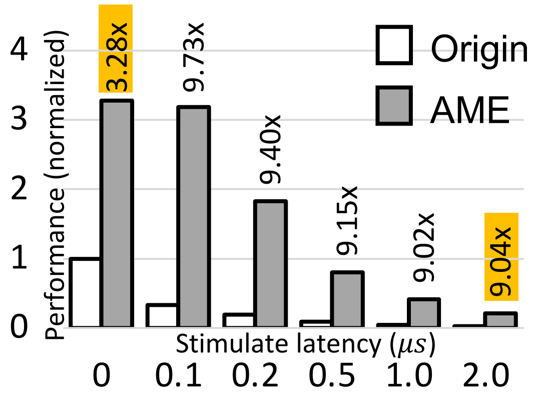
- **Goal:** Answer **3 questions** through **7 benchmarks**
  1. The improvement of memory access performance via AMU
  2. How does AMU accelerate memory access
  3. The acceleration effect of amu on the actual application

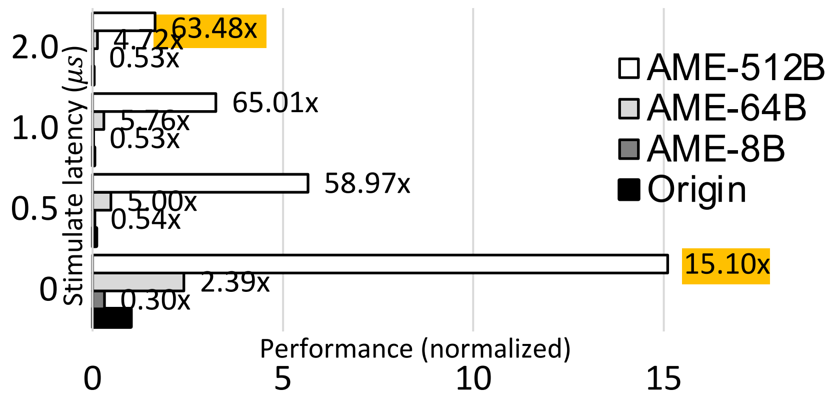
- **Setup**
  - Simulate a 2GHz processor
  - 0~10μs latency on
  - Emulated Far Memory

Benchmark	Abbr.	Type	Footprint	AMQLEN
Random Access	GUPS	Memory access	256MB	256
Sequence Access	SA	Memory access	256MB	256
Hash Join	HJ	Data processing	15.624MB	256
Integer Sort	IS	Data processing	8MB	256
Binary Search	BS	Data processing	97.65MB	256
Hash Table with Hand-over-hand Linked List	HL	Concurrent data structures	2MB	256
Graph500	G500	Graph Computing	16MB	8

Key Results 1: Memory Access Performance

	Local Memory	Far Memory ($2\mu s$ additional latency)
Random Access (GUPS, **8Byte**)	**3.28x**	**9.04x**
Sequential Access (SA, **512Byte**)	**15.10x**	**63.48x**





