A File I/O System for Many-core Based Clusters

Yuki Matsuo Taku Shimosawa Yutaka Ishikawa

University of Tokyo

ROSS 2012

June 29, 2012

Introduction

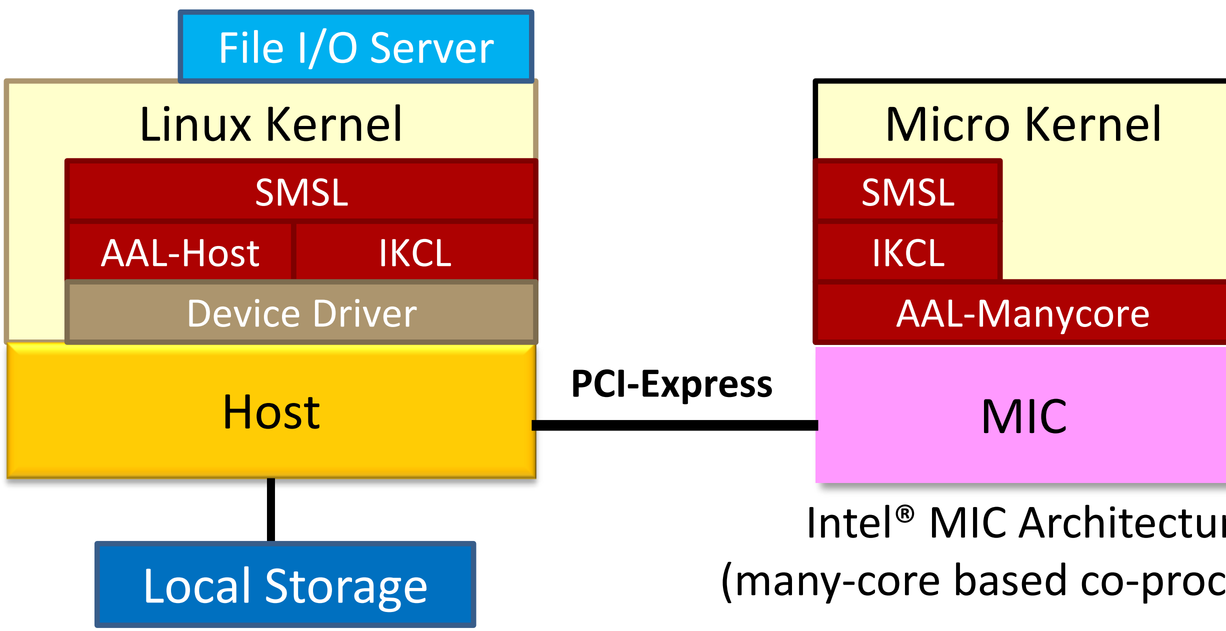
- Heterogeneous Systems
  - ➢ dedicated co-processors and general purpose multi-cores

- Dedicated co-processor
  - ➢ GPGPU
    - ✓ Number crunching work load is offloaded to GPGPU
  - ➢ Many-core based co-processor (Intel® MIC Architecture)
    - ✓ The whole of an application can be executed on the many-core
    - ✓ Applications running on the many-core may issue file I/O operations

⇨ <u>In this work, a file I/O system performed on the many-core is designed, implemented and evaluated</u>

Machine Environment

- MIC connected to the host processor with PCI-Express
- Local storage attached to the host processor
- A kernel currently developing from scratch runs on MIC
- File I/O server runs on the host



Intel® MIC Architecture
(many-core based co-processor)

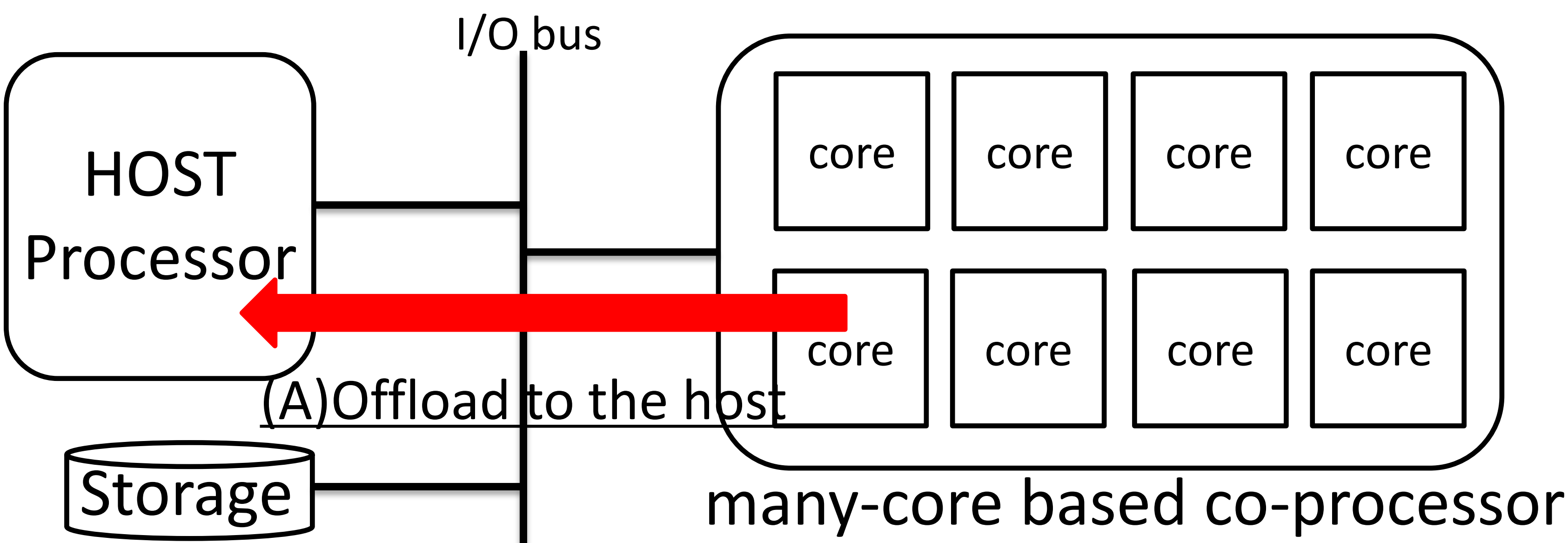
- Knights Ferry is adopted for implementation and evaluation
  - 32 core
  - L2 256KB/core
  - RAM 2GB

www.brightsideofnews.com - /Data/2011_6_20/Intel-Larrabee-Take-Two-Knights-Corner-in-2012-aims-ExaScale-2018/

