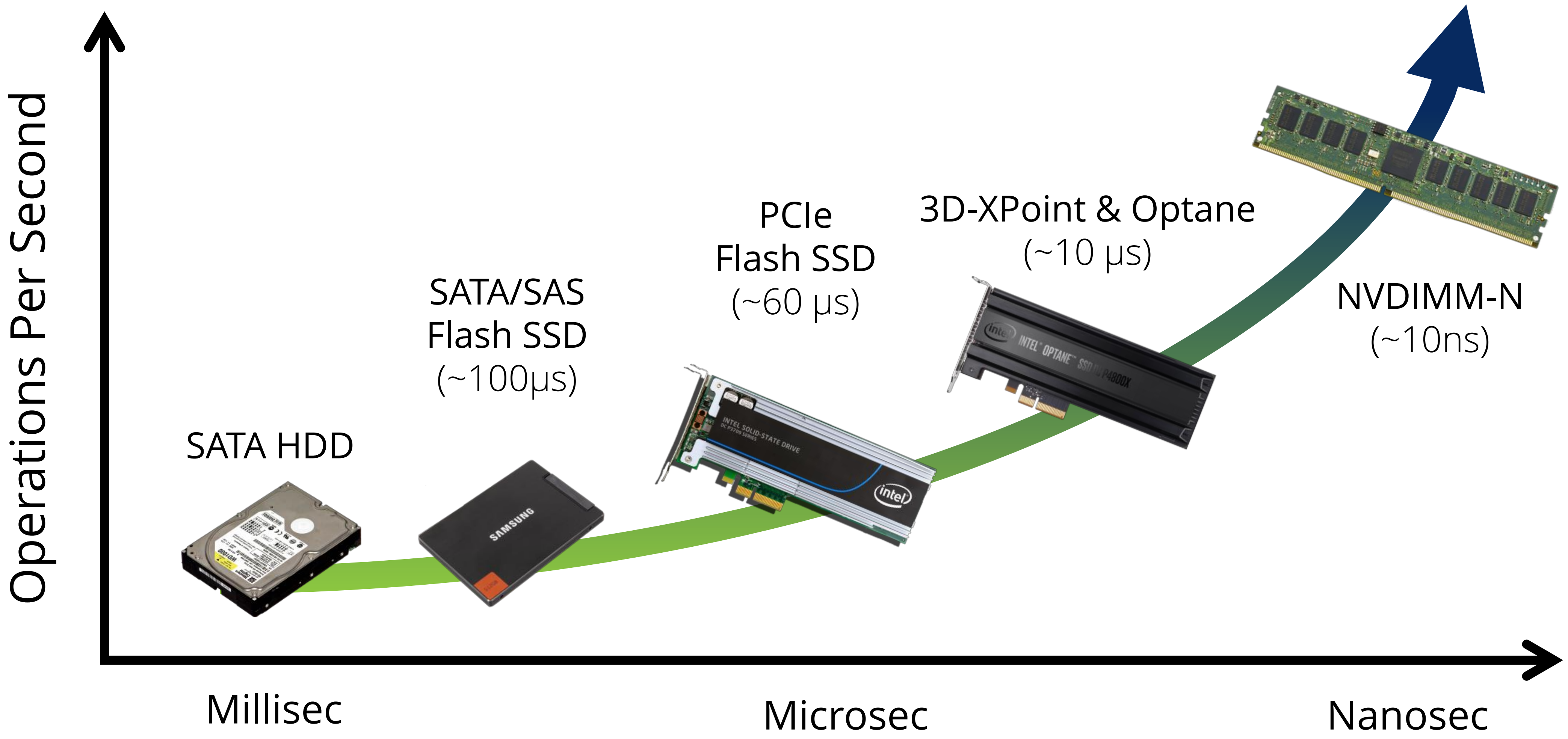
Efficient Memory Mapped File I/O for In-Memory File Systems