Key Results 1: Memory Access Performance

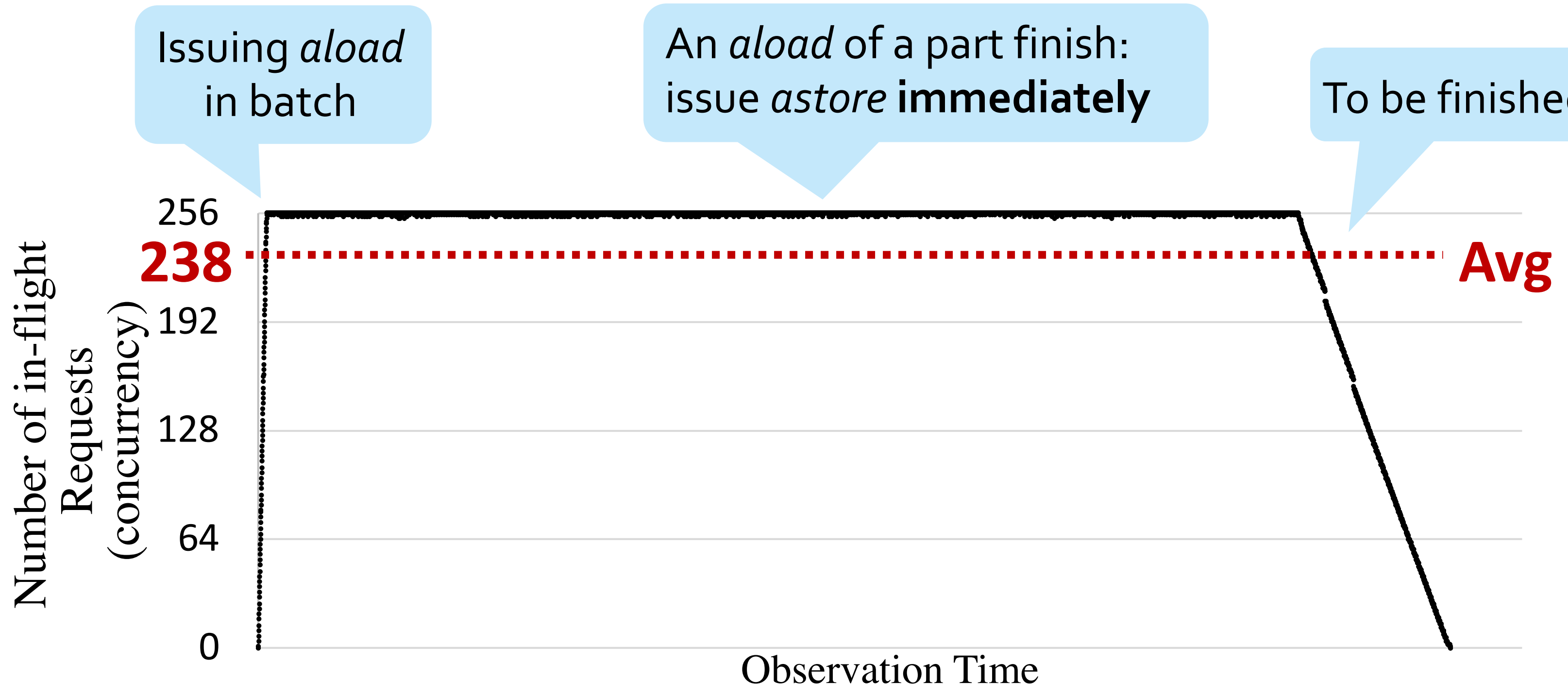
	Local Memory	Far Memory ($2\mu s$ additional latency)
Random Access (GUPS, **8Byte**)	**3.28x**	**9.04x**
Sequential Access (SA, **512Byte**)	**15.10x**	**63.48x**

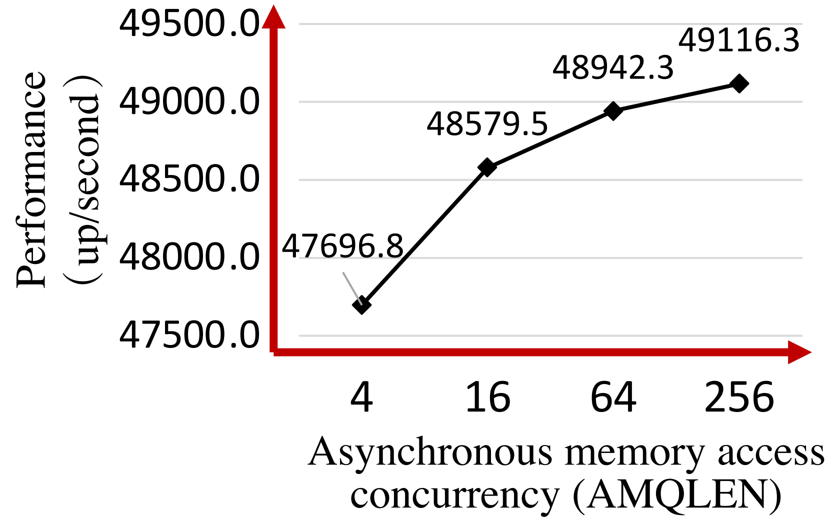
- **High MLP** significantly improves the performance of **memory access**.
- **Variable-grained** memory access better fits program **semantics**.