File I/O performed on the many-core

- Actual file data should be transferred from the storage attached to the host processor
- Two mechanisms to perform file I/O operations on the many-core
  (A) Executing all procedures on the host
  (B) Executing as much procedures as possible on the many-core

⇨ Furthermore, two mechanisms can be considered for (B)



Three Mechanisms for File I/O on MIC

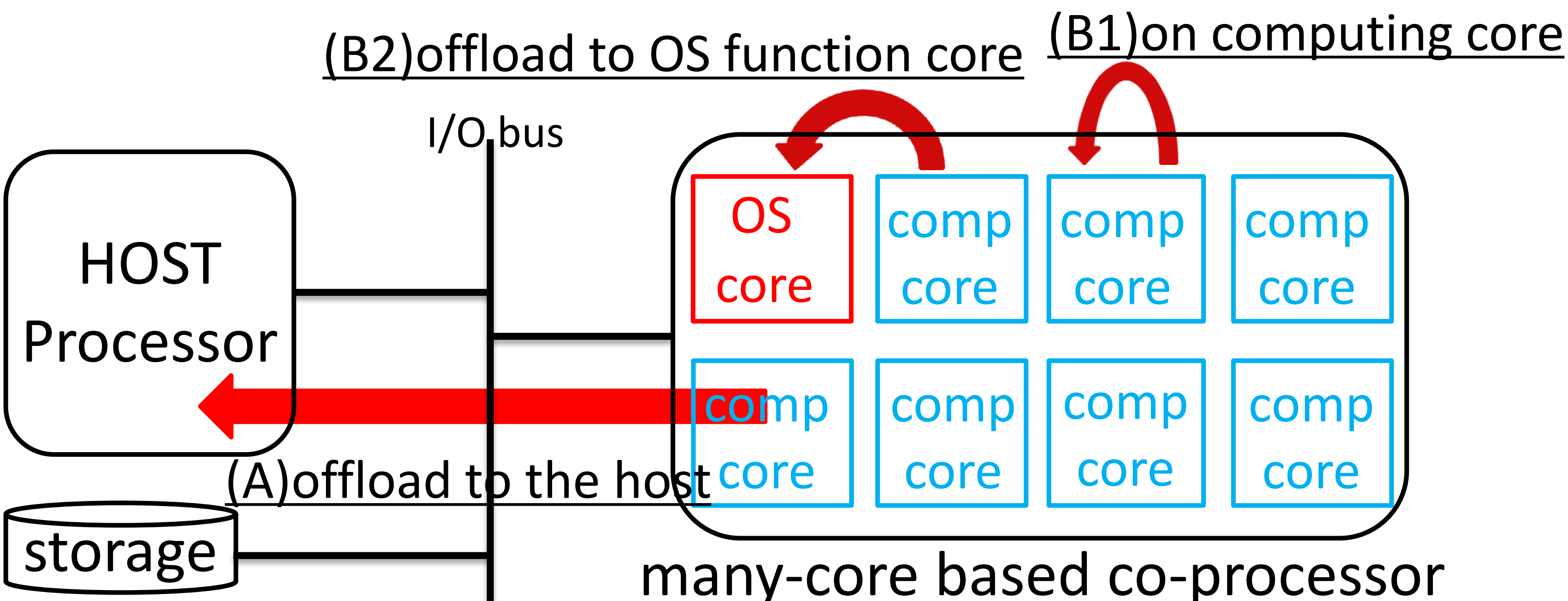
(A) Offloaded to the host processor

(B) Performed inside the many-core