Jungsik Choi[†], Jiwon Kim, Hwansoo Han

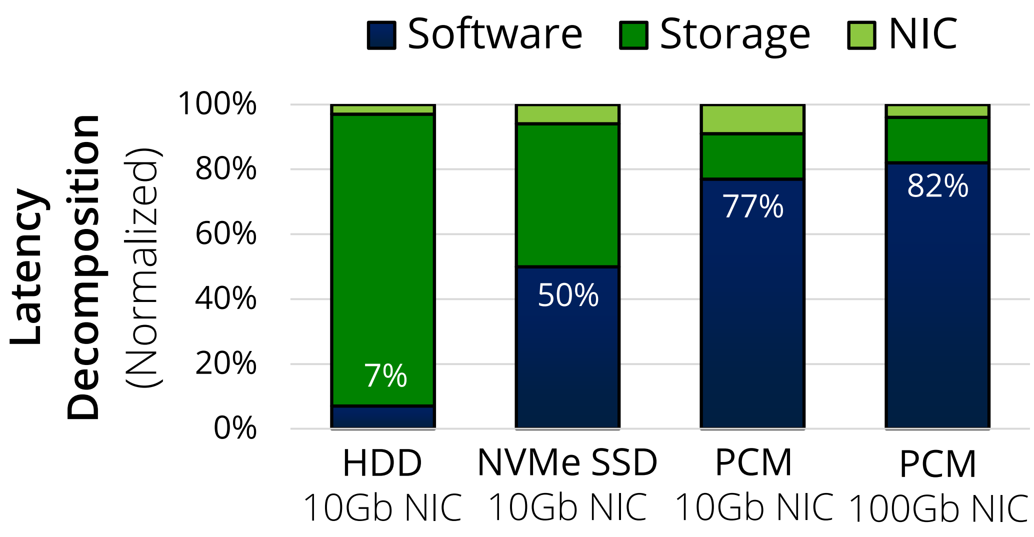
SUNG KYUN KWAN UNIVERSITY

Storage Latency Close to DRAM



Large Overhead in Software

- Existing OSes were designed for fast CPUs and slow block devices
- With low-latency storage, SW overhead becomes the largest burden
- Software overhead includes
  - Complicated I/O stacks
  - Redundant memory copies
  - Frequent user/kernel mode switches
- SW overhead must be addressed to fully exploit low-latency NVM storage



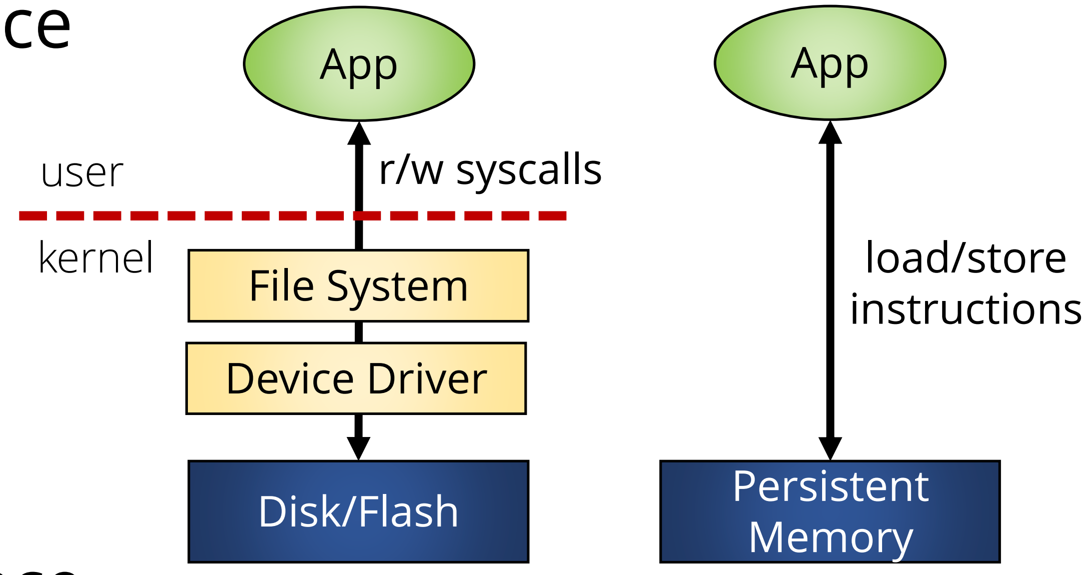
(Source: Ahn et al. *MICRO-48*)