Fuller and more precise use of bandwidth

Key Results 2: AME brings High MLP

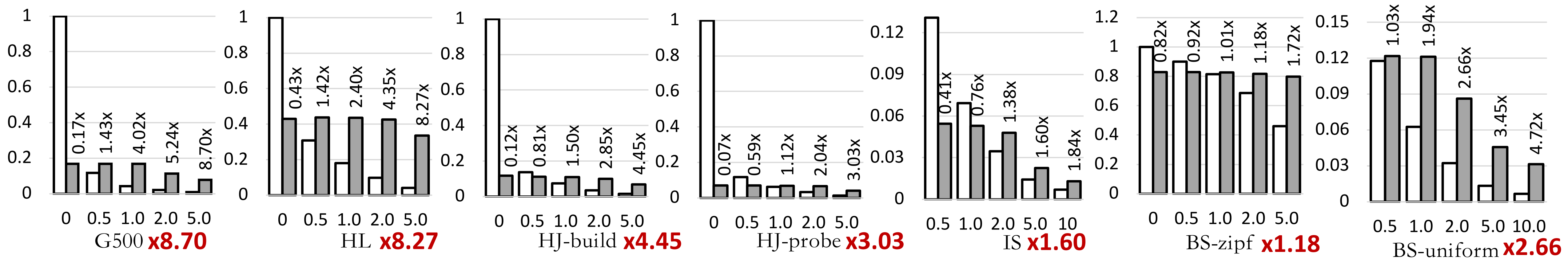
- AMU reaches **238 outstanding requests** at average
  - Workload: GUPS, at 2μs stimulated latency
  - **Larger the concurrency we set, the more performance we get**





Key Results 3: Practical Benchmarks

- For applications:
  - **Reduced latency sensitivity** through high MLP (at glance)
  - **Multiple performance improvements**
- Practical significance:
  - AMU can **improve throughput** and protect concurrency
  - With AMU equipped, modern CPU can improve MLP to accelerate the calculation of popular applications such as graph computing



Q&A

Architecture and RISC-V ISA Extension Supporting Asynchronous and Flexible Parallel Far Memory Access

Songyue Wang, Luming Wang, Tianyue Lu, Mingyu Chen

ICT, CAS

University of Chinese Academy of Sciences

Please contact Songyue Wang (wangsongyue18@mails.ucas.ac.cn) for any concerns.

