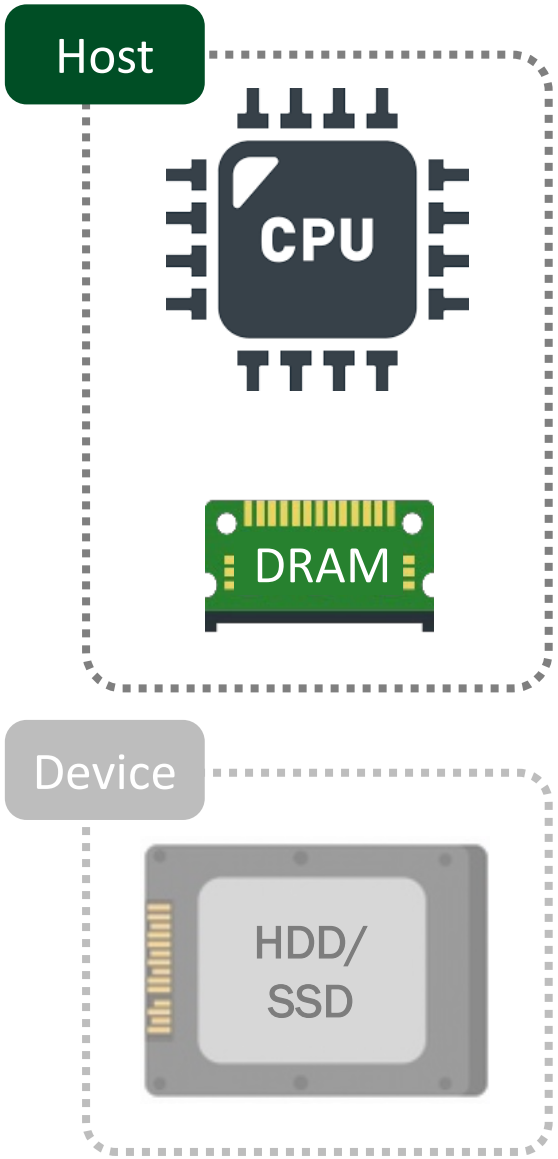
# λ-IO: A Unified IO stack for Computational Storage

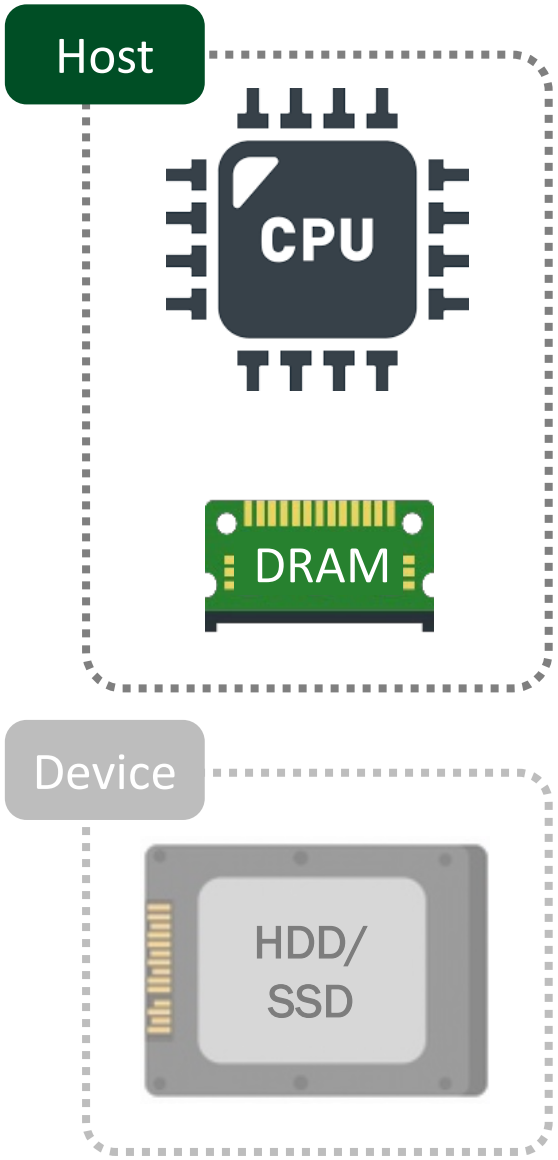
**Zhe Yang**, Youyou Lu, Xiaojian Liao, Youmin Chen,
Junru Li, Siyu He, Jiwu Shu



# Computational Storage



# Computational Storage




**Data-intensive applications**

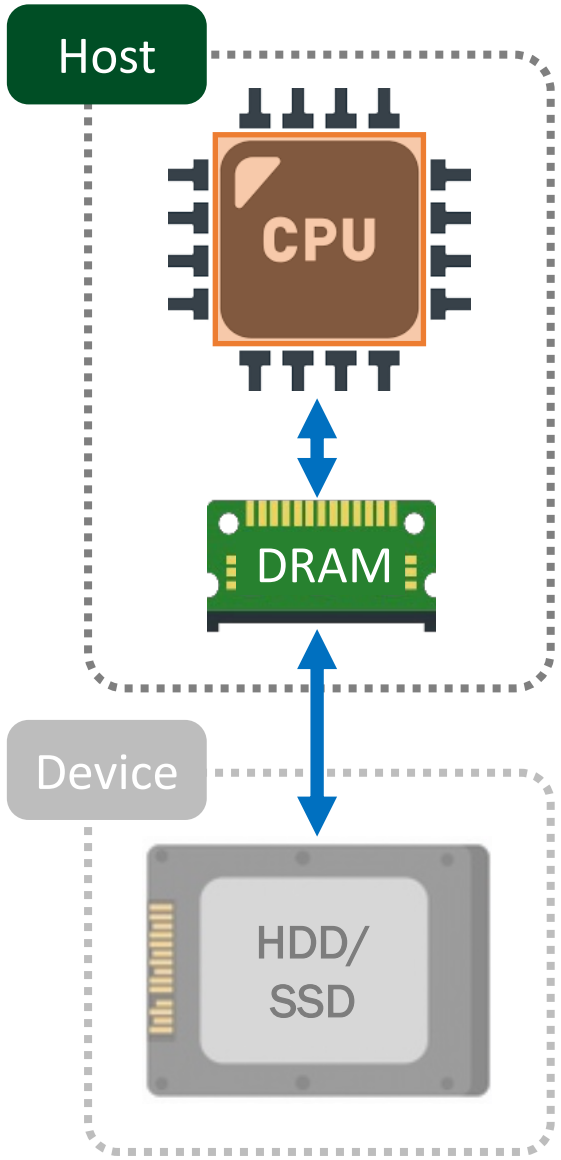






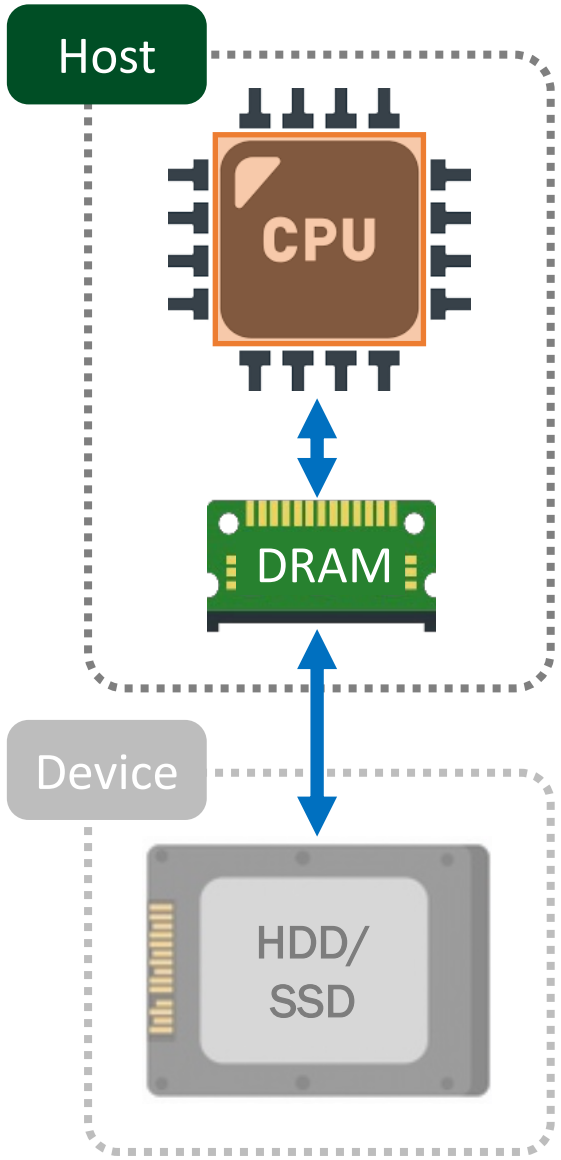


# Computational Storage




**Data-intensive applications**

# Computational Storage




**Data-intensive applications**



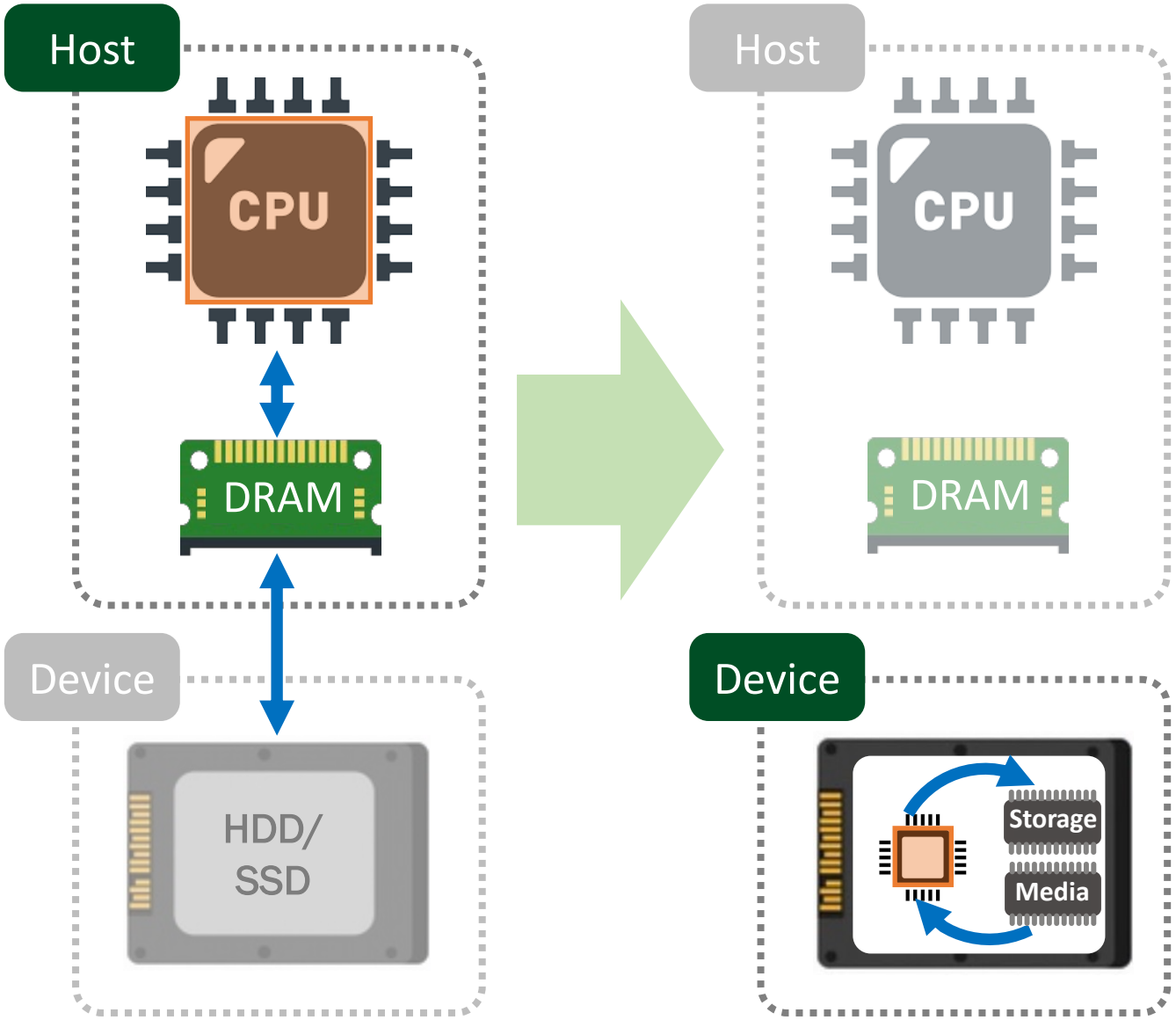




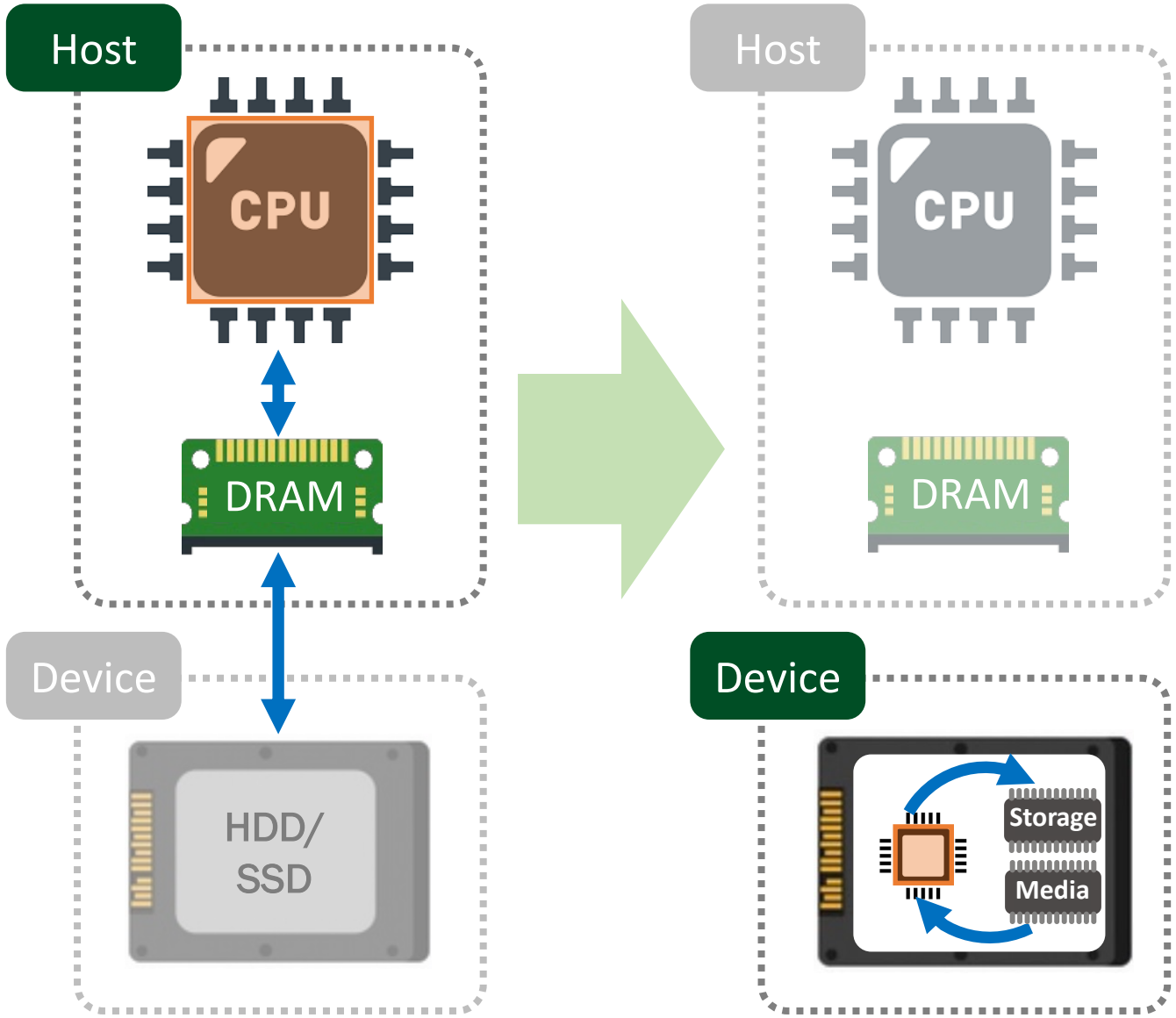


**The host suffers from accessing sizable data on the device.**

# Computational Storage



# Computational Storage




**Encapsulate computation resources to the device**

- **ActiveDisks** [1998]
- **ActiveStorage** [VLDB'98]
- **ActiveFlash** [FAST'13]
- **iSSD** [ICS'13]