Eliminate SW Overhead with `mmap`

- In recent studies, memory mapped I/O has been commonly proposed
  - Memory mapped I/O can expose the NVM storage's raw performance

- Mapping files onto user memory space
  - Provide memory-like access
  - Avoid complicated I/O stacks
  - Minimize user/kernel mode switches
  - No data copies

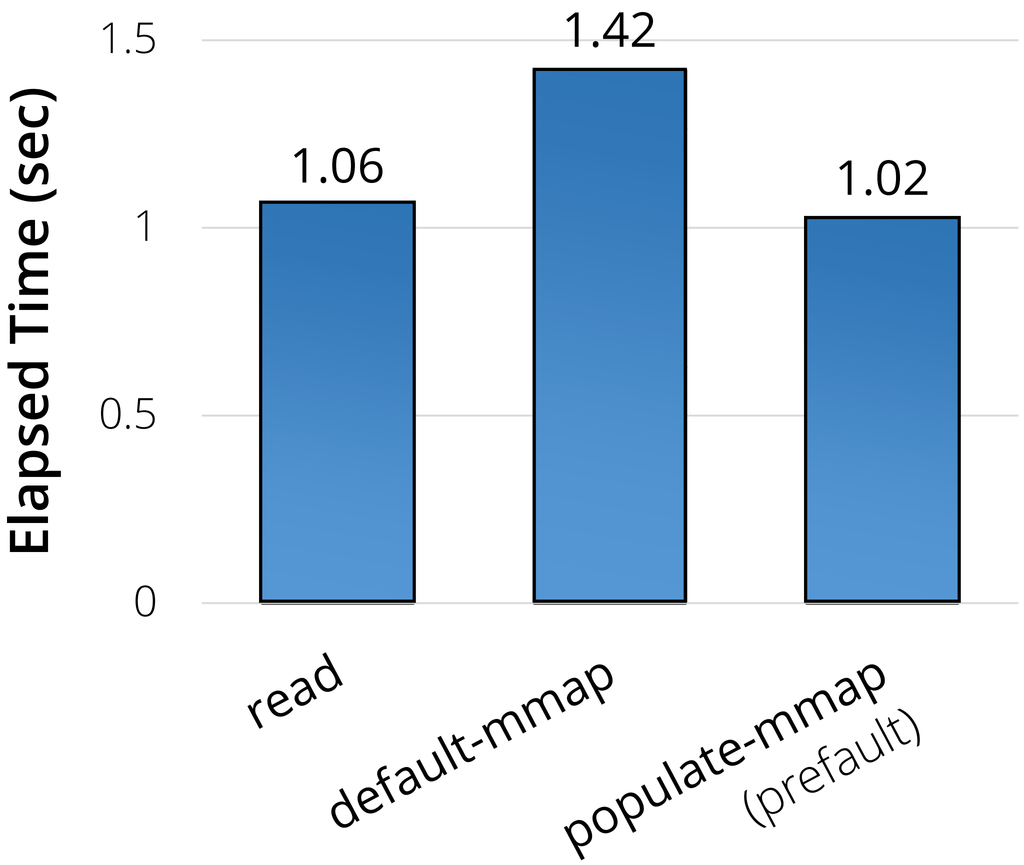
- A `mmap` syscall will be a critical interface