  Possibility of cache pollution due to its small cache size

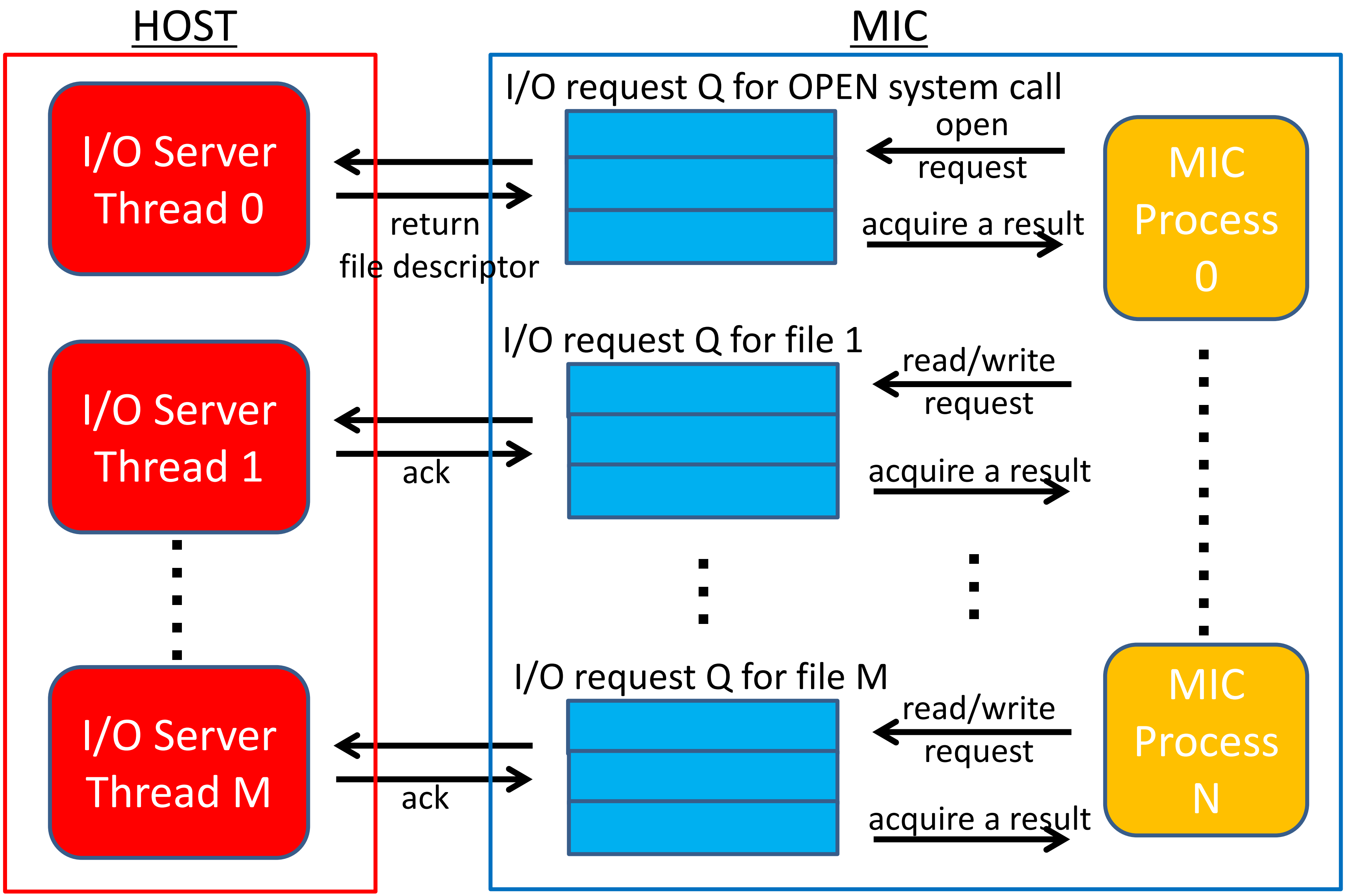
  (B1) Executing on computing core

  (B2) Executing on dedicated core for OS functionality



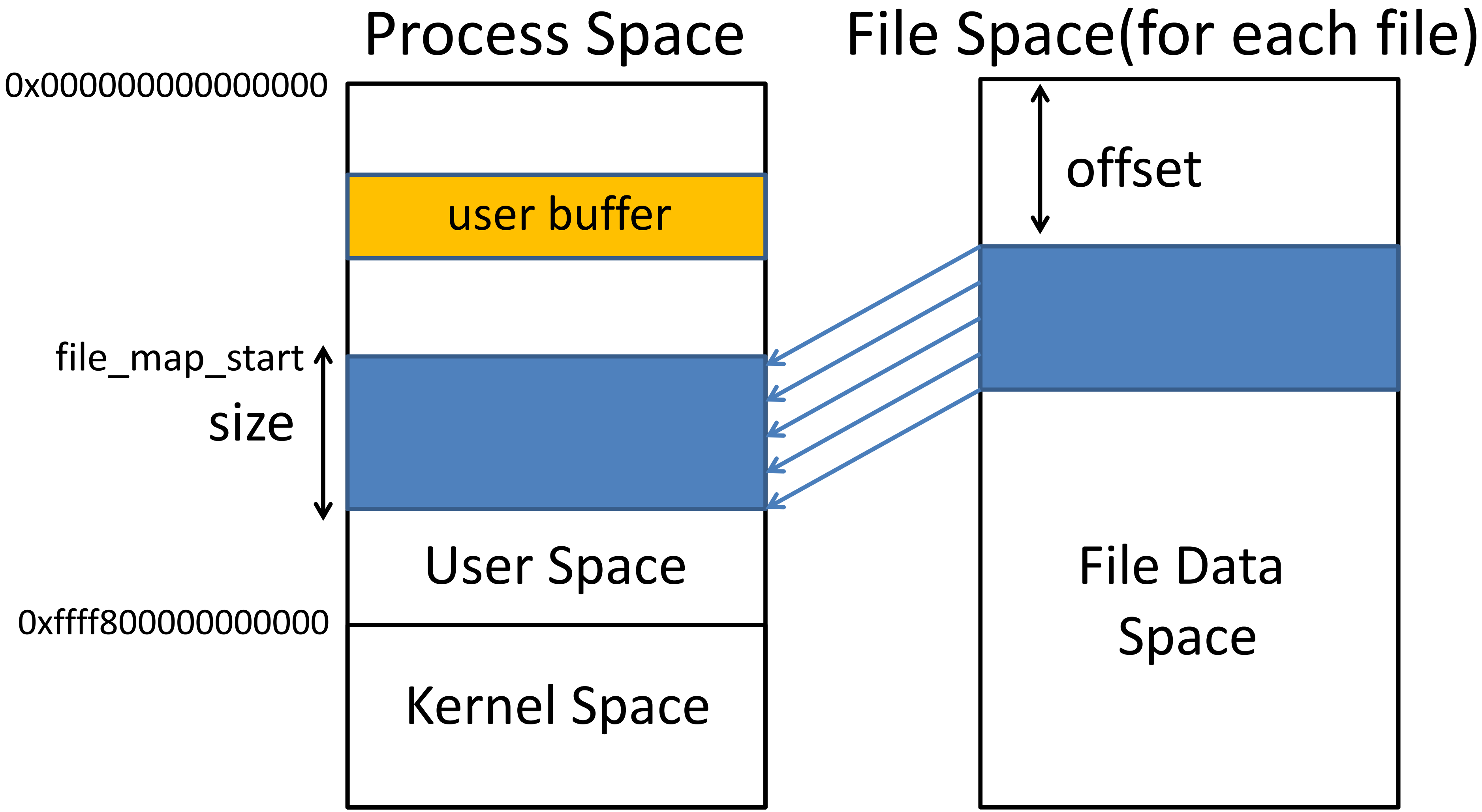
File I/O Server on the host Linux

- Each file I/O server thread polls a specified request queue



Design of File Cache on MIC

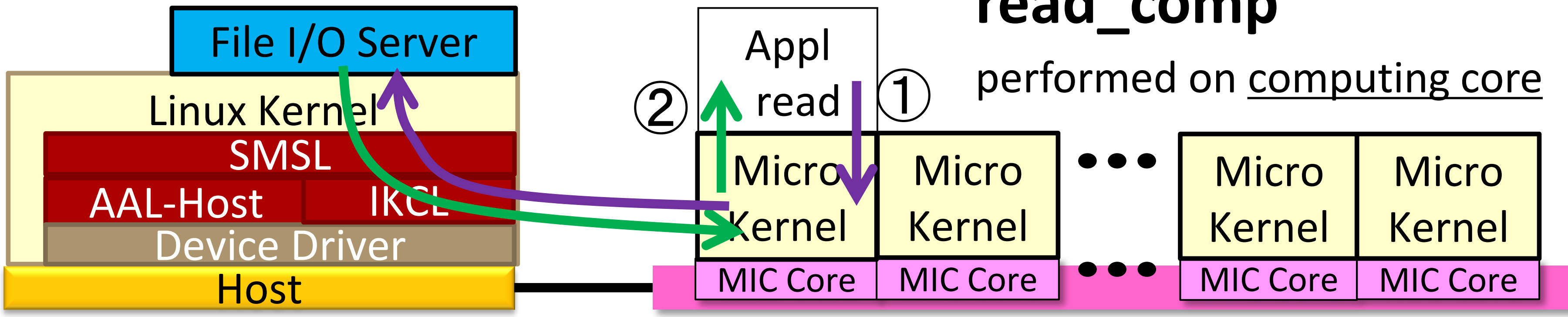
- Read/write system calls on computing core or OS function core are performed through file cache inside the many-core