# Computational Storage

**YourSQL** [VLDB'16]

**CognitiveSSD** [ATC'19]

**GraFBoost** [ISCA'18]

**Willow** [OSDI'14]

**BlueDBM** [ISCA'15]

**Summarizer** [MICRO'17]

**TimeSSD** [EuroSys'19]

**Biscuit** [ISCA'16]

**MetaIFS** [EuroSys'20]

**CoinPurse** [DAC'20]

**INSIDER** [ATC'19]

**RecSSD** [ASPLOS'21]

**BlockNDP** [Middleware'20]

**RM-SSD** [HPCA'22]

# Computational Storage

**YourSQL** [VLDB'16]

**CognitiveSSD** [ATC'19]

**GraFBoost** [ISCA'18]

**Willow** [OSDI'14]

**BlueDBM** [ISCA'15]

**Summarizer** [MICRO'17]

**TimeSSD** [EuroSys'19]

**Biscuit** [ISCA'16]

**MetalFS** [EuroSys'20]

**CoinPurse** [DAC'20]

**INSIDER** [ATC'19]

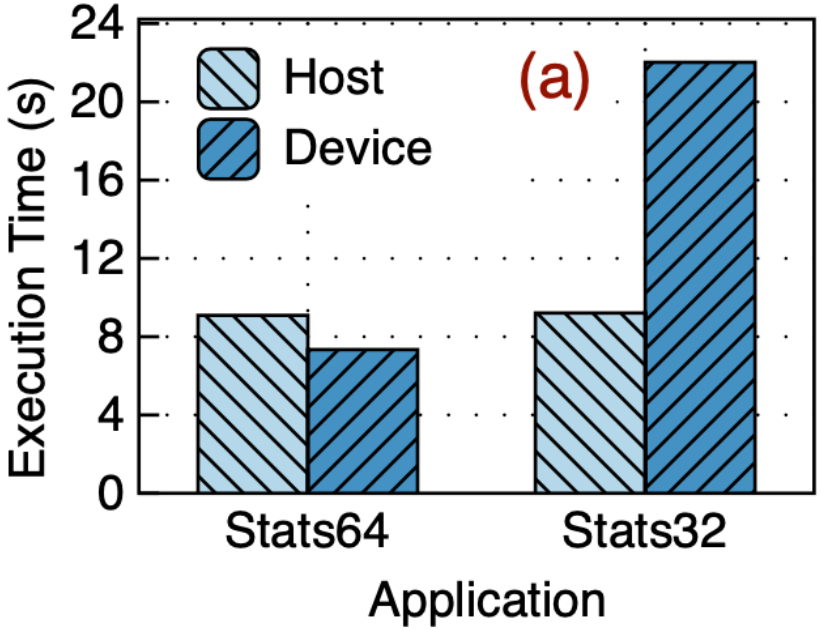
**Re SD** [ASPLOS'21]

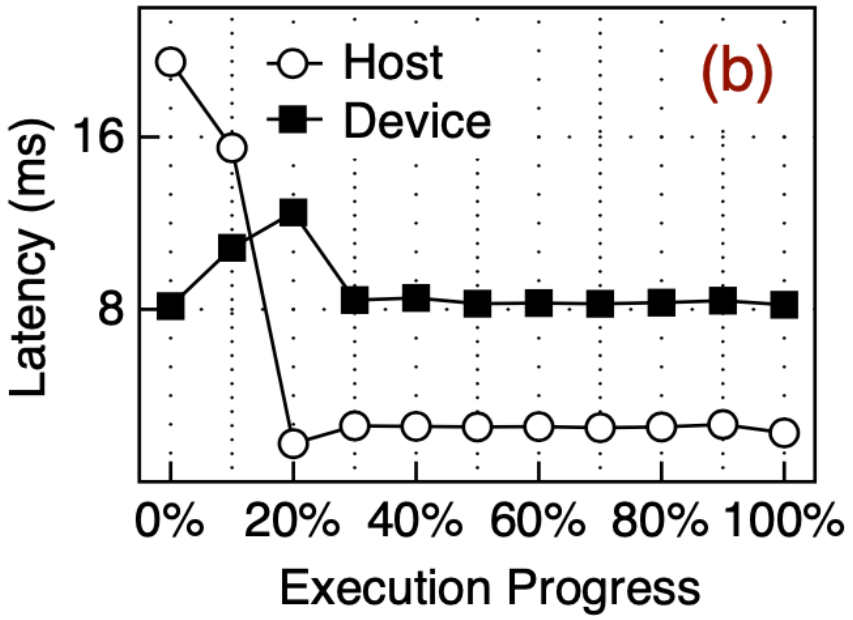
**BlockNDP** [Middleware'20]

! **Existing works focus on accelerating in the device but overlook the host.**

# Host vs. Device

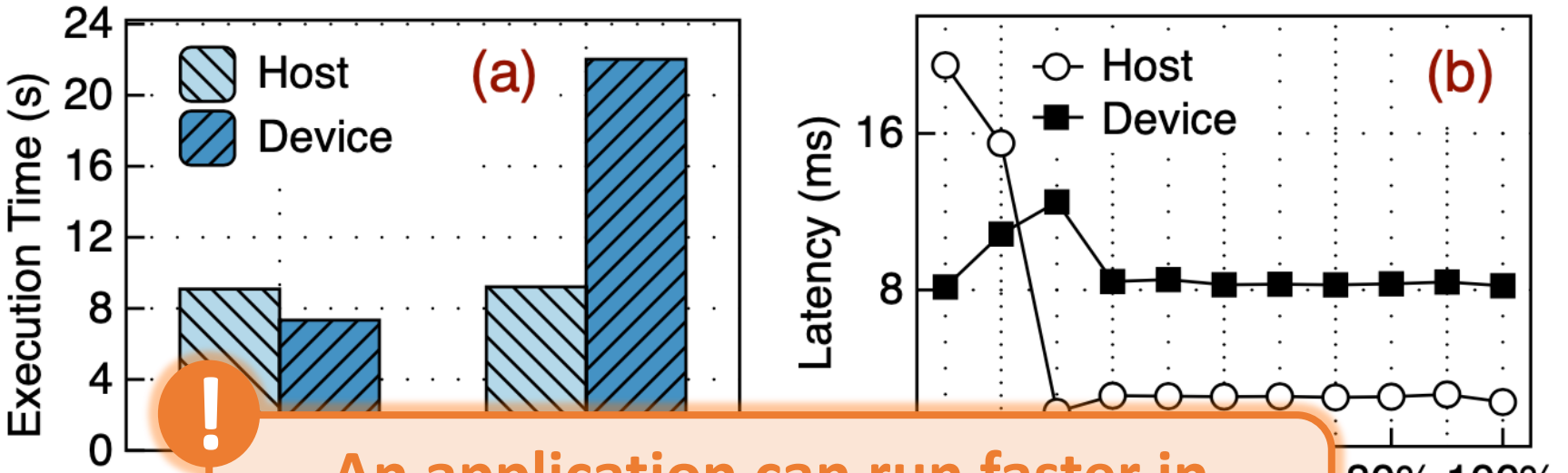
- **Stats64/32**
  - view the file as 64/32-bit integers and calculate statistics
  - (a) Different applications (Stats64 & Stats32)
  - (b) One application (Stats64)



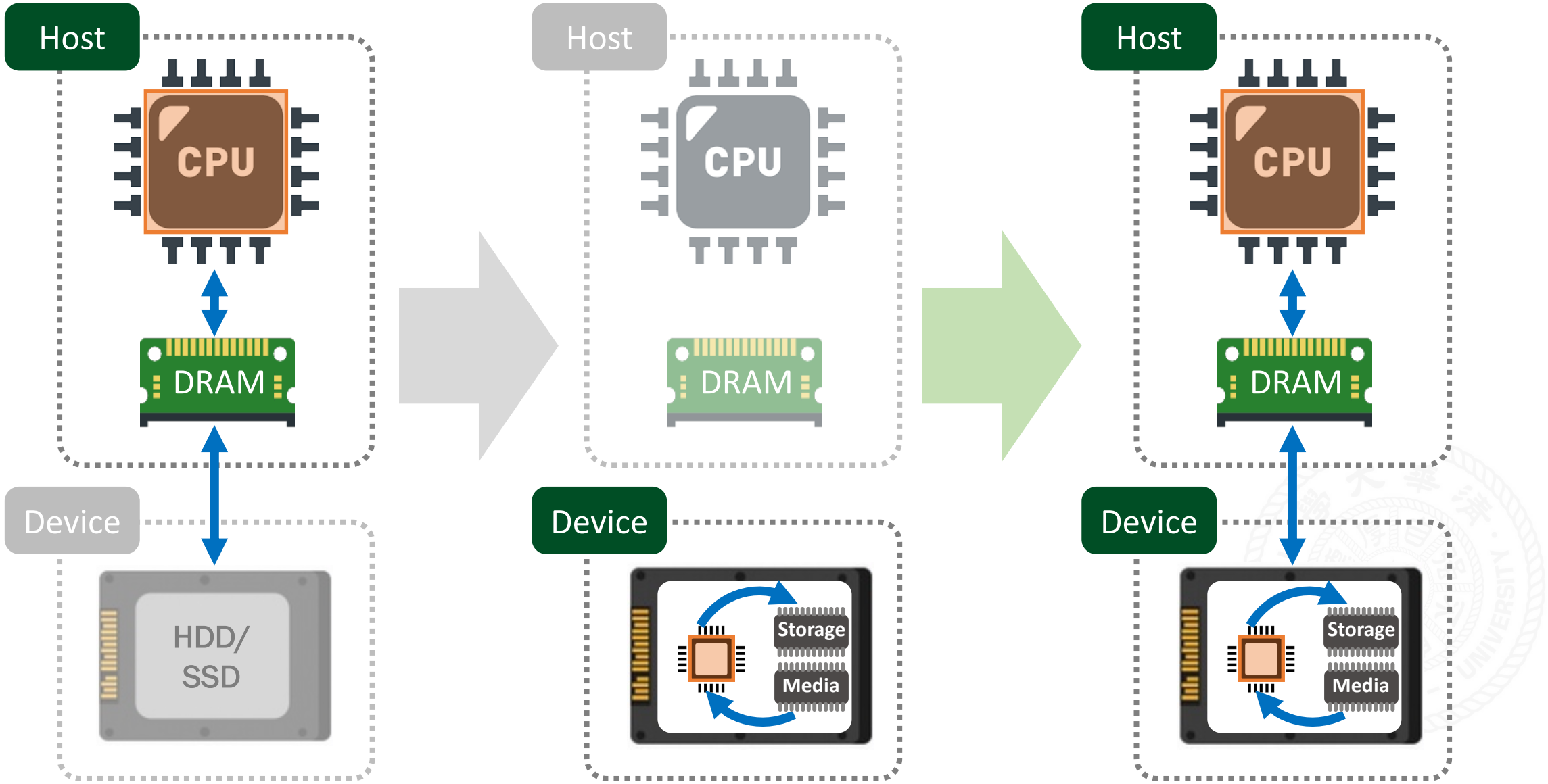



# Host vs. Device

- **Stats64/32**
  - view the file as 64/32-bit integers and calculate statistics
  - (a) Different applications (Stats64 & Stats32)
  - (b) One application (Stats64)



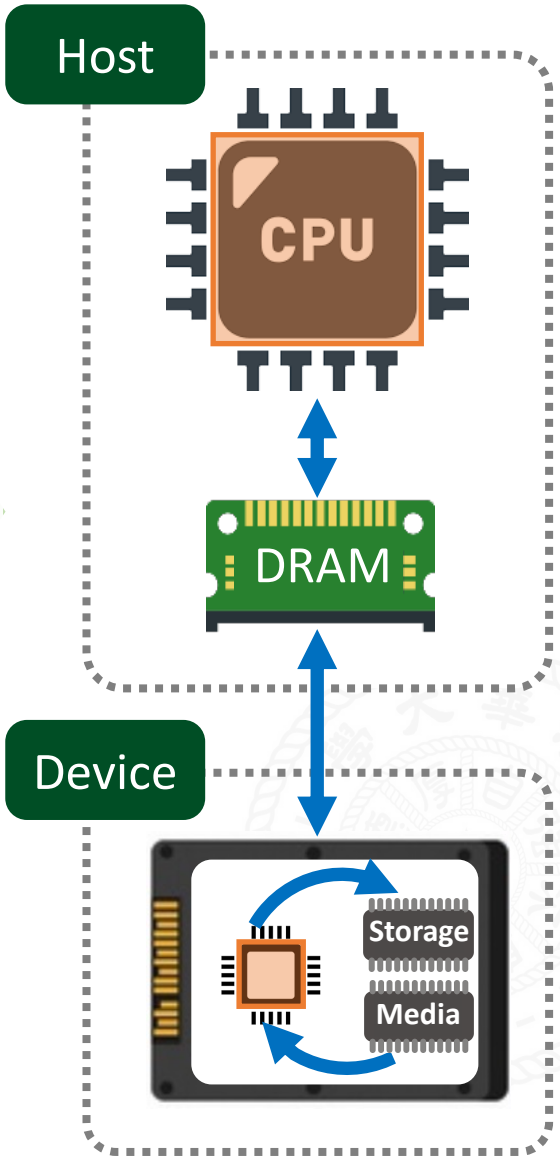
# Exploiting Both Sides



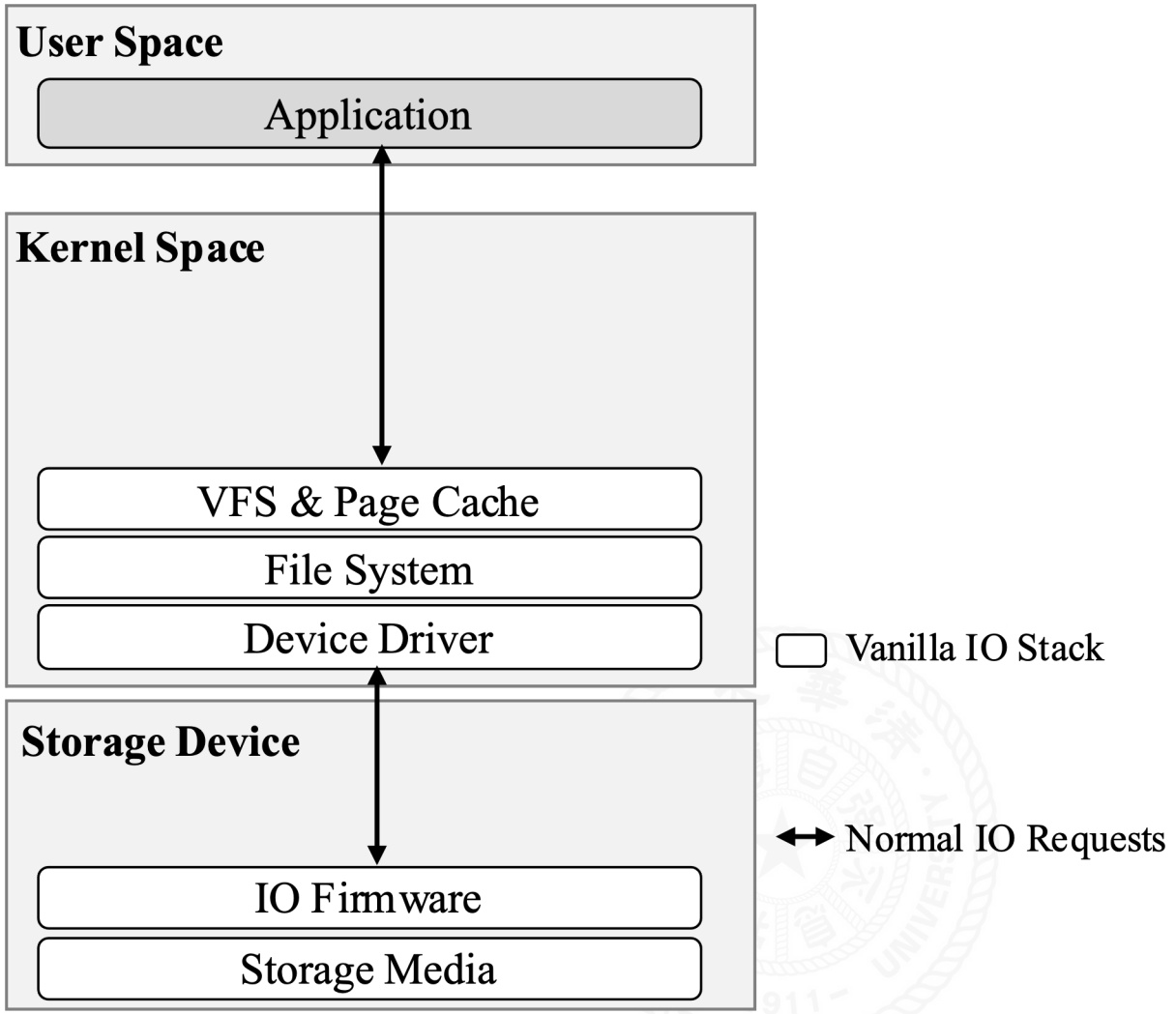
# Exploiting Both Sides

?