Traditional I/O   Memory mapped I/O

Memory Mapped I/O is Not as Fast as Expected

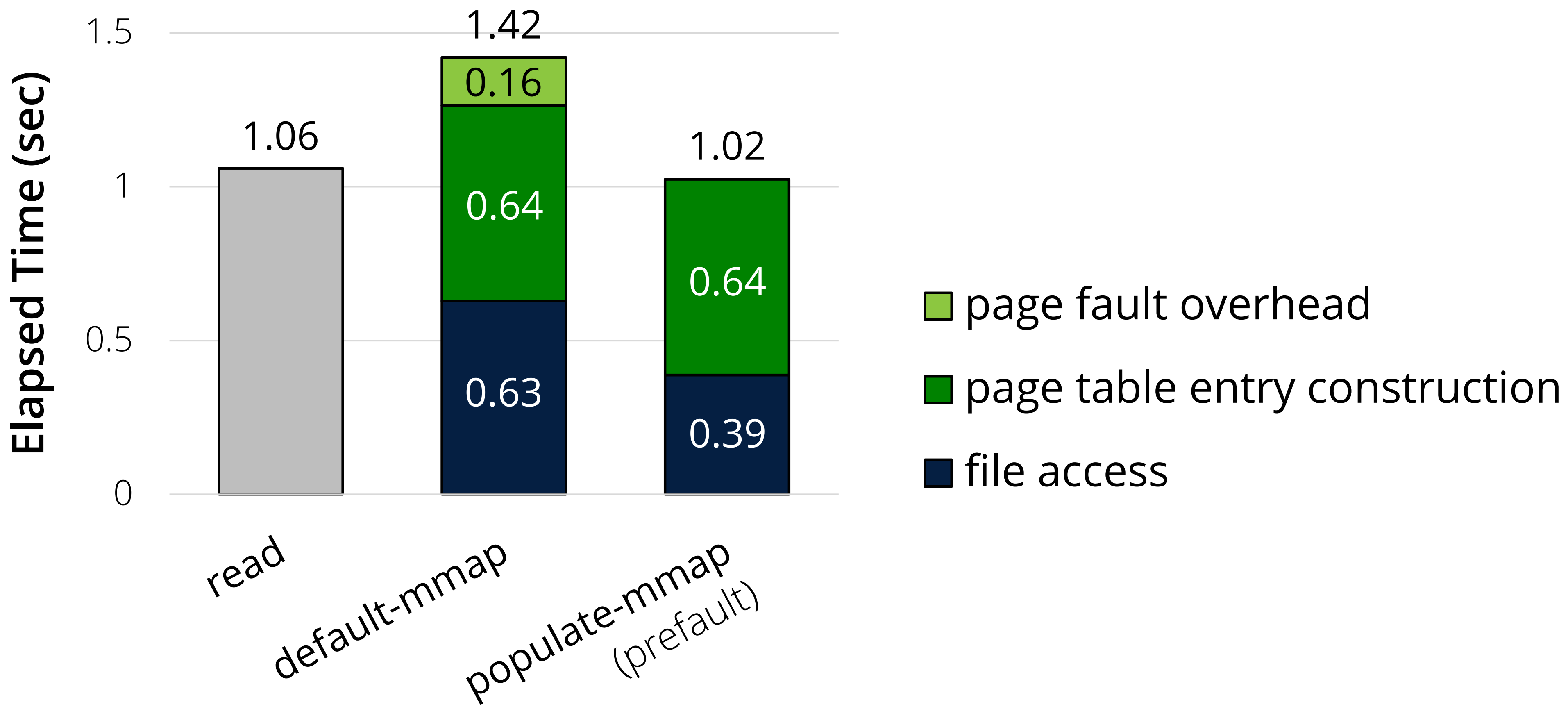
- Microbenchmark: sequential read, a 4GB file (Ext4-DAX on NVDIMM-N)



- read :
  `read` system calls
- default-mmap :
  `mmap` without any special flags
- populate-mmap :
  `mmap` with a `MAP_POPULATE` flag

Memory Mapped I/O is Not as Fast as Expected

- Microbenchmark: sequential read, a 4GB file (Ext4-DAX on NVDIMM-N)