Design of File I/O - Three kinds of read syscalls

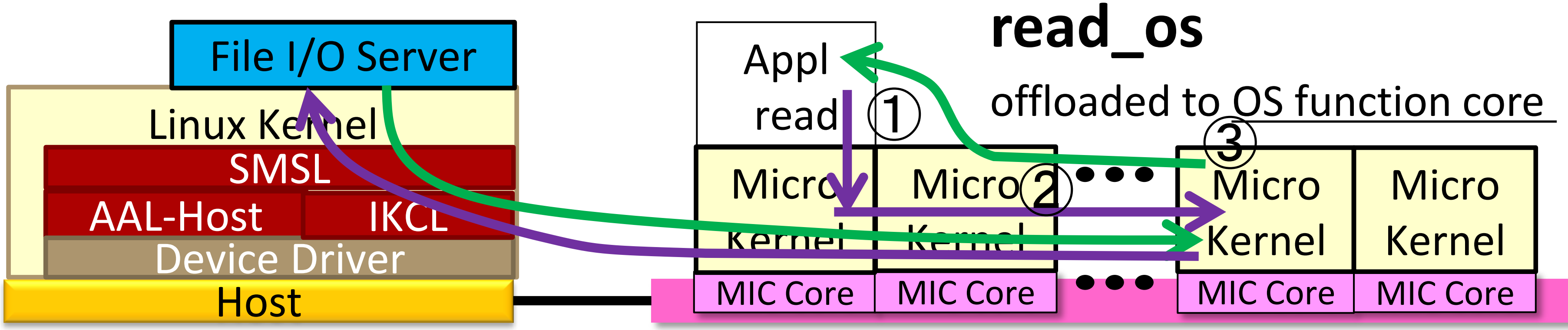
read_comp

performed on computing core



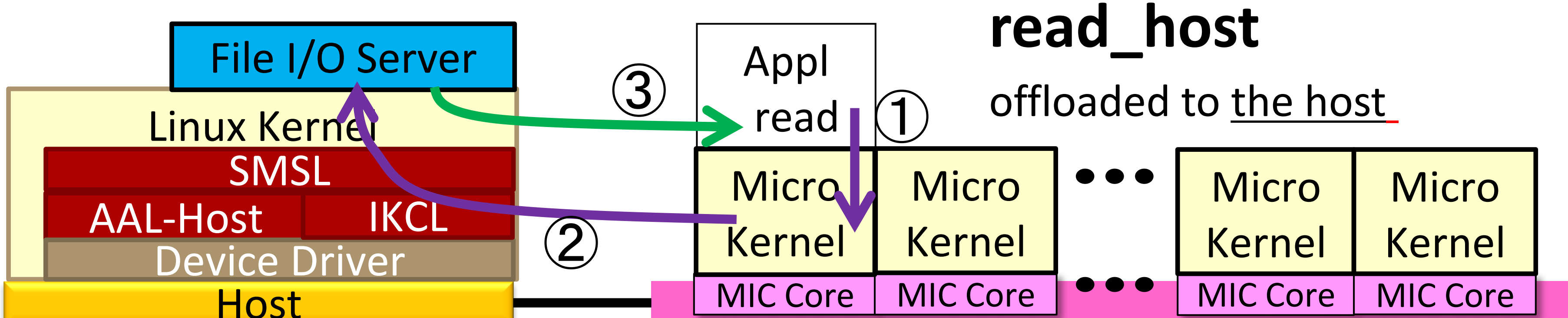
read_os

offloaded to OS function core

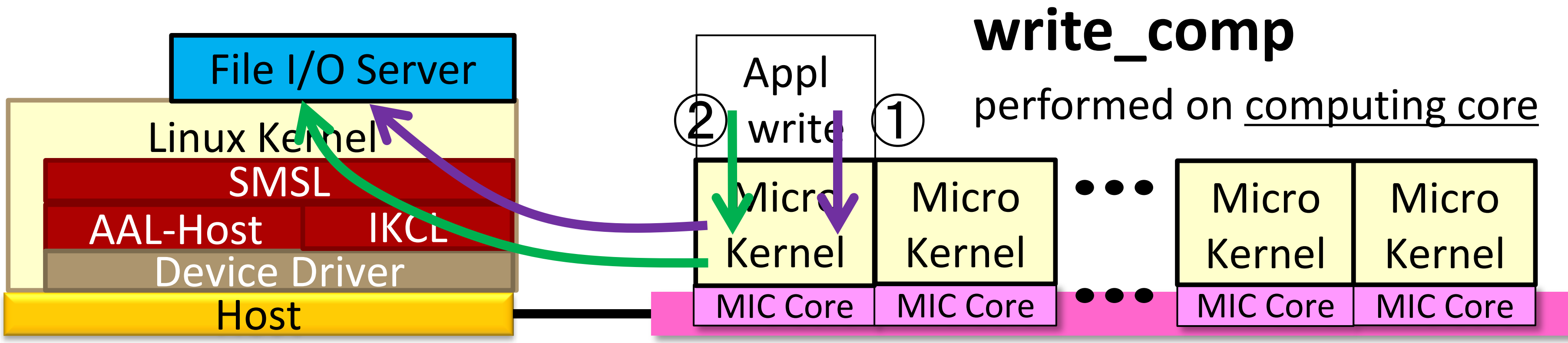


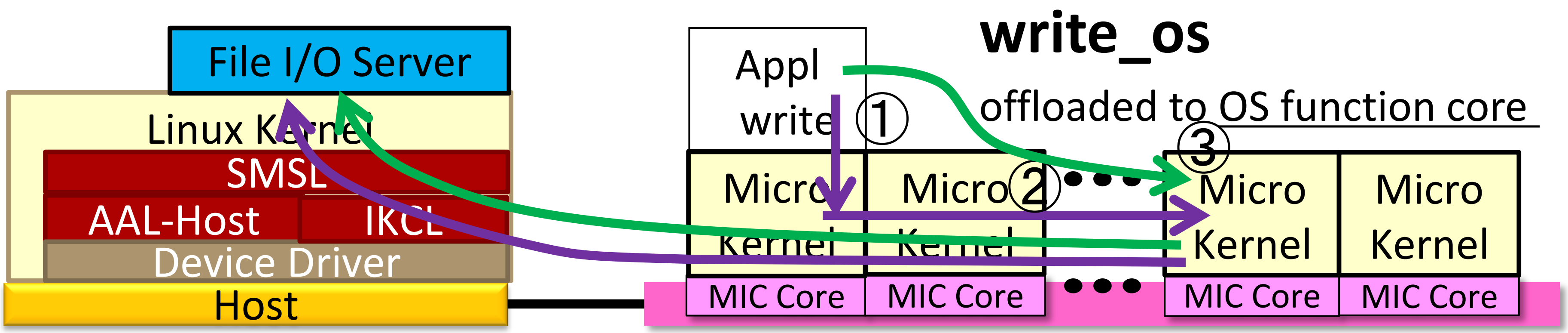
read_host

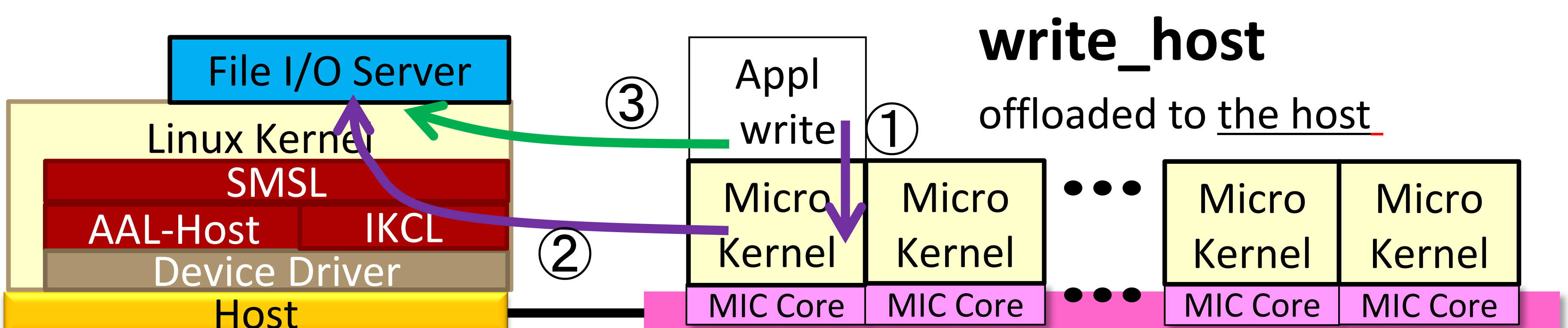
offloaded to the host



Design of File I/O - Three kinds of write syscalls