**How to build a unified IO stack that exploits both the host and the device?**



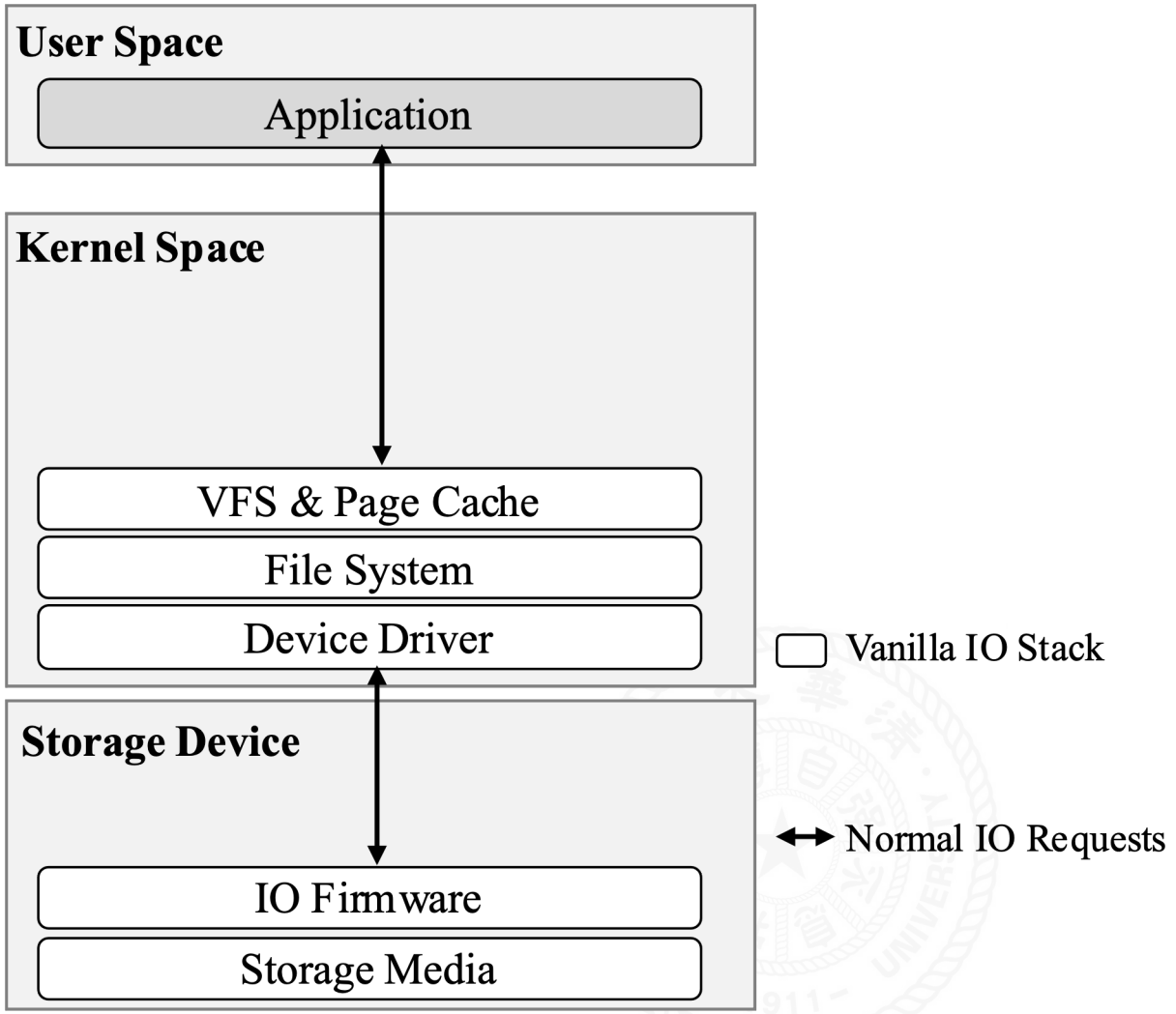
# Challenges


User Space
Application
Kernel Space
VFS & Page Cache
File System
Device Driver
Storage Device
IO Firmware
Storage Media
Vanilla IO Stack
Normal IO Requests

# Challenges

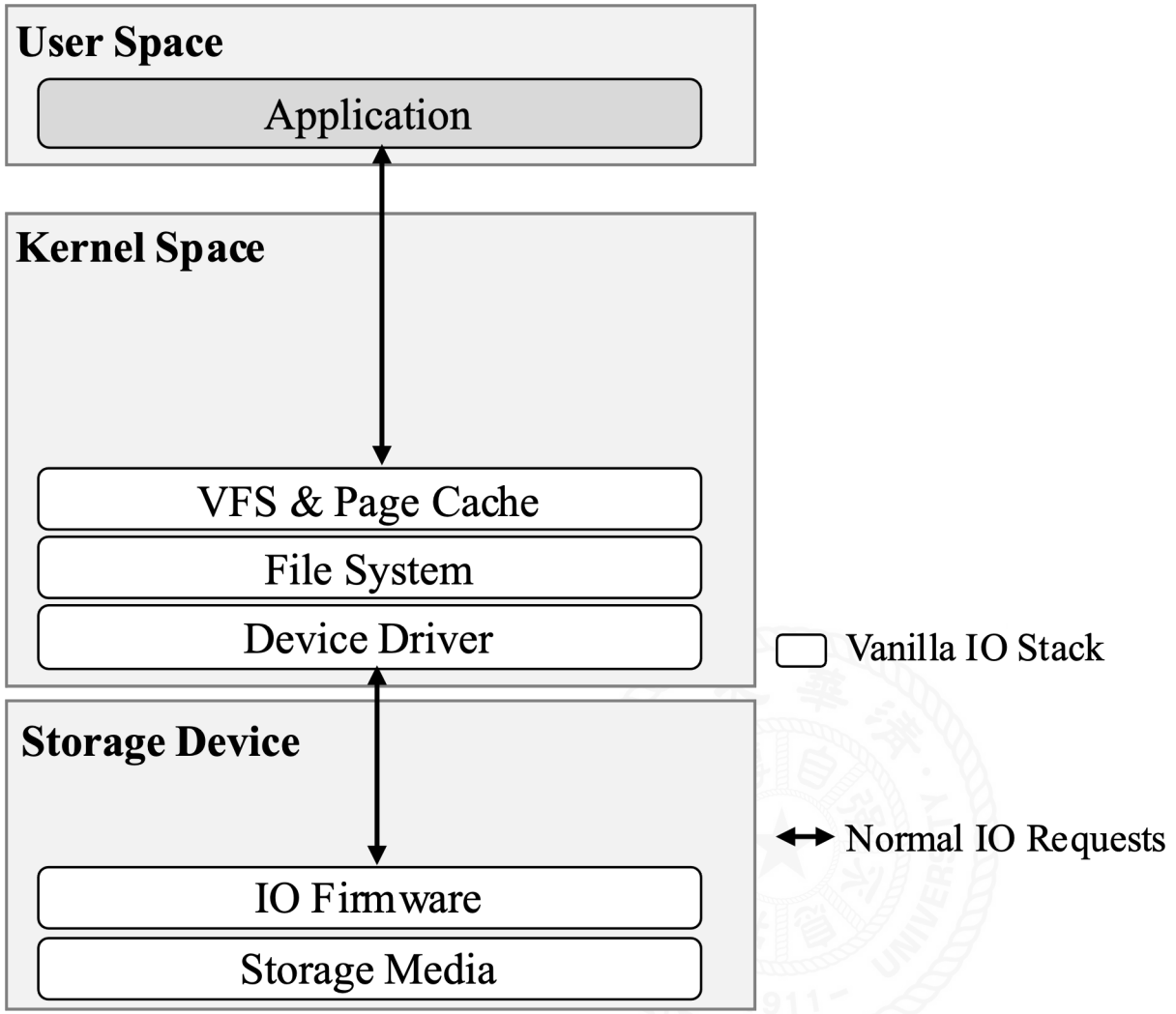
## ❶ How to use (interface)

✗ Easy-to-use APIs



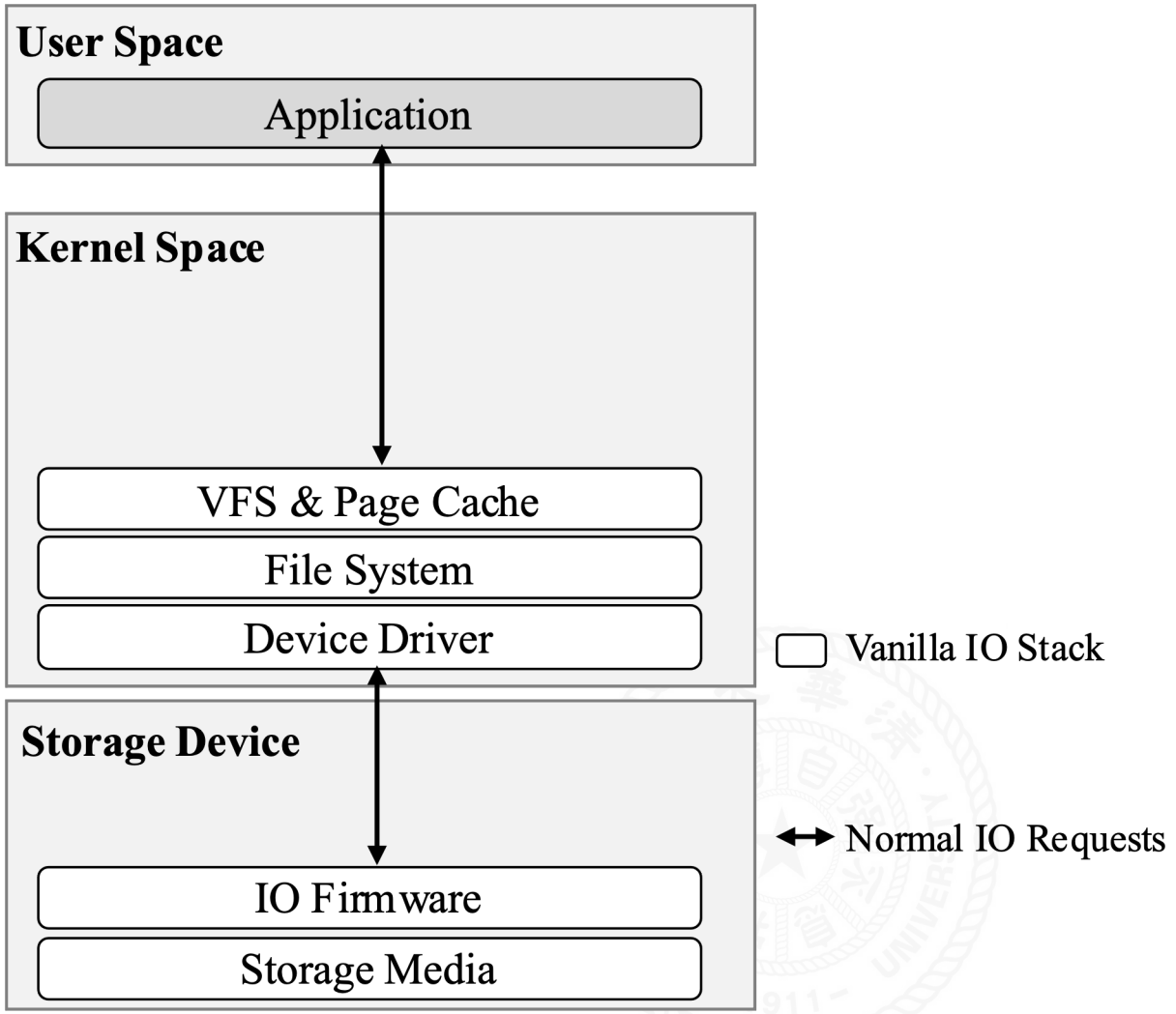
# Challenges

1. **How to use (interface)**
   - ✗ Easy-to-use APIs
2. **How to compute (runtime)**
   - ✗ Compute on both sides



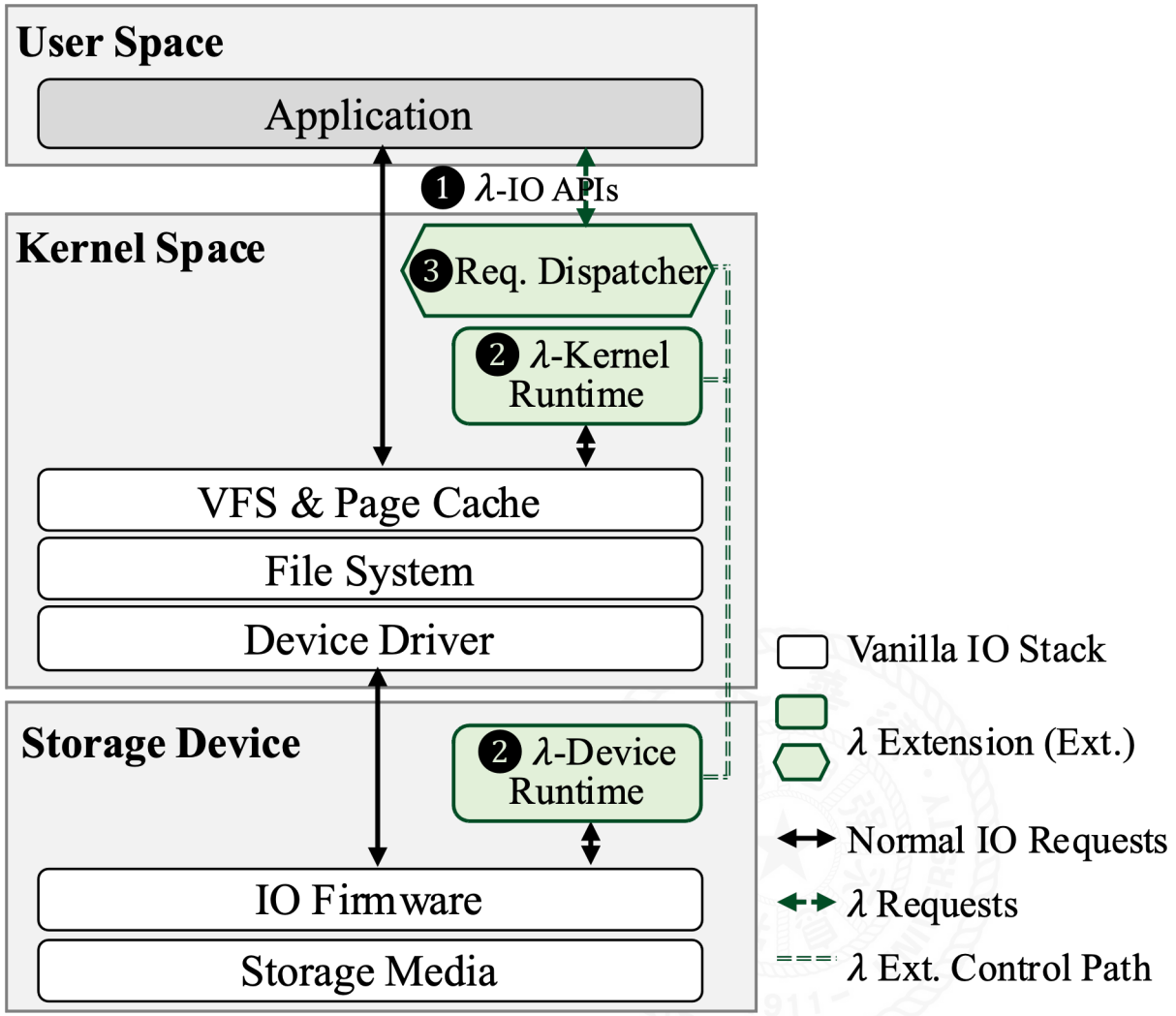
# Challenges

1. **How to use (interface)**
   - ✗ Easy-to-use APIs
2. **How to compute (runtime)**
   - ✗ Compute on both sides
3. **How to dispatch (scheduling)**
   - ✗ Dispatch to both sides



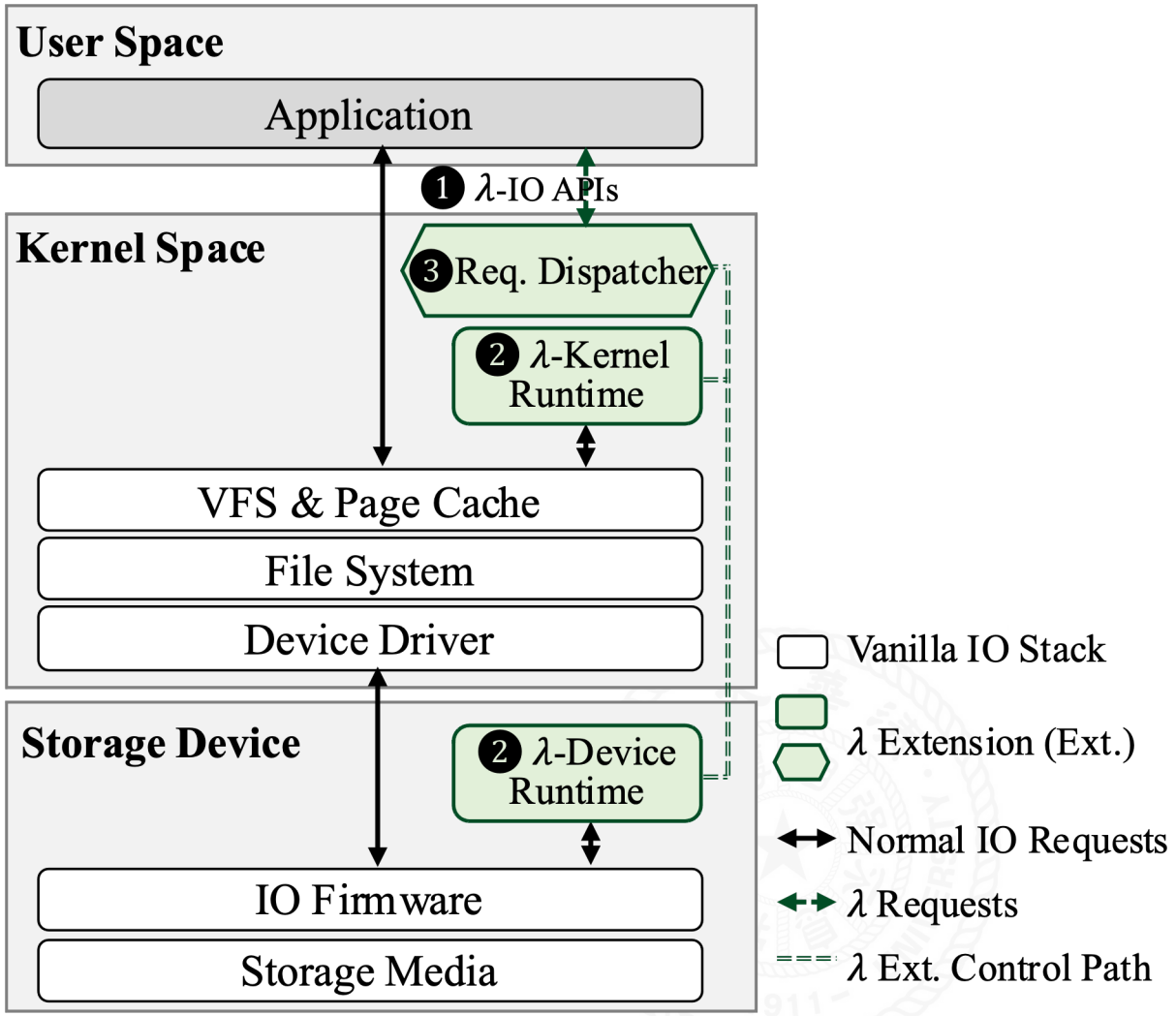
# λ-IO Overall Architecture

1. **How to use (interface)**
   - ✗ Easy-to-use APIs
2. **How to compute (runtime)**
   - ✗ Compute on both sides
3. **How to dispatch (scheduling)**
   - ✗ Dispatch to both sides