Overhead in Memory Mapped File I/O

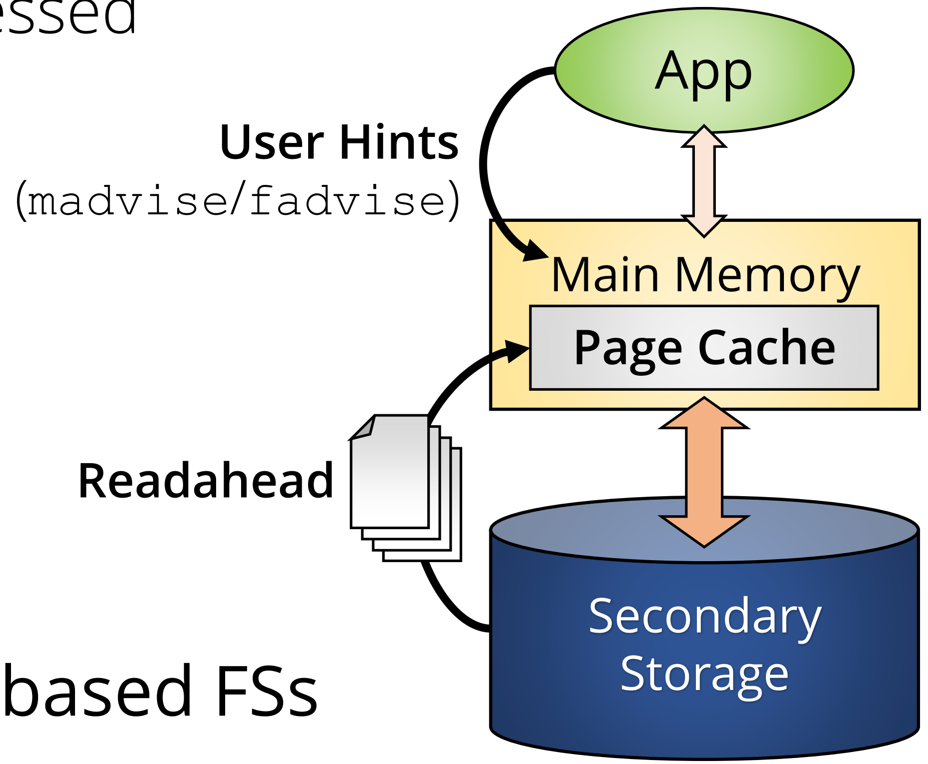
- Memory mapped I/O can avoid the SW overhead of traditional file I/O

- However, Memory mapped I/O causes another SW overhead
  - Page fault overhead
  - TLB miss overhead
  - PTE construction overhead

- It decreases the advantages of memory mapped file I/O

Techniques to Alleviate Storage Latency

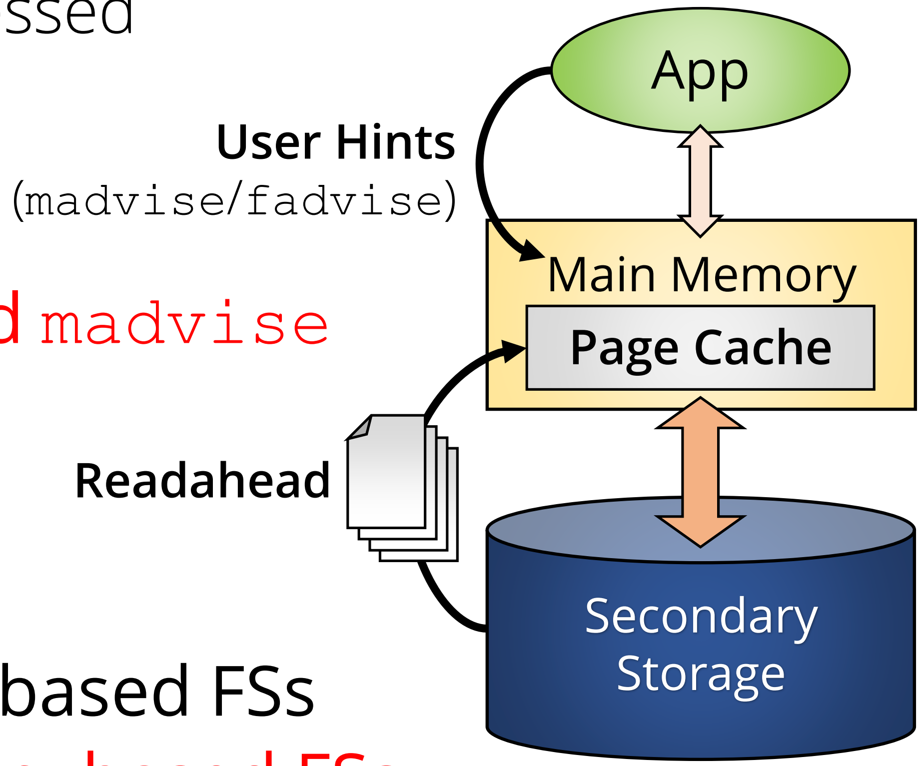
- Readahead
  - Preload pages that are expected to be accessed
- Page cache
  - Cache frequently accessed pages
- `fadvise`/`madvise` interfaces
  - Utilize user hints to manage pages