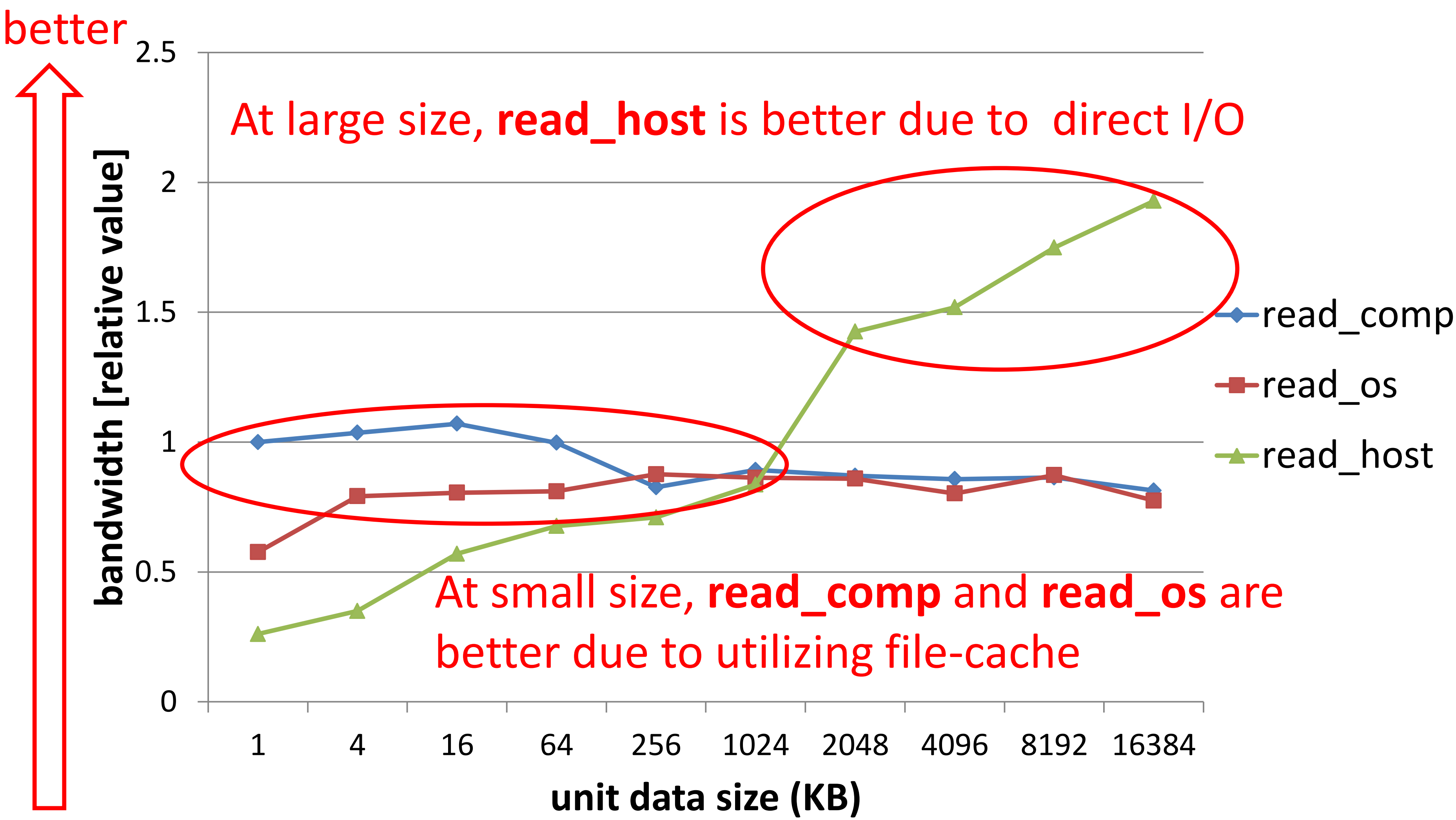






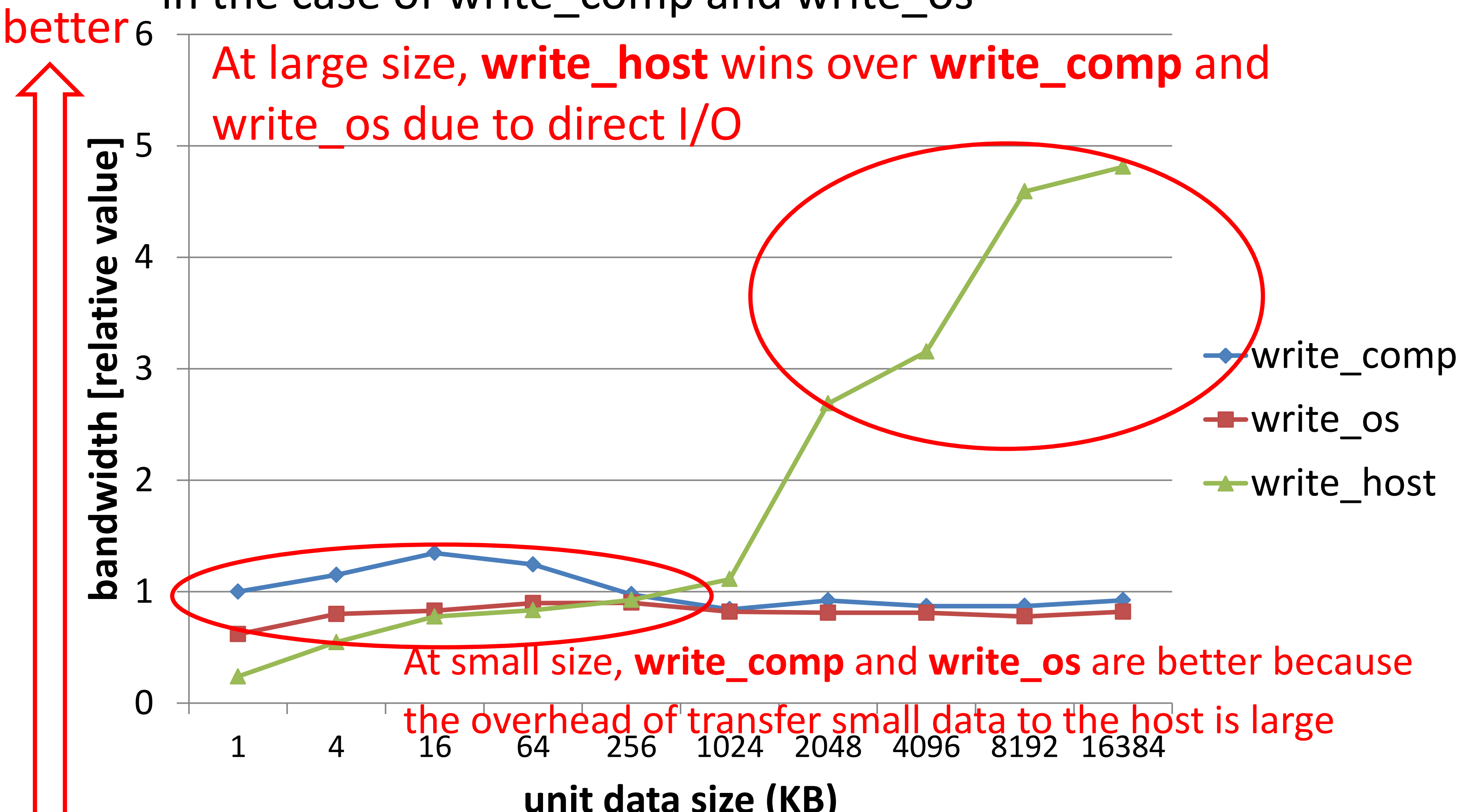
Bandwidth of Read System Calls

- In order to ascertain the positive effect of file cache on the many-core, sequential read of a file(total 16MB) is performed



Bandwidth of Write System Calls

- Sequential write of total 16MB
- **sync** system call is executed at the end of the evaluation in the case of write_comp and write_os