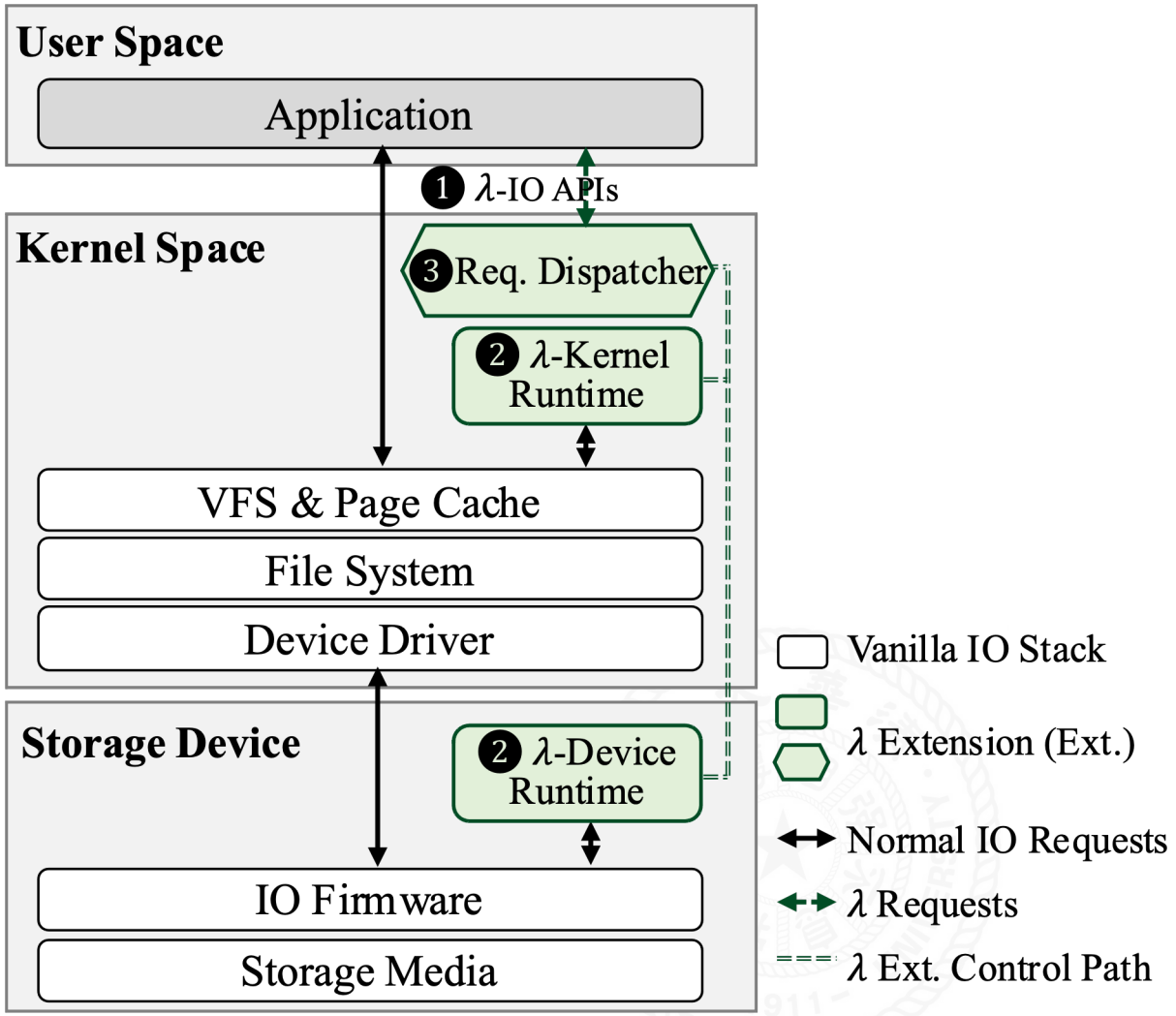
# λ-IO Overall Architecture

1. **How to use (interface)**
   - ✗ Easy-to-use APIs
   - ✓ λ-IO APIs and workflow
2. **How to compute (runtime)**
   - ✗ Compute on both sides
3. **How to dispatch (scheduling)**
   - ✗ Dispatch to both sides



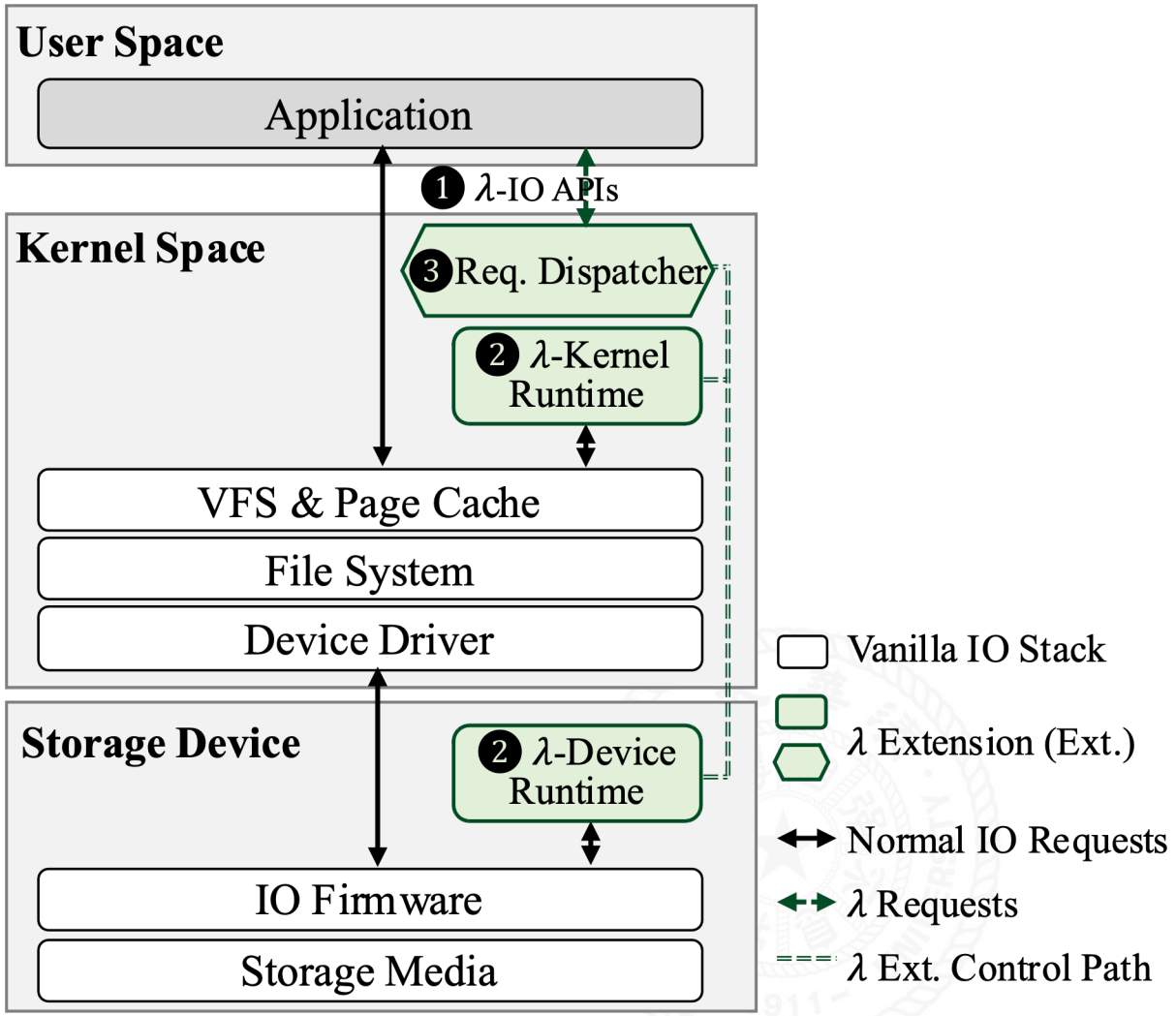
# λ-IO Overall Architecture

1. **How to use (interface)**
   - ✗ Easy-to-use APIs
   - ✓ λ-IO APIs and workflow
2. **How to compute (runtime)**
   - ✗ Compute on both sides
   - ✓ Cross-platform λ runtime
3. **How to dispatch (scheduling)**
   - ✗ Dispatch to both sides



# λ-IO Overall Architecture

1. **How to use (interface)**
   - ✗ Easy-to-use APIs
   - ✓ λ-IO APIs and workflow
2. **How to compute (runtime)**
   - ✗ Compute on both sides
   - ✓ Cross-platform λ runtime
3. **How to dispatch (scheduling)**
   - ✗ Dispatch to both sides
   - ✓ Dynamic request dispatching



# Design 1: λ-IO APIs and Workflow

- **APIs: λ, load, read, write**

| Op | Interface |
|---|---|
| λ | `ssize_t compute(void *output, void *input, size_t length_i)` |
| load | `λ_id = load_λ(λ_path)` |
| open | `fd = open(file_path)` |
| close | `close(fd)` |
| read | `pread(fd, buf, length, offset)`<br>`pread_λ(fd, buf, length, offset, λ_id)` |
| write | `pwrite(fd, buf, length, offset)`<br>`pwrite_λ(fd, buf, length, offset, λ_id)` |

# Design 1: λ-IO APIs and Workflow

- **APIs: λ, load, read, write**

| Op | Interface |
|---|---|
| λ | `ssize_t compute(void *output, void *input, size_t length_i)` |
| load | `λ_id = load_λ(λ_path)` |
| open | `fd = open(file_path)` |
| close | `close(fd)` |
| read | `pread(fd, buf, length, offset)`<br>`pread_λ(fd, buf, length, offset, λ_id)` |
| write | `p`<br>`pwr` |

Go on with an example to sum integers in a file

# Design 1: λ-IO APIs and Workflow

- **λ**
  - user-defined computational logic
  - `ssize_t compute(void *output, void *input, size_t length_i)`
  - Consume data in `input`
  - Produce data in `output`

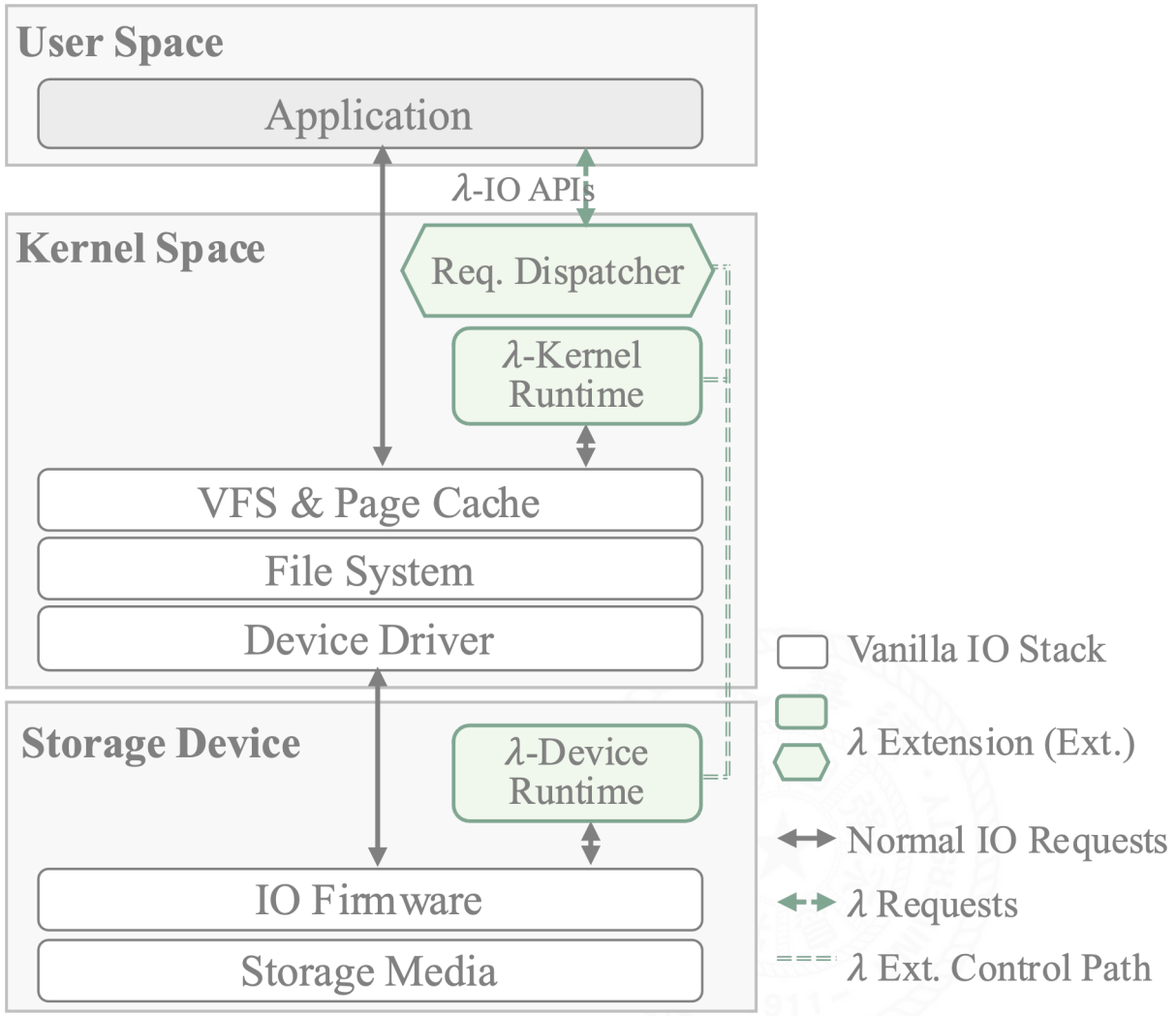
# Design 1: λ-IO APIs and Workflow

- **λ**
  - user-defined computational logic
  - `ssize_t compute(void *output, void *input, size_t length_i)`
  - Consume data in `input`
  - Produce data in `output`

```
1  // sum.c
2  ssize_t compute(void *output, void *input, size_t
       length_i) {
3       int sum = 0;
4       for (int i = 0; i < length_i / sizeof(int); i++)
       {
5           sum += ((int *)input)[i];
6       }
7       *output = sum;
8       return sizeof(int); // return the output
       size
9  }
```

# Design 1: λ-IO APIs and Workflow

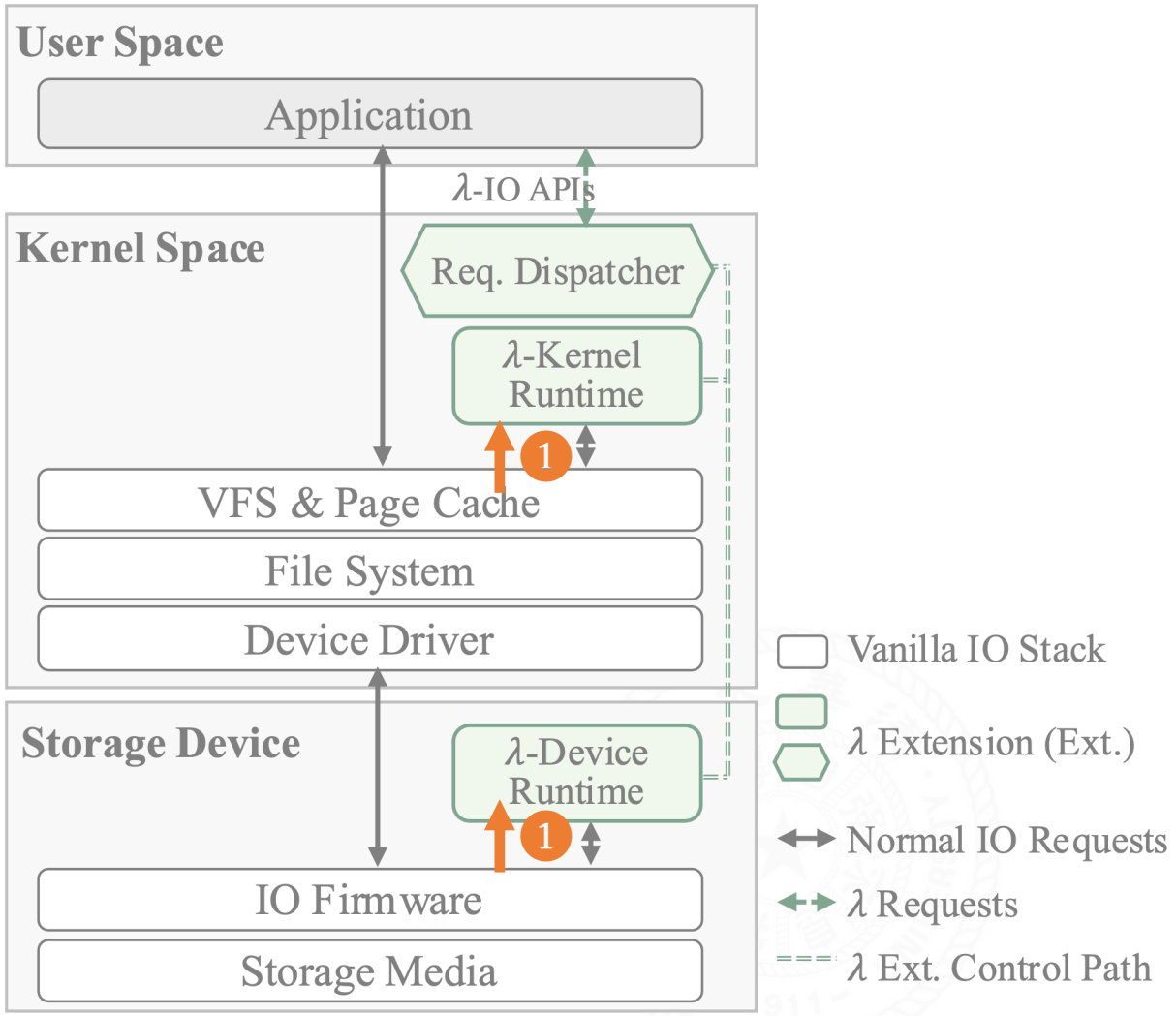
- **Read**
  - Execute during read
  - `pread_λ(fd, buf, length, offset, λ_id)`