- However, these can't be used in memory based FSs

Techniques to Alleviate Storage Latency

- Readahead ⇒ Map-ahead
  - Preload pages that are expected to be accessed
- Page cache ⇒ Mapping cache
  - Cache frequently accessed pages
- `fadvise`/`madvise` interfaces ⇒ Extended `madvise`
  - Utilize user hints to manage pages

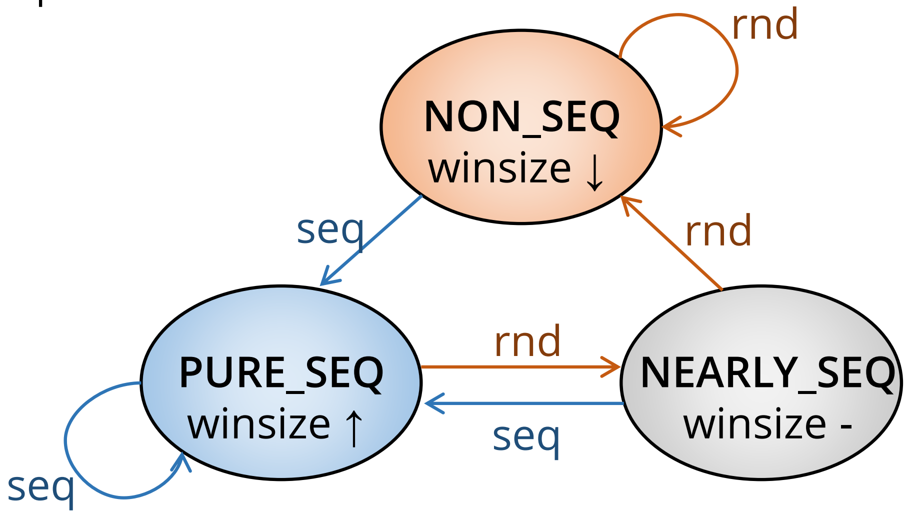


- However, these can't be used in memory based FSs

  ⇒ New optimization is needed in memory based FSs

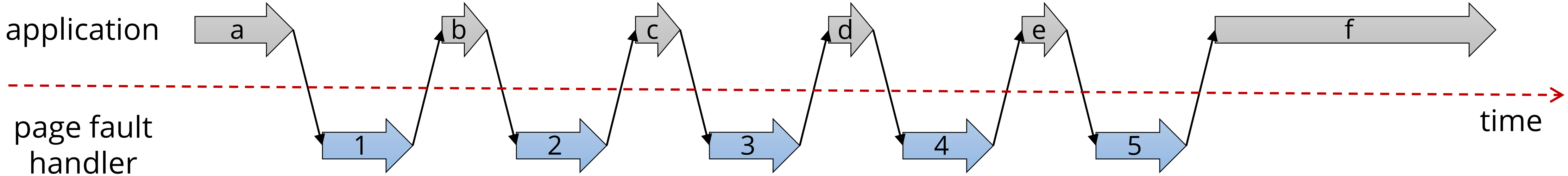
Map-ahead Constructs PTEs in Advance

- When a page fault occurs, the page fault handler handles
  - A page that caused the page fault (existing demand paging)
  - Pages that are expected to be accessed (map-ahead)
- Kernel analyzes page fault patterns to predict pages to be accessed
  - Sequential fault : map-ahead window size ↑
  - Random fault : map-ahead window size ↓
- Map-ahead can reduce # of page faults