Read Benchmark

- Total read size is 16MB
- The total time to run the benchmark is evaluated

<u>iterative</u>

```
sum = 0;
for(n = 0;  n < DIVISOR;  n++) {
        read(fd, buf, size);
        for(i = 0;  i < size/4;  i++) {
                sum += buf[i];
        }
}
```

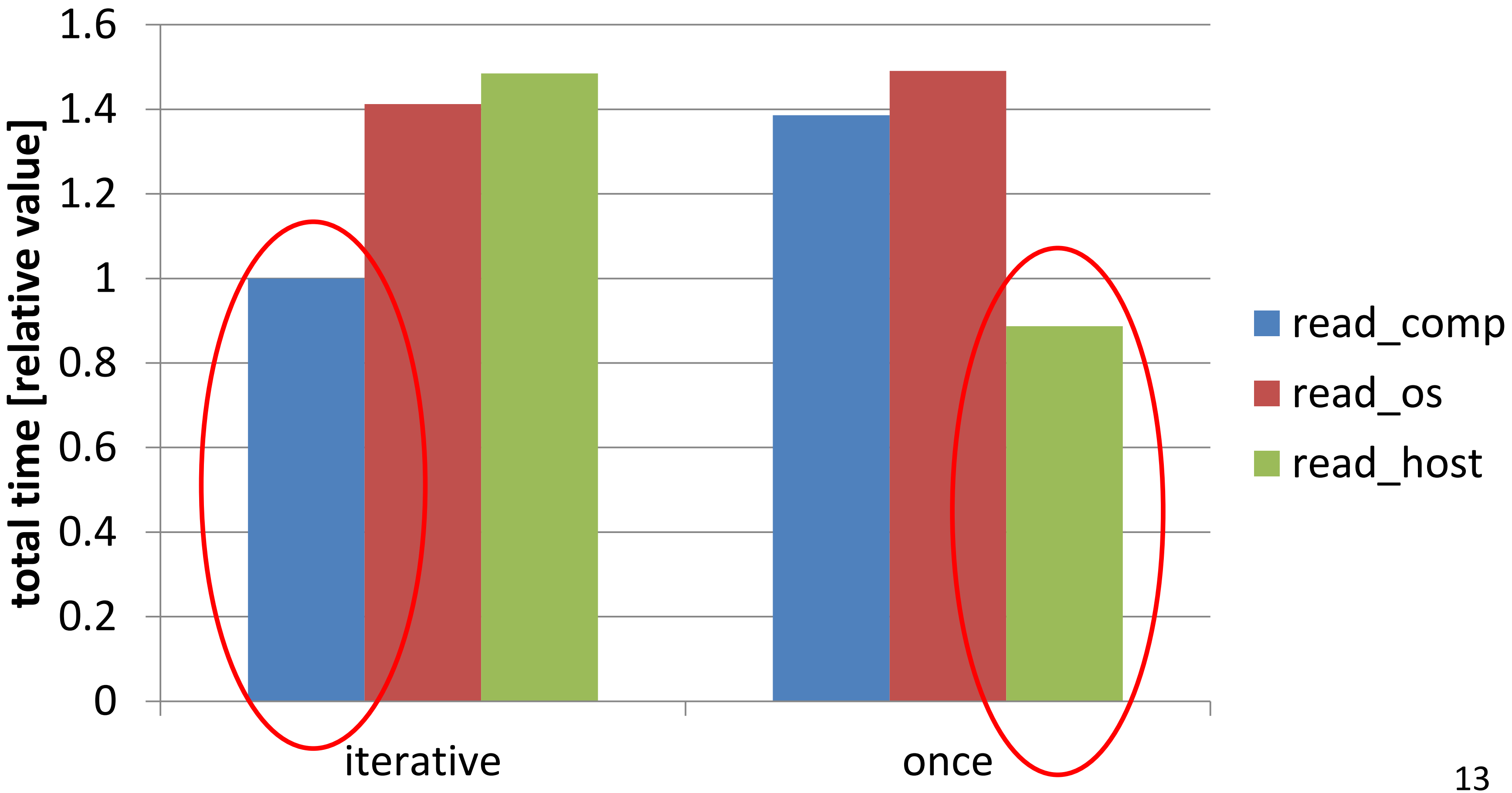
<u>once</u>

```
j = 0;
sum = 0;
read(fd, buf, size*DIVISOR);
for(n = 0;  n < DIVISOR;  n++) {
        for(i = 0;  i < size/4;  i++) {
           sum += buf[j++];
        }
}
```

<u>unit data size(64KB) < L2 cache size(256KB)</u>

Read Benchmark - Result

- The best: **read_host** in the one time benchmark
  - ➢ large bandwidth
- The second best: **read_comp** in the iterative benchmark
  - ➢ user buffer data exists on L2 cache when the user code try to access it