# Design 1: λ-IO APIs and Workflow

- **Read**
  - Execute during read
  - `pread_λ(fd, buf, length, offset, λ_id)`

❶ read file to `input`

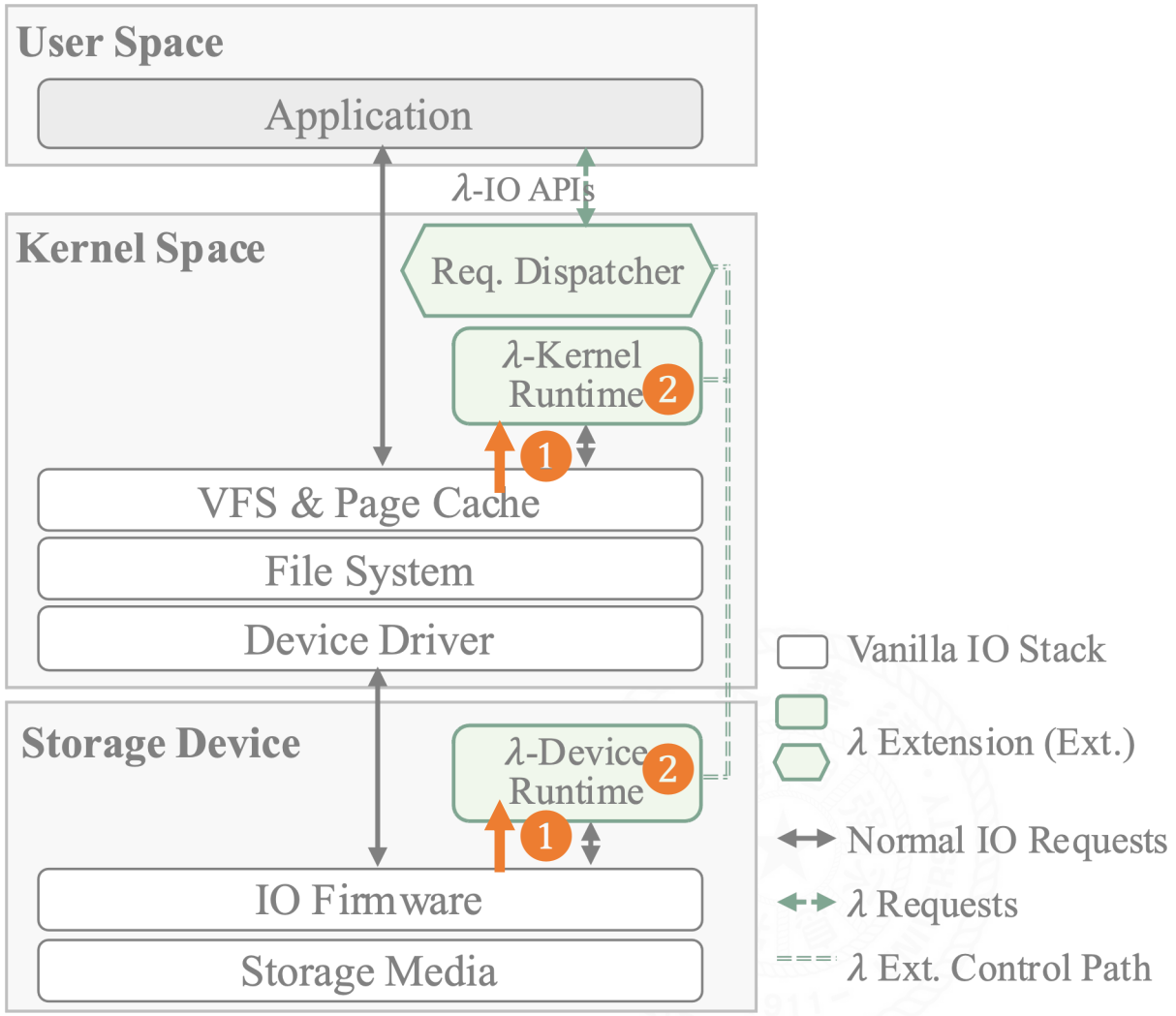


# Design 1: λ-IO APIs and Workflow

- **Read**
  - Execute during read
  - `pread_λ(fd, buf, length, offset, λ_id)`

❶ read file to `input`

❷ execute λ



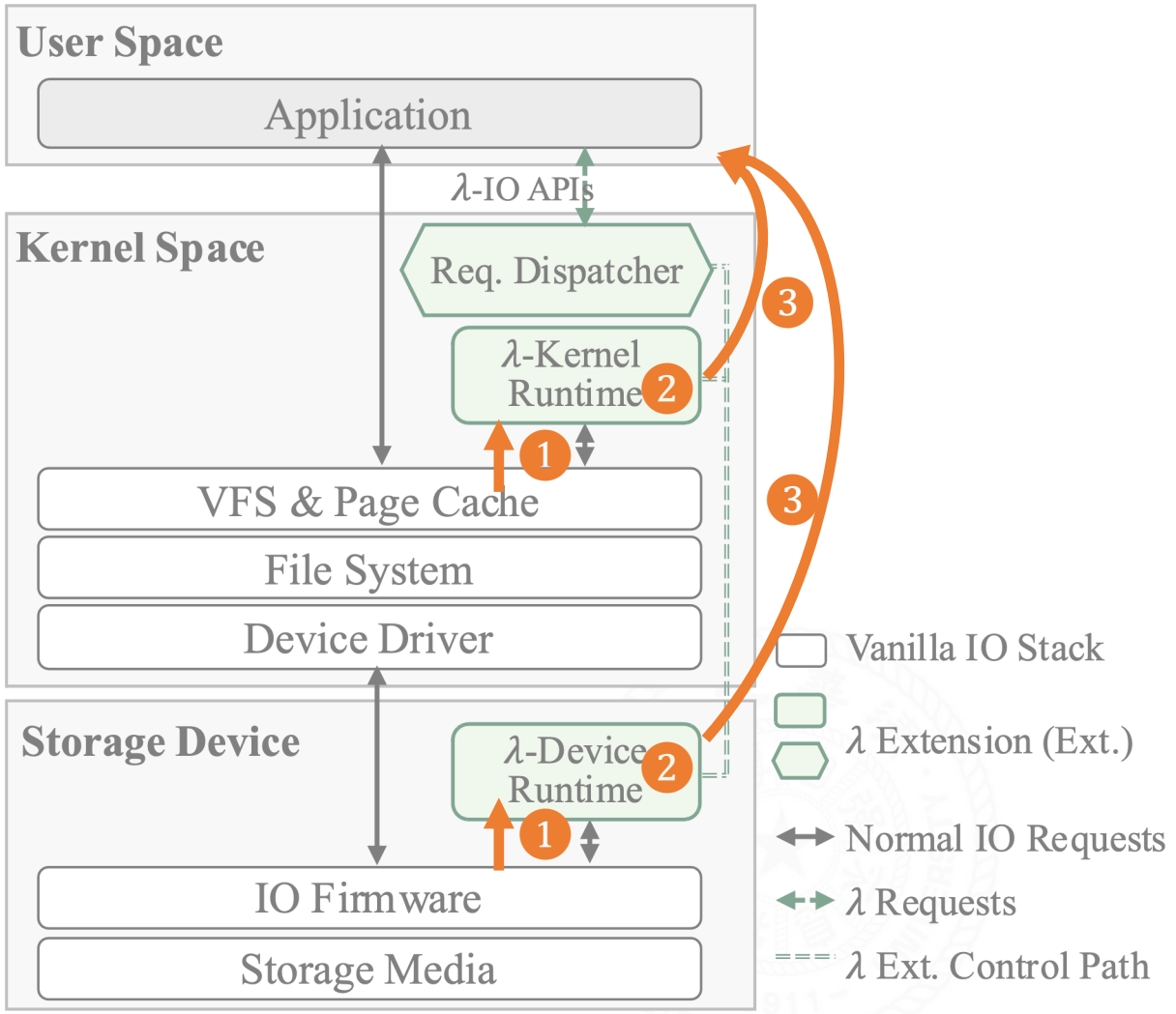
# Design 1: λ-IO APIs and Workflow

- **Read**
  - Execute during read
  - `pread_λ(fd, buf, length, offset, λ_id)`

❶ read file to `input`

❷ execute λ

❸ return `output`

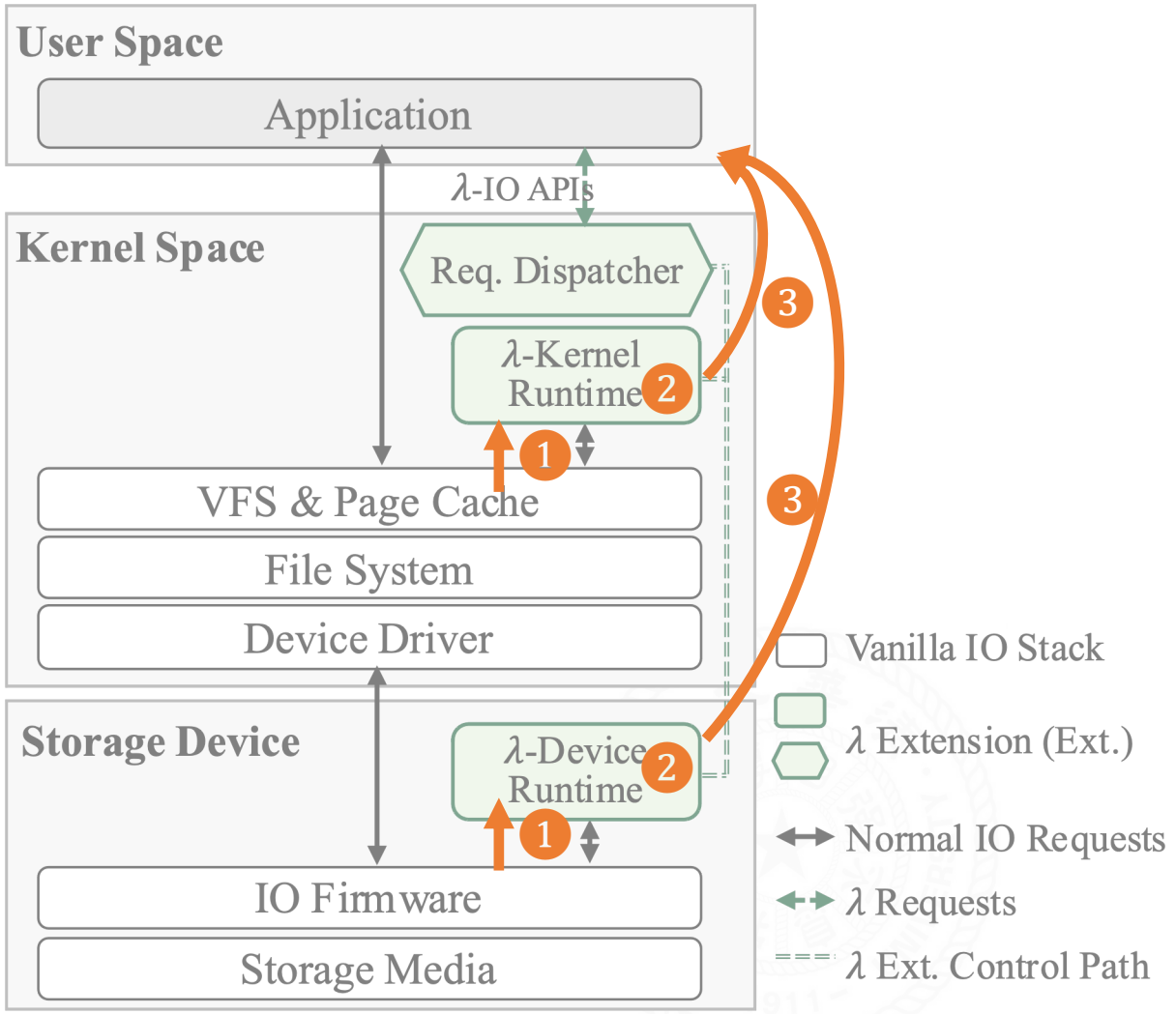


# Design 1: λ-IO APIs and Workflow

- **Read**
  - Execute during read
  - `pread_λ(fd, buf, length, offset, λ_id)`
  1. read file to `input`
  2. execute λ
  3. return `output`

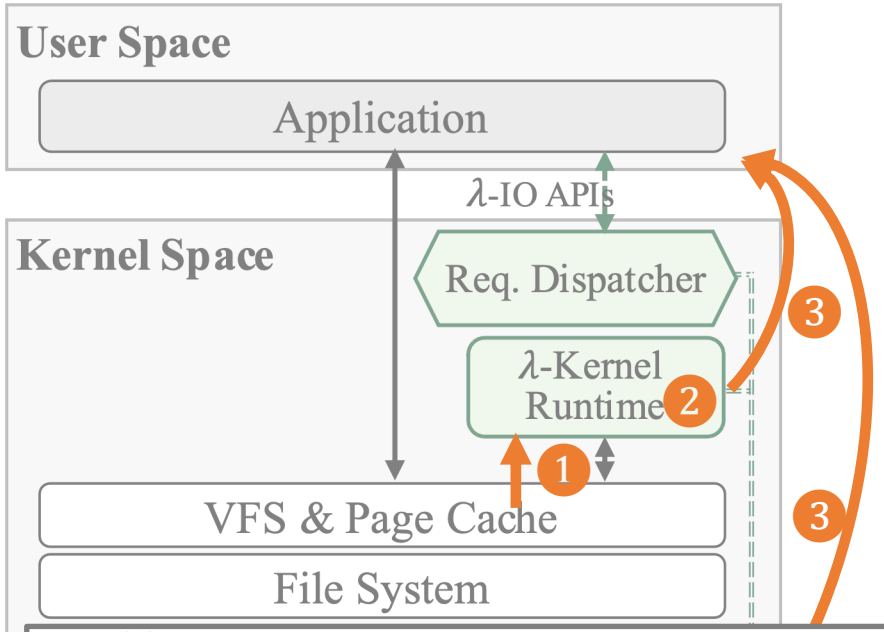
```
23  int lambda_io_sum(int fd, int file_size) {
24      int λ_id = load_λ("sum.sbpf");
25      char buf[sizeof(int)];
26      int sum = 0;
27      for (int i = 0; i < file_size; i += BUF_SIZE) {
28          pread_λ(fd, buf,  BUF_SIZE, i, λ_id);
29          sum += *(int *)buf;
30      }
31      return sum;
32  }
```

λ-IO



# Design 1: λ-IO APIs and Workflow

- **Read**
  - Execute during read
  - `pread_λ(fd, buf, length, offset, λ_id)`
  1. read file to `input`
  2. execute λ
  3. return `output`




```
23  int lambda_io_sum(int fd, int file_size) {
24      int λ_id = load_λ("sum.sbpf");
25      char buf[sizeof(int)];
26      int sum = 0;
27      for (int i = 0; i < file_size; i += BUF_SIZE) {
28          pread_λ(fd, buf,  BUF_SIZE, i, λ_id);
29          sum += *(int *)buf;
30      }
31      return sum;
32  }
```

λ-IO

```
11  // main.c
12  int vanilla_io_sum(int fd, int file_size) {
13      char buf[BUF_SIZE];
14      int sum = 0, sum_s;
15      for (int i = 0; i < file_size; i += BUF_SIZE) {
16          pread(fd, buf, BUF_SIZE, i);
17          compute(&sum_s, buf, BUF_SIZE);
18          sum += sum_s;
19      }
20      return sum;
21  }
```

Vanilla IO

# Design 2: Cross-Platform λ Runtime

- **eBPF is inapplicable to computational storage**
  - Pointer access to memory buffers
  - Dynamic-length loop bounded by parameters

```
1  // sum.c
2  ssize_t compute(void *output, void *input, size_t
       length_i) {
3       int sum = 0;
4       for (int i = 0; i < length_i / sizeof(int); i++)
       {
5           sum += ((int *)input)[i];
6       }
7       *output = sum;
8       return sizeof(int); // return the output
       size
9  }
```