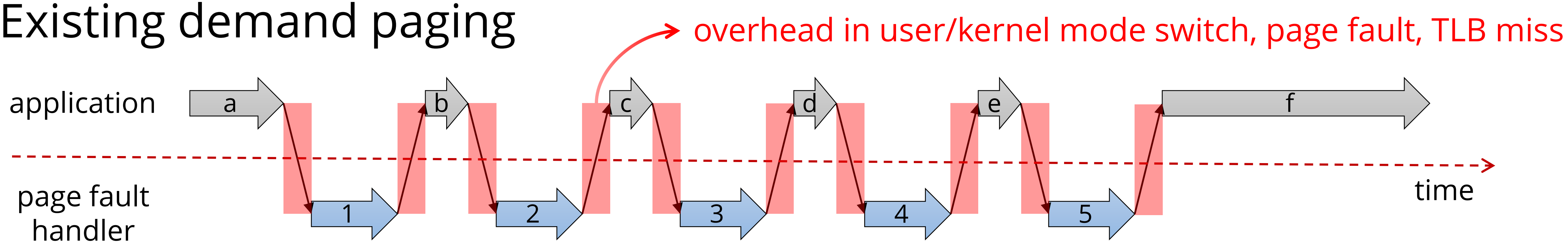
Comparison of Demand Paging & Map-ahead

- Existing demand paging



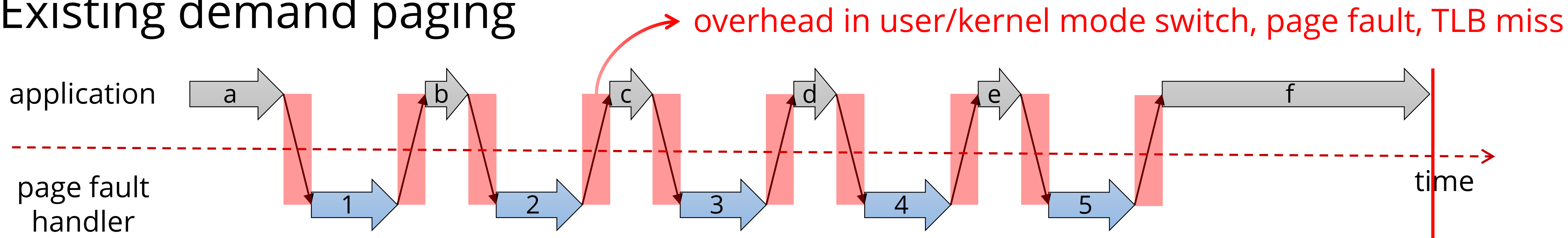
Comparison of Demand Paging & Map-ahead

- Existing demand paging

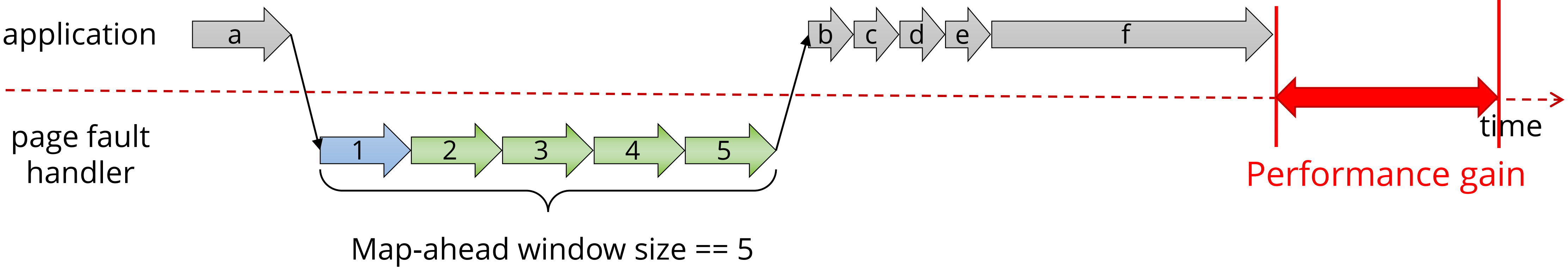


Comparison of Demand Paging & Map-ahead