中国科学院大学
University of Chinese Academy of Sciences

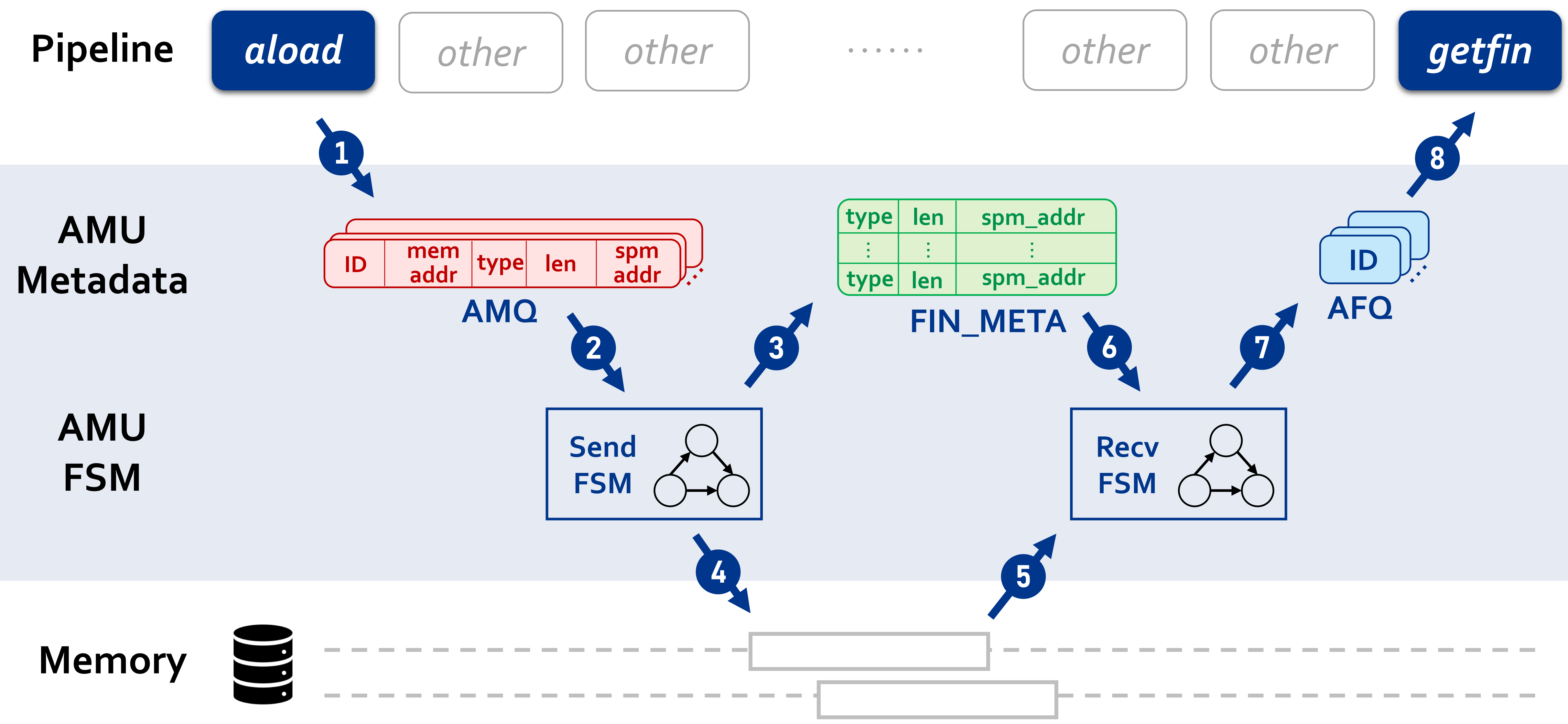
中国科学院计算技术研究所
INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES

先进计算机系统研究中心
Center for Advanced Computer Systems

Conclusion

- **Problem**: Far memory scenarios brings potential **higher bandwidth and higher access latency**.
- **Goal**: Break the limitations of existing processors to improve MLP and mask latency with **high concurrent memory access**.
- **Challenge**:
  - Modern CPUs have little space for handling outstanding memory requests
  - Load/store instructions occupy critical resources in pipeline for a long time
- **Idea**:
  - Asynchronous Memory access ISA Extension (AME)
  - Asynchronous Memory access Unit (AMU) inside processors
- **Key Result**:
  - Average **11x** at 2μs latency for 2GHz CPU
  - MLP reaches 238 running GUPS

Process of an asynchronous memory access


Pipeline
aload
other
other
......
other
other
getfin
1
AMU Metadata
ID
mem addr
type
len
spm addr
AMQ
type
len
spm_addr
type
len
spm_addr
FIN_META
8
ID
AFQ
2
3
6
7
AMU FSM
Send FSM
Recv FSM
4
5
Memory