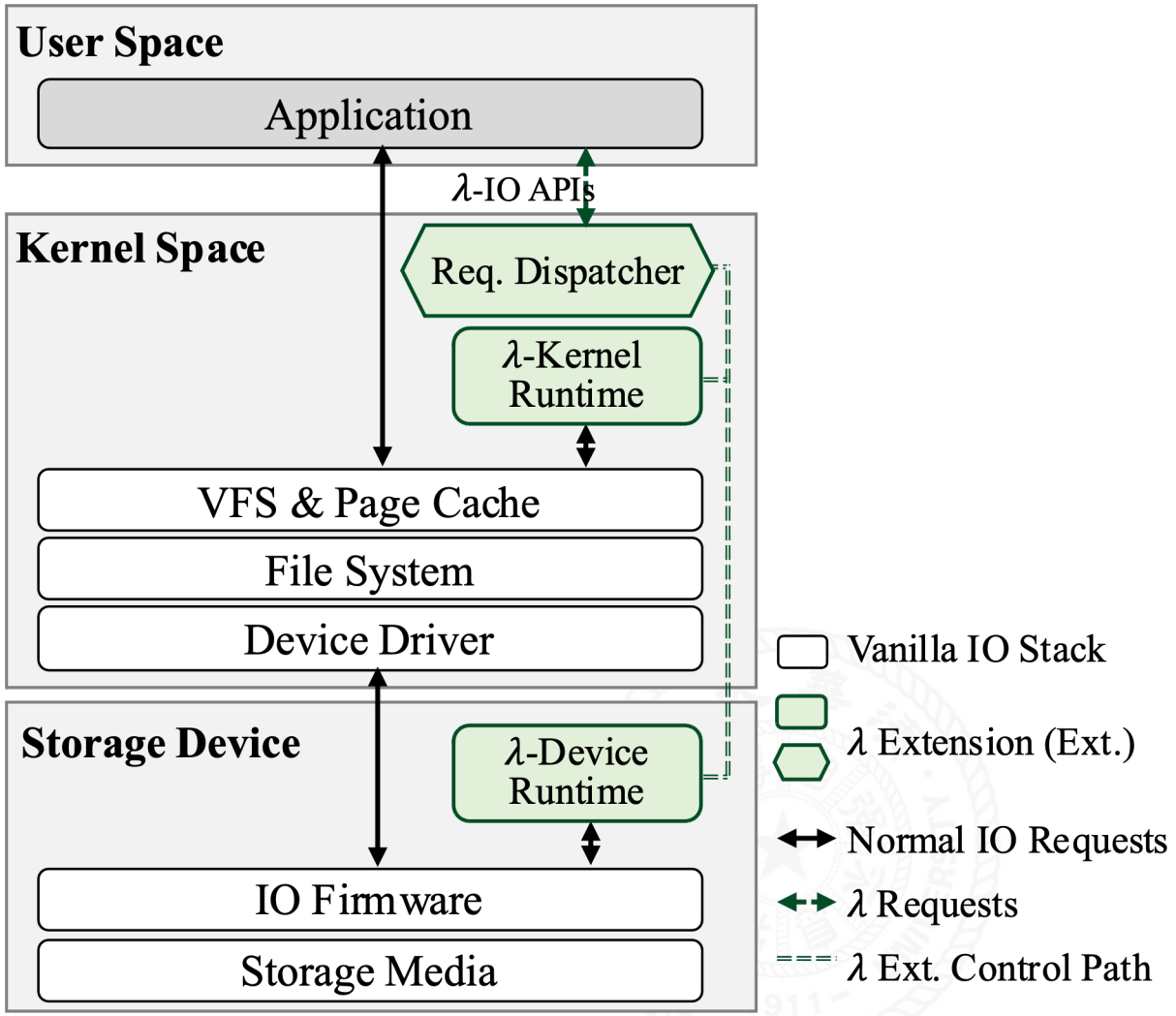
# Design 2: Cross-Platform λ Runtime

- **Extending eBPF to sBPF**
  - Loose the static verification
  - Introduce dynamic checking
- **Pointer access**
  - check during running
- **Dynamic-length loop**
  - count and limit loop number during running

```
1  // sum.c
2  ssize_t compute(void *output, void *input, size_t
     length_i) {
3      int sum = 0;
4      for (int i = 0; i < length_i / sizeof(int); i++)
     {
5          sum += ((int *)input)[i];
6      }
7      *output = sum;
8      return sizeof(int); // return the output
     size
9  }
```

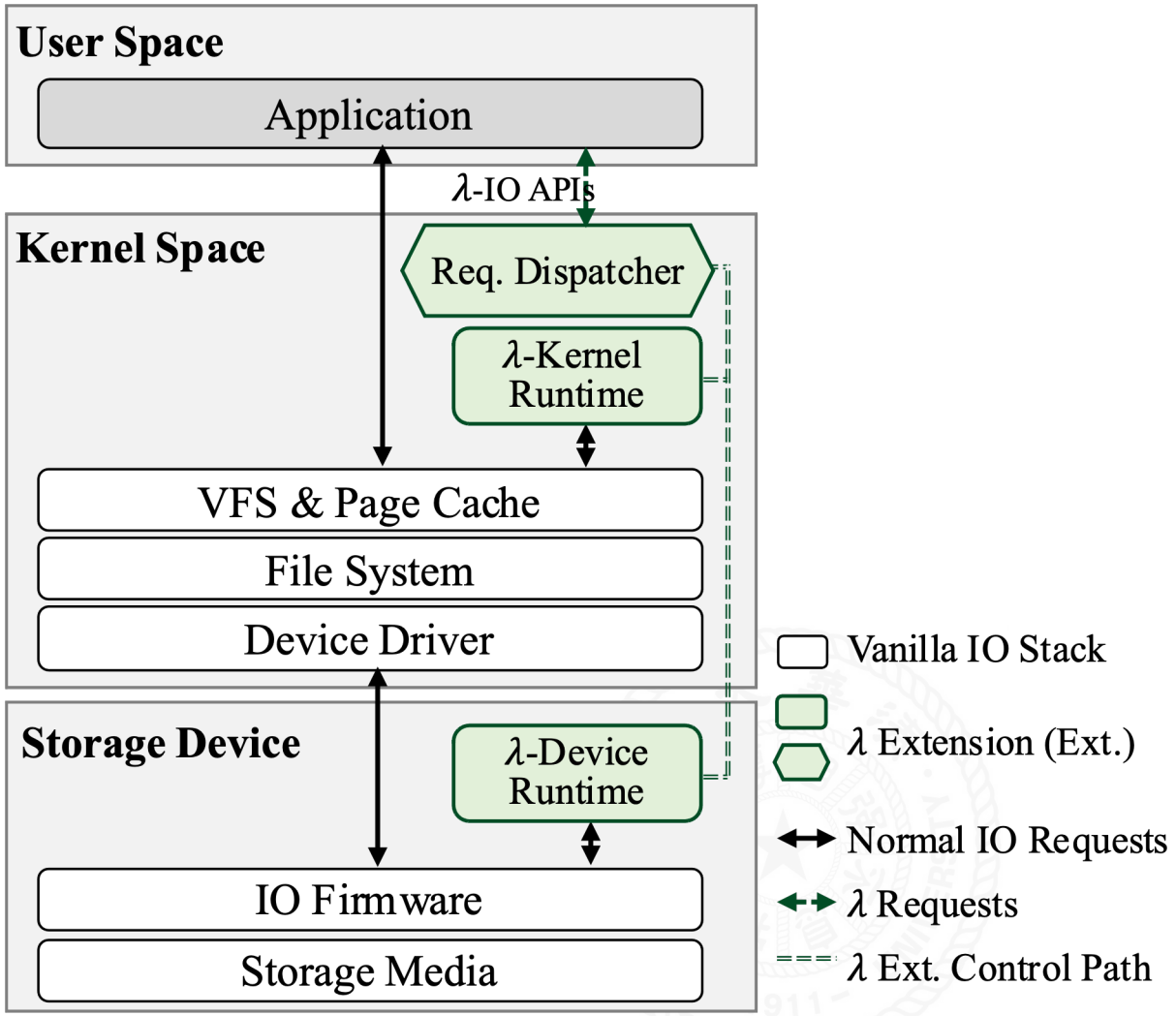
# Design 3: Dynamic Request Dispatching

- **Effectiveness**
  - Modeling execution time
  - Predicting and finding the faster side



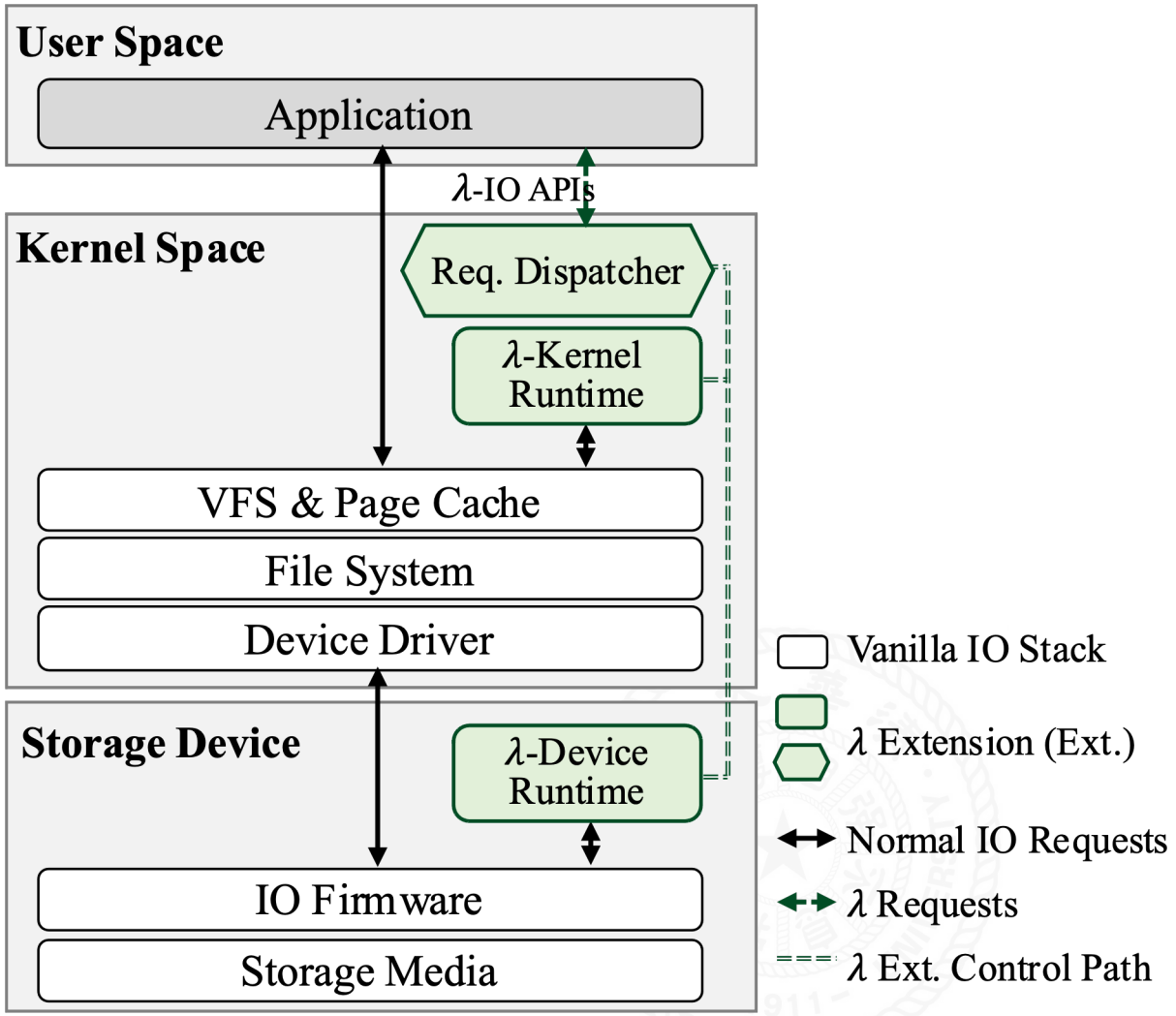
# Design 3: Dynamic Request Dispatching

- **Effectiveness**
  - Modeling execution time
  - Predicting and finding the faster side
- **Efficiency**
  - Periodical profiling and dispatching
  - Introduce low scheduling overhead



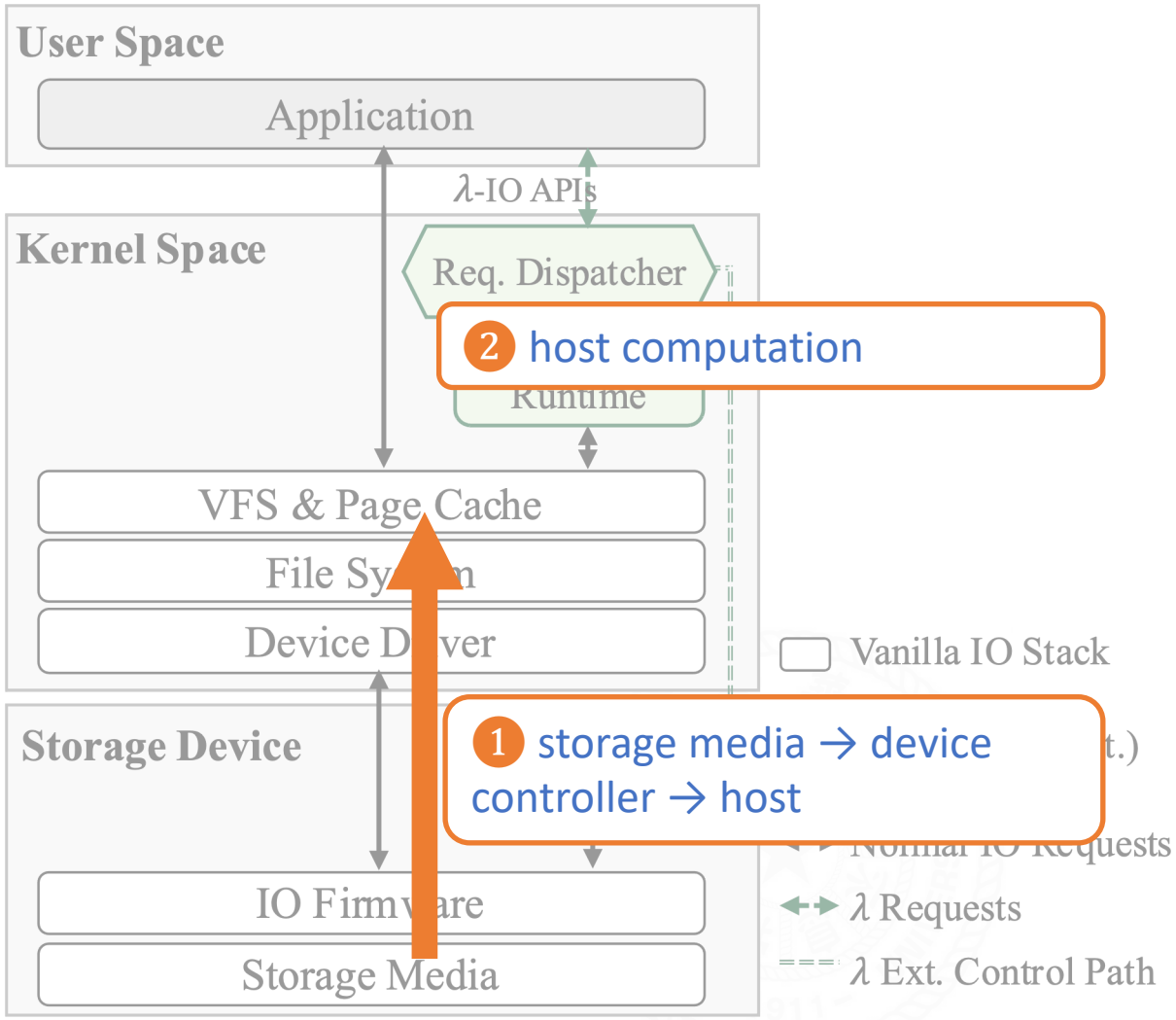
# Design 3: Dynamic Request Dispatching

- **Modeling execution time**
  - Transfer + compute
  - Both sides: the host and the device
  - Factors: computation complexity, data size, cache ratio, real-time status…