Write Benchmark

- Total write size is 16MB
- The total time to run the benchmark is evaluated

<u>iterative</u>

```
for(n = 0;  n < DIVISOR;  n++) {
        for(i = 0;  i < size/4;  i++) {
                buf[i] = n;
        }

        write(fd, buf, size);
}
```

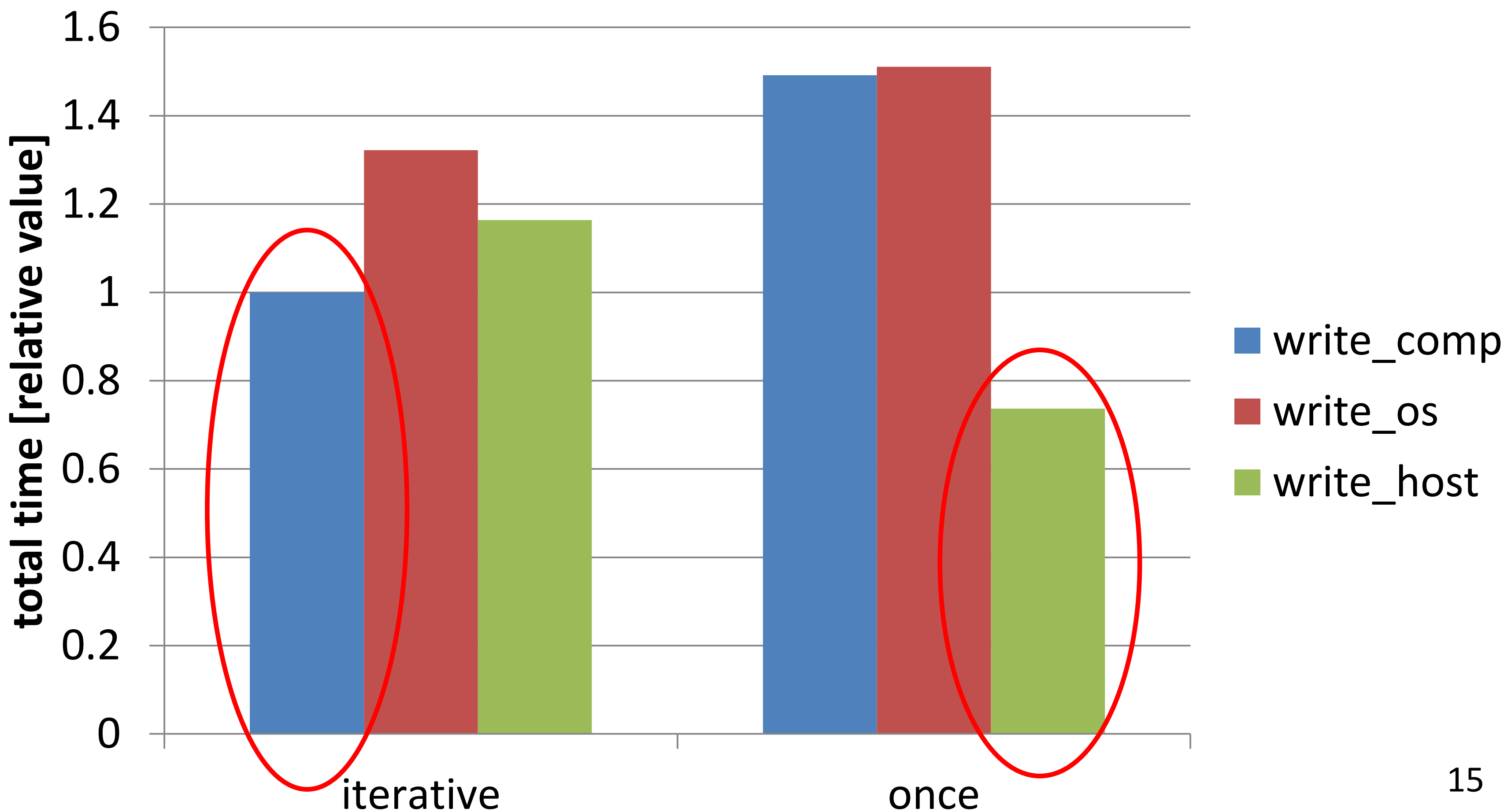
<u>once</u>

```
j = 0;
for(n = 0;  n < DIVISOR;  n++) {
        for(i = 0;  i < size/4;  i++) {
            buf[j++] = n;
        }
}
write(fd, buf, size*DIVISOR);
```

<u>unit data size(64KB) < L2 cache size(256KB)</u>

Write Benchmark - Result

- The best: **write_host** in the one time benchmark
  - ➢ Large bandwidth
- The second best: **write_comp** in the iterative benchmark
  - ➢ Write system call can be executed efficiently because of user buffer exists on L2 cache



Related Work

- Shimizu et al. (2010)
  - Remote file I/O for heterogeneous cluster system
  - Direct I/O between I/O node and user buffer in computing node
  - High bandwidth at large data, low bandwidth at small data

⇨ In our work, the bandwidth can maintain high value at small data size by introducing file cache on the many-core

- Soares et al. (2010)
  - FlexSC: Flexible System Call Scheduling with Exception-Less System Calls
  - Negative effects of executing system calls on user program code
    - ✓ Cost of switching the privilege mode
    - ✓ Cache pollution caused by the system call

⇨ Where the data is utilized in the user code should also be considered when discussing file I/O system call's foot print

Summary

- A file I/O system performed on many-core based co-processor connected to the high performance host
  - Three types of file I/O system calls
    - ✓ Performed on computing core in the many-core
    - ✓ Offloaded to OS function core in the many-core
    - ✓ Offloaded to the host

- The bandwidth of file I/O system calls
  - At small data, the system calls performed inside the many-core are better
  - At large data, the system call offloaded to the host wins

- Total execution time of simple read/write benchmarks
  - The bandwidth of file I/O system calls has more significant effect rather than the factor that the data exists on the CPU cache.

Thank you