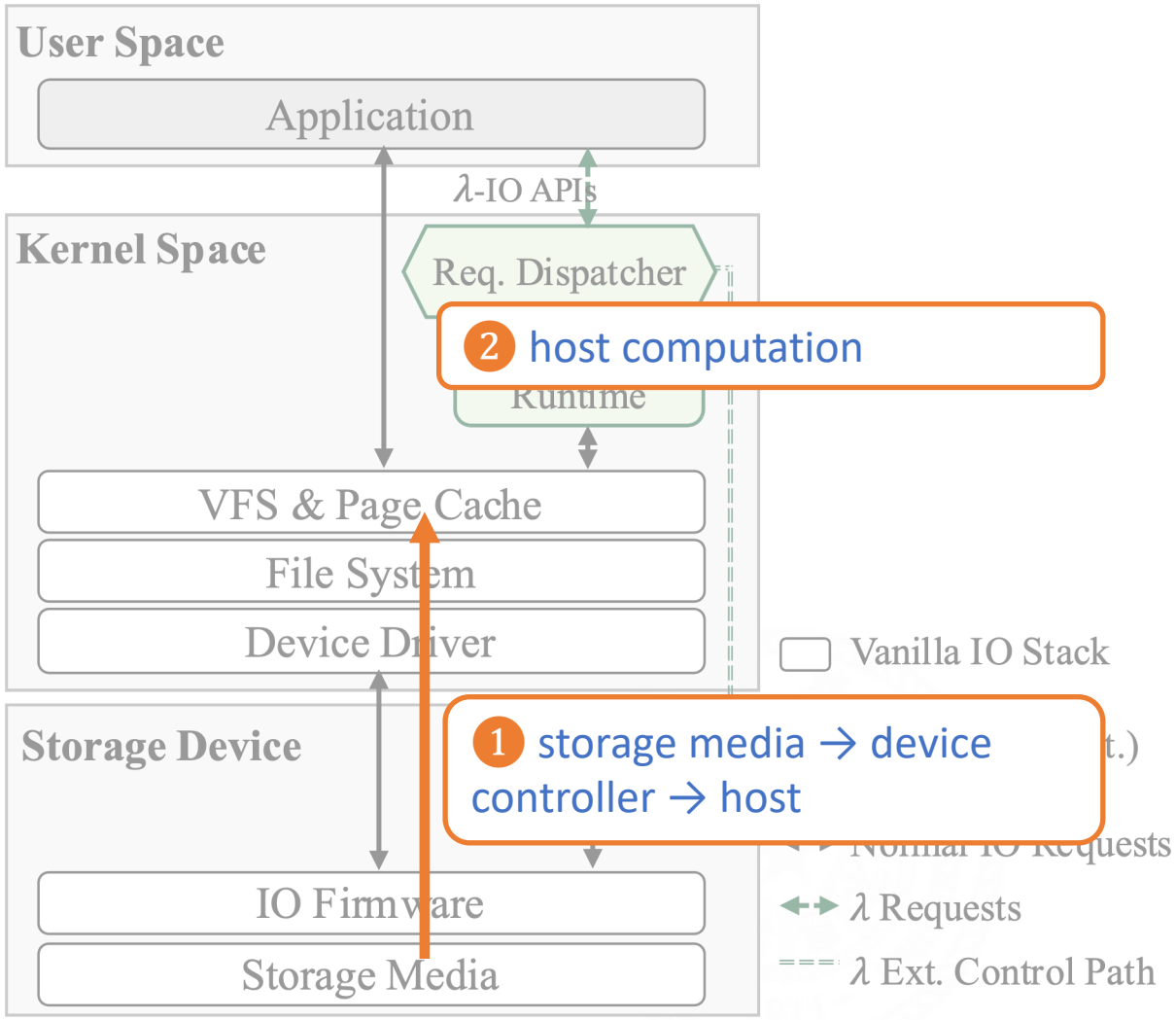
# Design 3: Dynamic Request Dispatching

- **Modeling execution time**
  - Transfer + compute
  - Both sides: the host and the device
  - Factors: computation complexity, data size, cache ratio, real-time status…
- **Cache ratio**
  - Read in the host without cache



# Design 3: Dynamic Request Dispatching

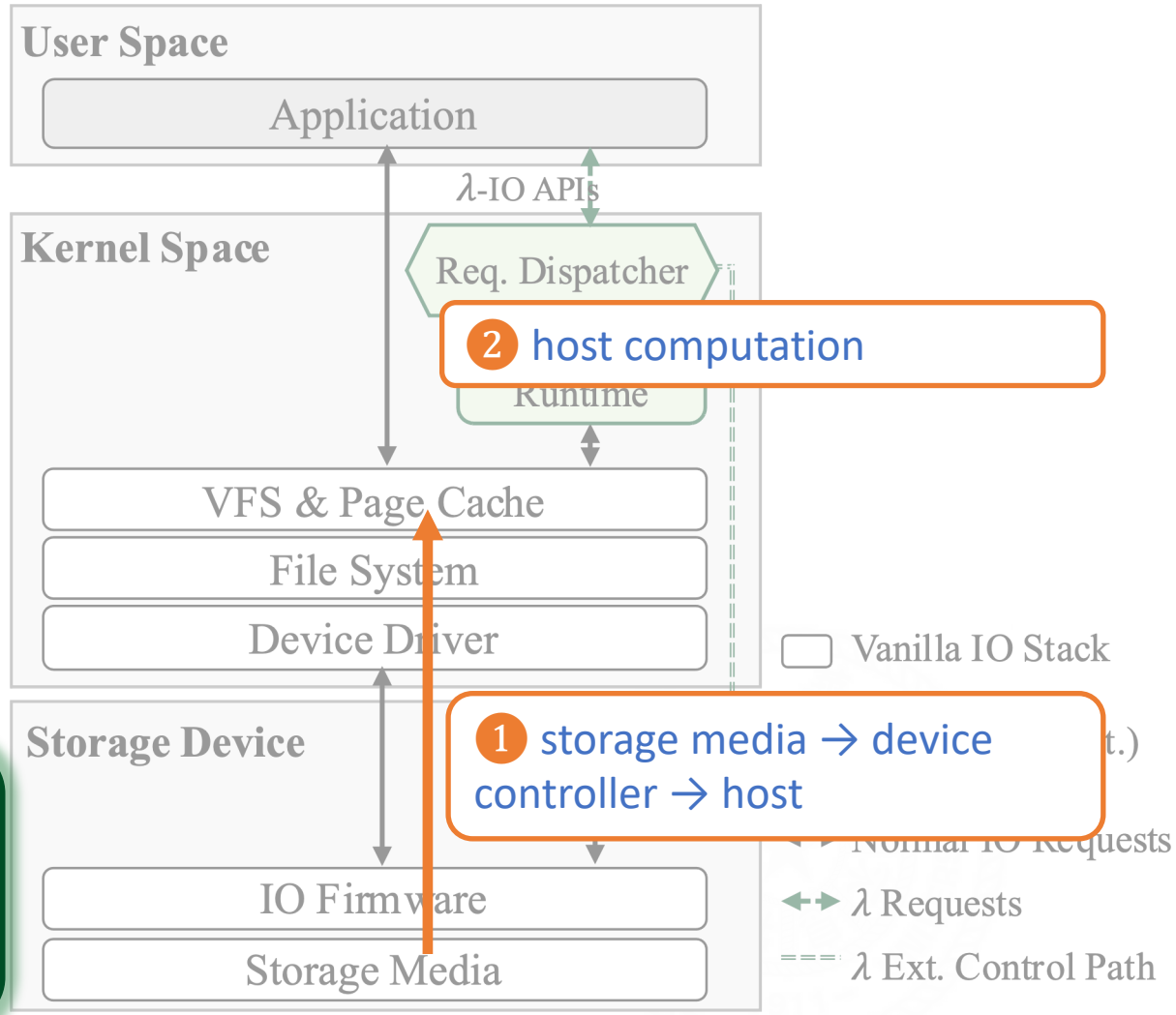
- **Modeling execution time**
  - Transfer + compute
  - Both sides: the host and the device
  - Factors: computation complexity, data size, cache ratio, real-time status…
- **Cache ratio**
  - Read in the host without cache
  - Read in the host with cache



# Design 3: Dynamic Request Dispatching

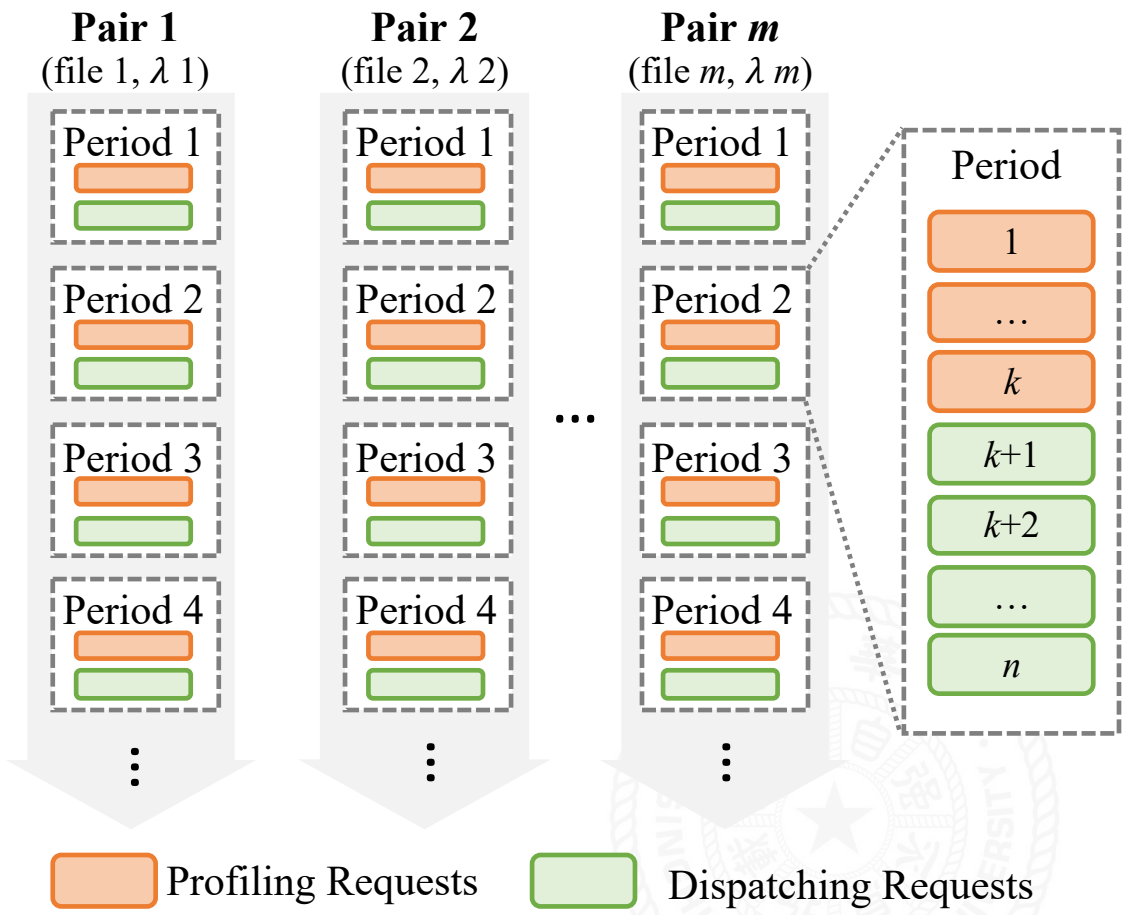
- **Modeling execution time**
  - Transfer + compute
  - Both sides: the host and the device
  - Factors: computation complexity, data size, cache ratio, real-time status…
- **Cache ratio**
  - Read in the host without cache
  - Read in the host with cache

**The higher the cache ratio is,**
**the less data is transferred,**
**the more we tend to execute in the host.**



# Design 3: Dynamic Request Dispatching

- **Periodical profiling and dispatching**
  - For each (file, λ) pair separately
  - Profiling period: n requests
  - Profiling length: the first k requests
- **Profiling (k)**
  1. submit to both sides
  2. profiles real-time status
- **Dispatching (n - k)**
  1. compute execution time
  2. dispatch to the faster side



# Evaluation

- **Testbed**
  - Full-stack software & hardware
  - Device prototype: Daisy OpenSSD (NVMe device)

| | | |
|---|---|---|
| Host | CPU | Intel Core i7-7700 @3.6GHz (4C8T) |
| | Memory | 16GB |
| | OS | Ubuntu 20.04.2 LTS |
| | Kernel | Linux 5.10.21 |
| Device | SoC | Xilinx Zynq Ultrascale+ ZU17EG |
| | Memory | 2GB |
| | Storage | 64GB |



# Evaluation

- **Synthetic applications**

| App. | Description | ISC LoC | |
|---|---|---|---|
| | | λ-IO | INSIDER |
| Stats64 | Read the file data as 64-bit integers and calculate the sum, maximum, and minimum. | 30 | 240 |
| Stats32 | Read the file data as 32-bit integers and calculate the sum, maximum, and minimum. | 30 | 240 |
| KNN | Read vectors in the file and calculate distances between a target vector. | 32 | 99 |
| Grep | Read rows in the file and match a target string. | 35 | 254 |
| Bitmap | Decompress a bitmap and write to the file. | 20 | 188 |

read

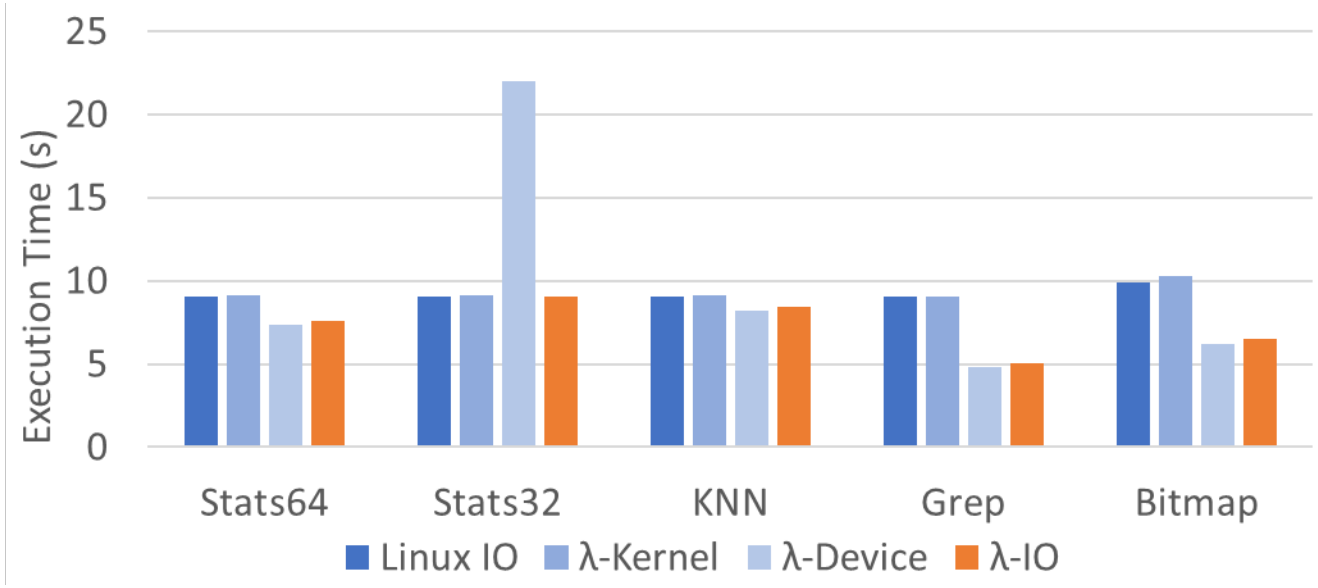
write

# Evaluation: Synthetic Application

- **Modes**
  - **Vanilla Linux IO:** Linux IO
  - **Computational Storage:** λ-Kernel, λ-Device, λ-IO

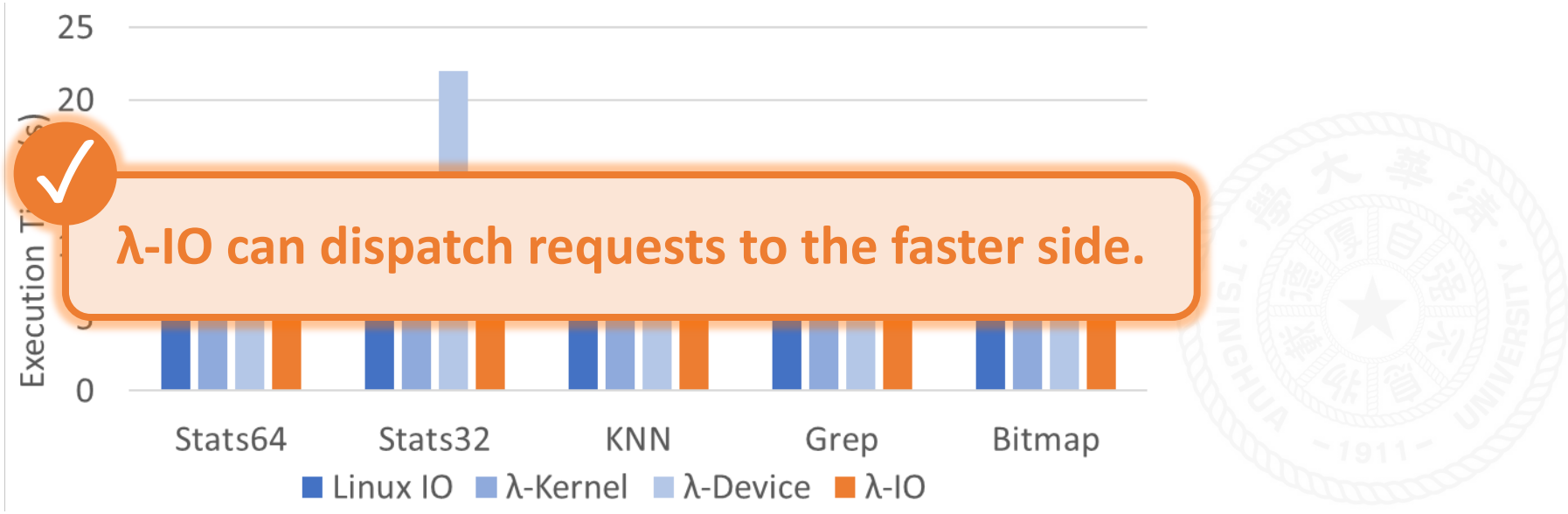
# Evaluation: Synthetic Application

- **Modes**
  - **Vanilla Linux IO:** Linux IO
  - **Computational Storage:** λ-Kernel, λ-Device, λ-IO
- **Without warmup**
  - Drop the page cache before each execution



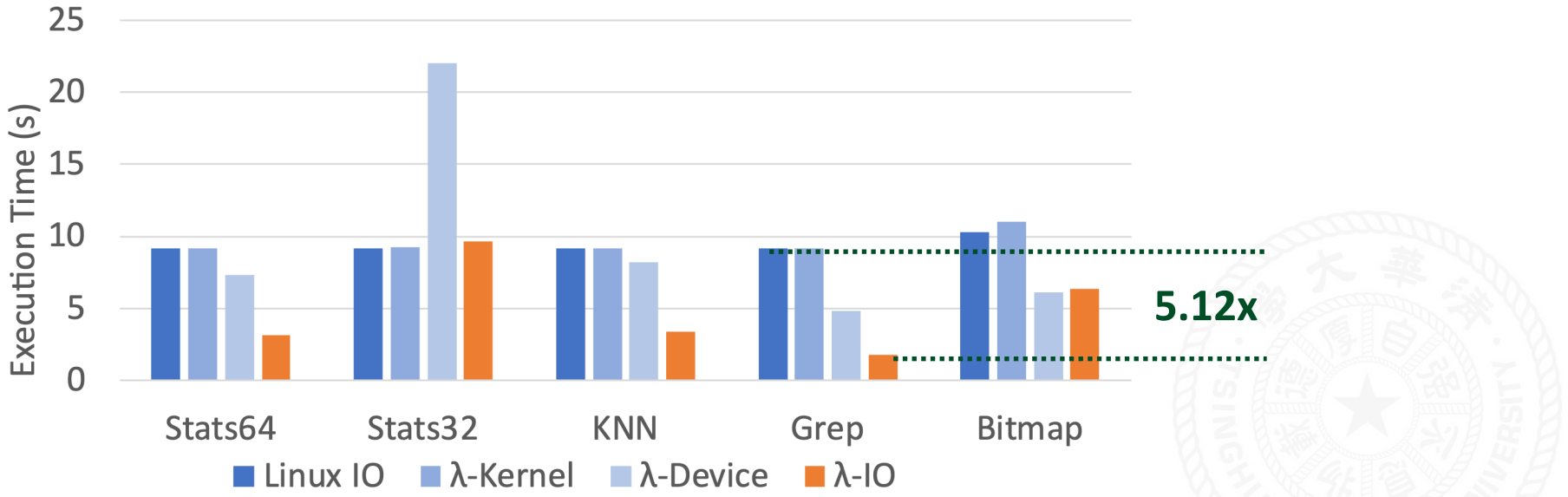
# Evaluation: Synthetic Application

- **Modes**
  - **Vanilla Linux IO:** Linux IO
  - **Computational Storage:** λ-Kernel, λ-Device, λ-IO
- **Without warmup**
  - Drop the page cache before each execution



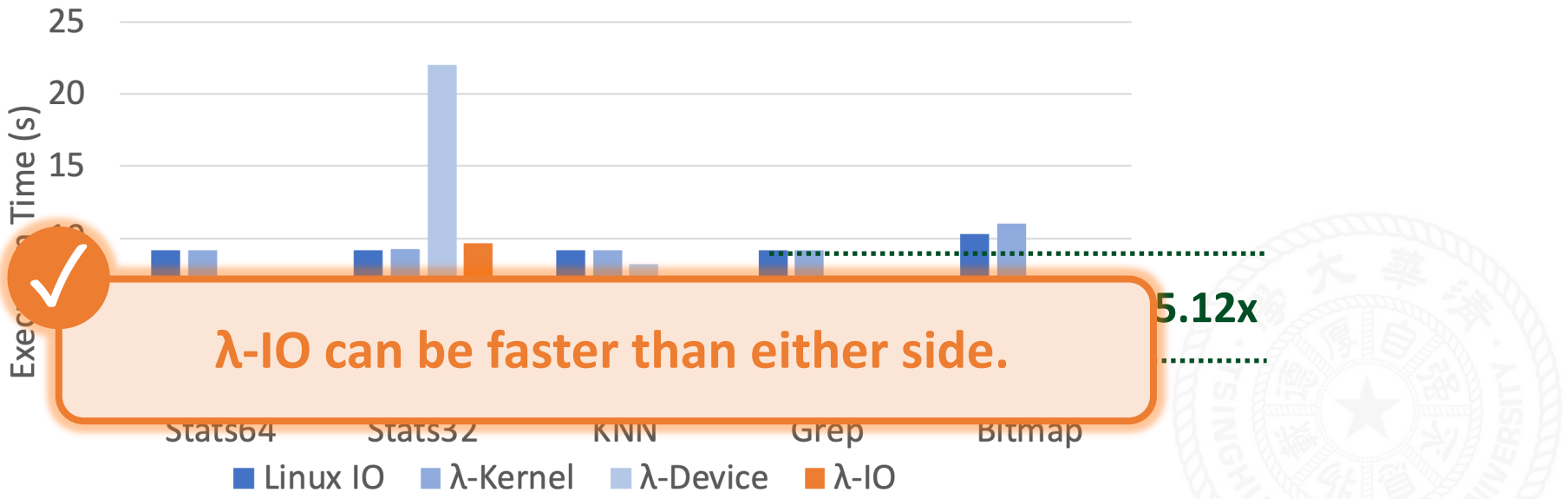
# Evaluation: Synthetic Application

- **With warmup**
  - Read the file sequentially by Buffer IO before each execution
  - Cached data → host, uncached data → device



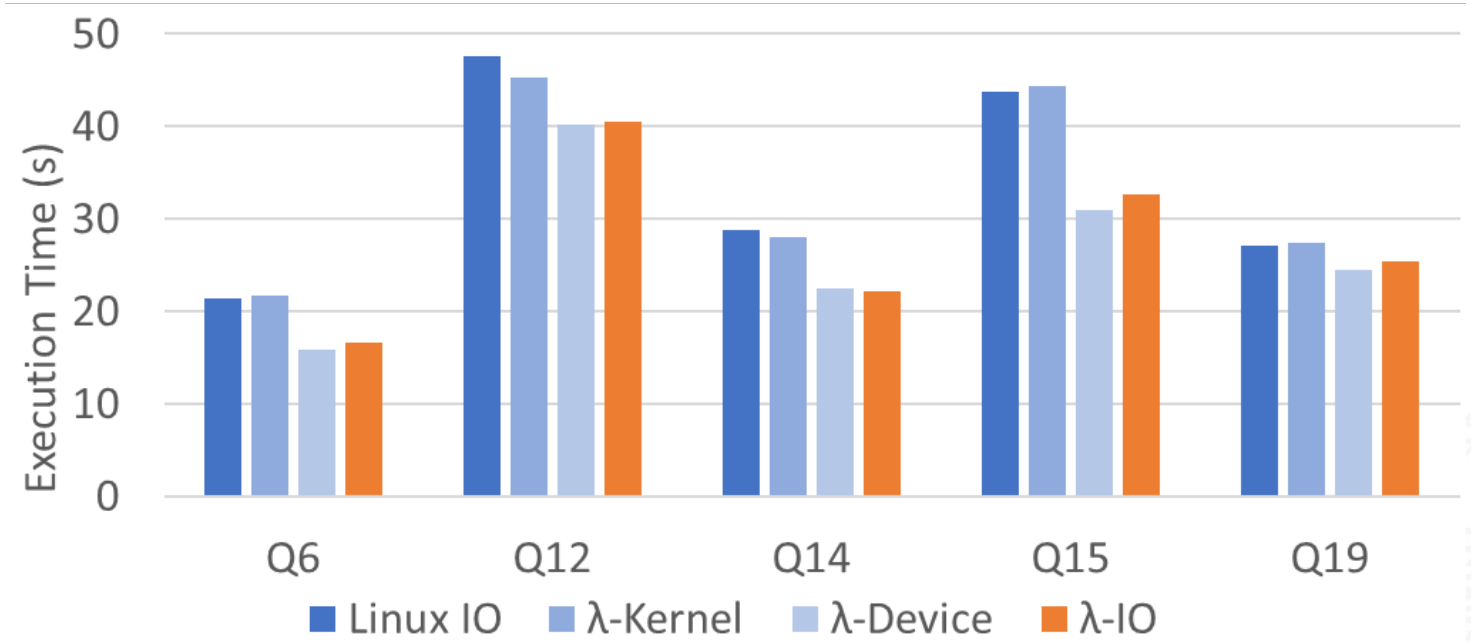
# Evaluation: Synthetic Application

- **With warmup**
  - Read the file sequentially by Buffer IO before each execution
  - Cached data → host, uncached data → device



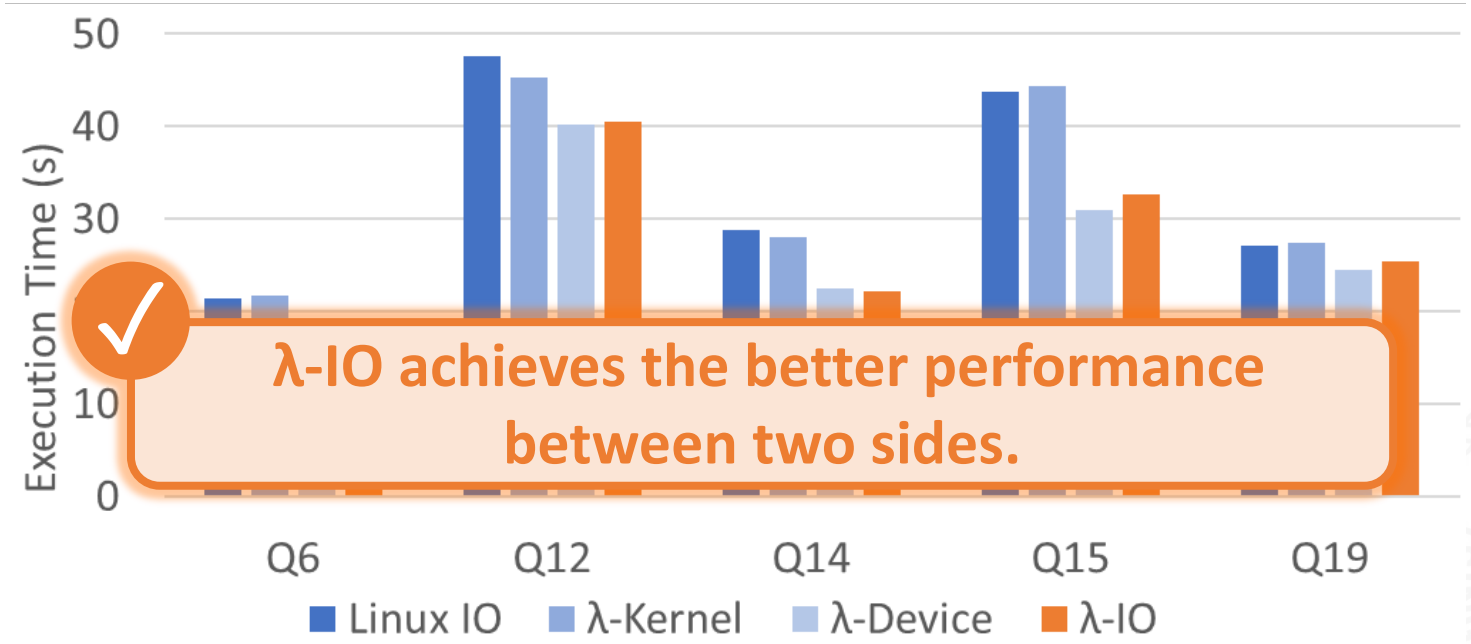
# Evaluation: Case Study - Spark SQL

- **TPC-H without warmup**
  - OLAP workloads



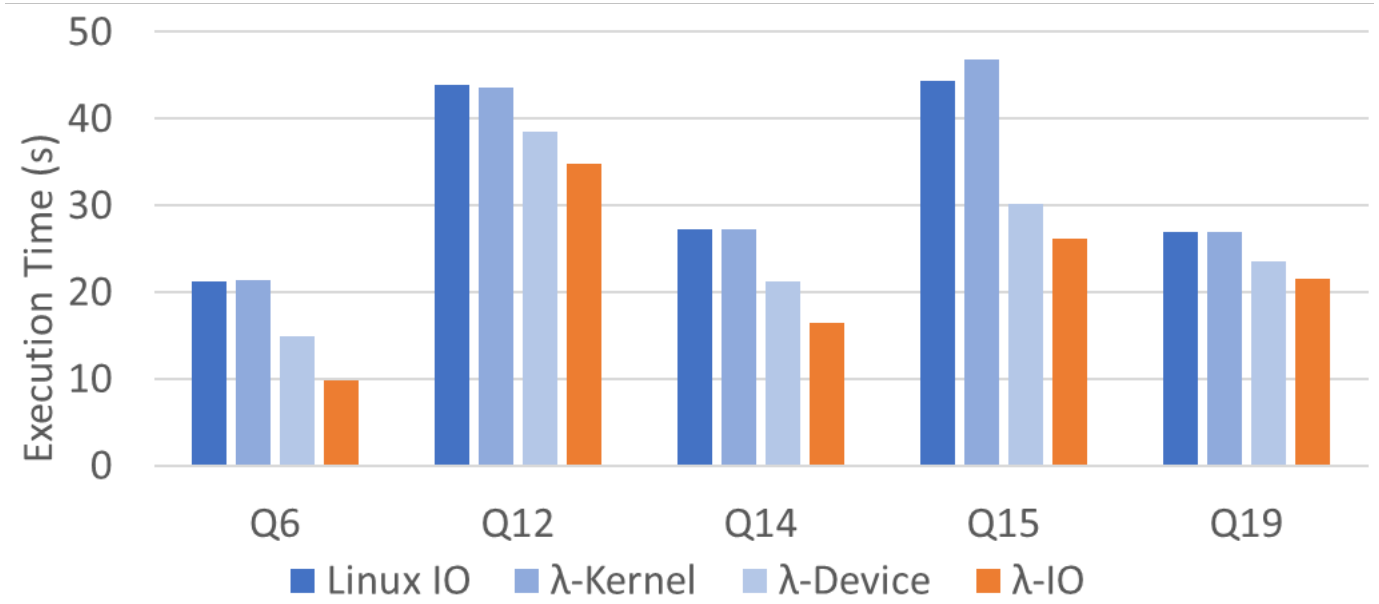
# Evaluation: Case Study - Spark SQL

- **TPC-H without warmup**
  - OLAP workloads



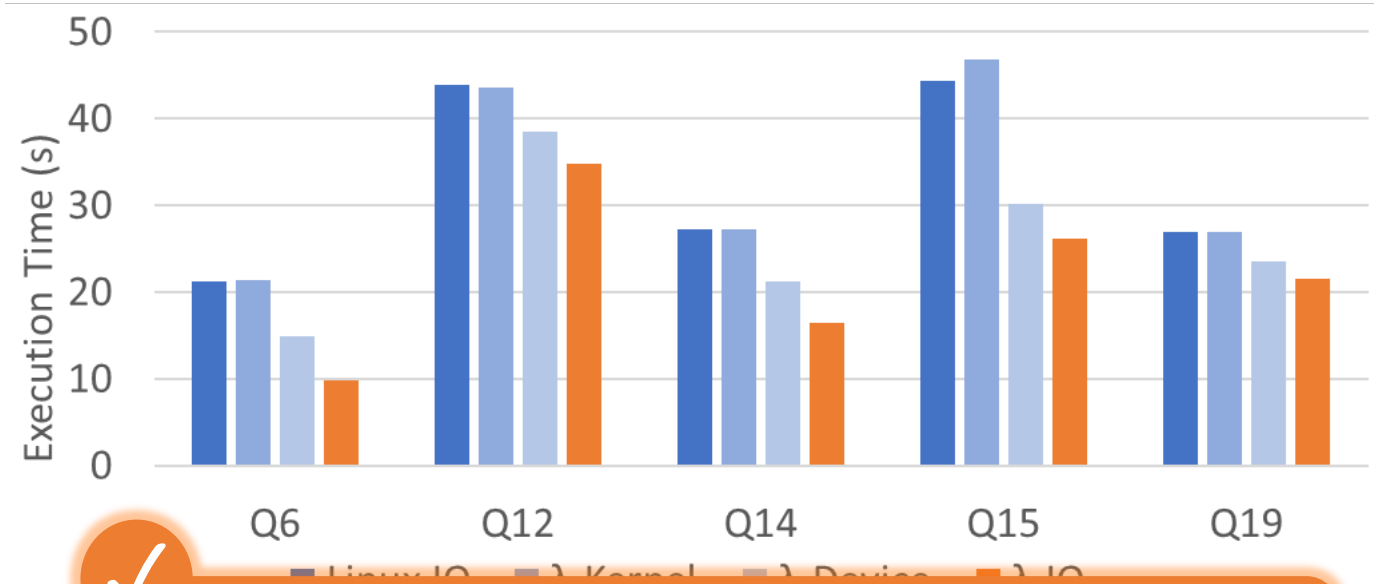
# Evaluation: Case Study - Spark SQL

- **TPC-H with warmup**



# Evaluation: Case Study - Spark SQL

- **TPC-H with warmup**




✓ **Real applications, e.g. Spark SQL, can benefit from λ-IO.**

# Conclusion

# Conclusion

- λ-IO is a unified IO stack managing computation and storage resources across the host and the device.

# Conclusion

- λ-IO is a unified IO stack managing computation and storage resources across the host and the device.

- λ-IO offloads a user-defined computational logic during read/write and dispatches requests between both sides.

# Conclusion

- λ-IO is a unified IO stack managing computation and storage resources across the host and the device.
- λ-IO offloads a user-defined computational logic during read/write and dispatches requests between both sides.
- λ-IO outperforms vanilla Linux IO significantly (up to 5.12x).



# Thanks! Q&A

# λ-IO: A Unified IO stack for Computational Storage

**Zhe Yang**, Youyou Lu, Xiaojian Liao, Youmin Chen,
Junru Li, Siyu He, Jiwu Shu

yangz18@mails.tsinghua.edu.cn

https://github.com/thustorage/lambda-io