- Existing demand paging

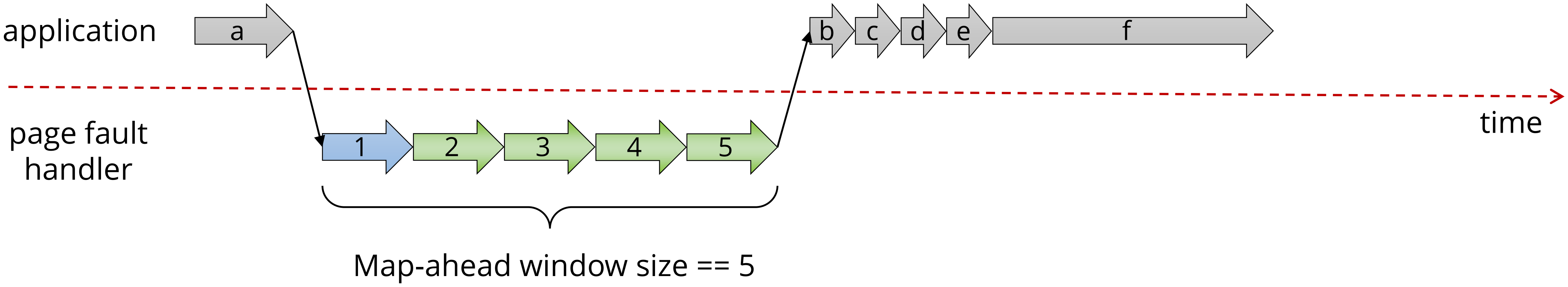


- Map-ahead



Async Map-ahead Hides Mapping Overhead

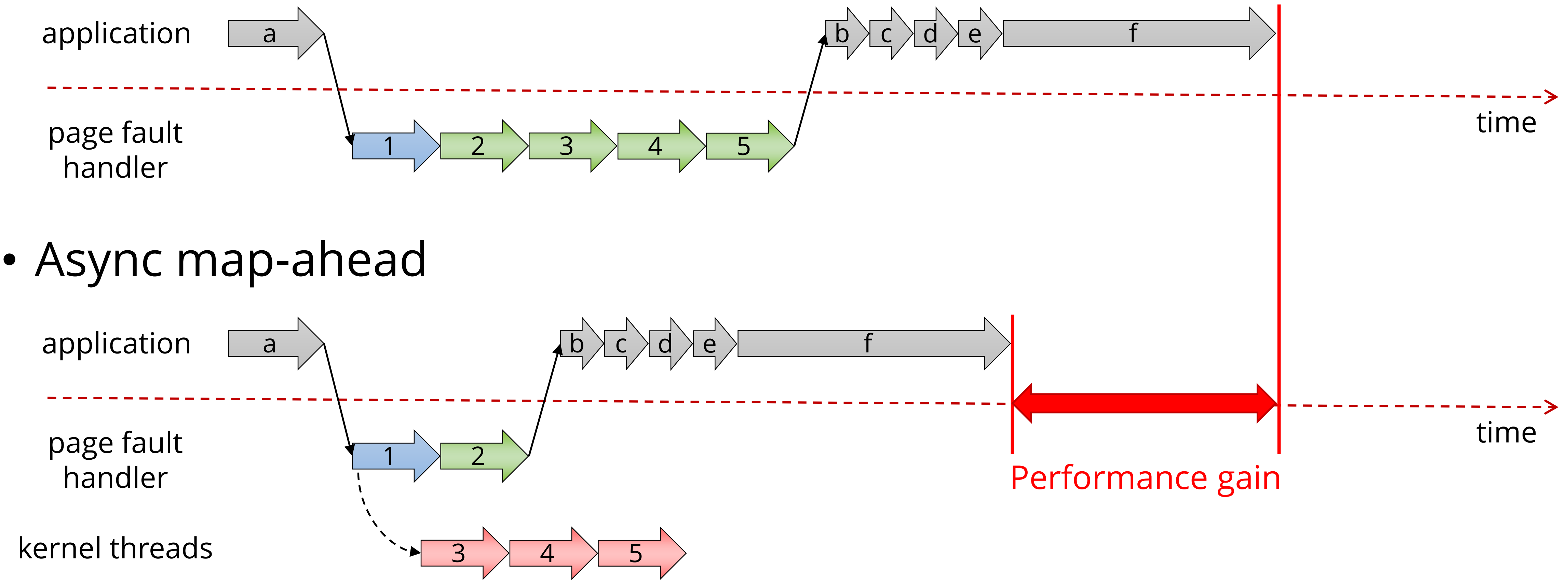
- Sync map-ahead



Async Map-ahead Hides Mapping Overhead

- Sync map-ahead

- Async map-ahead



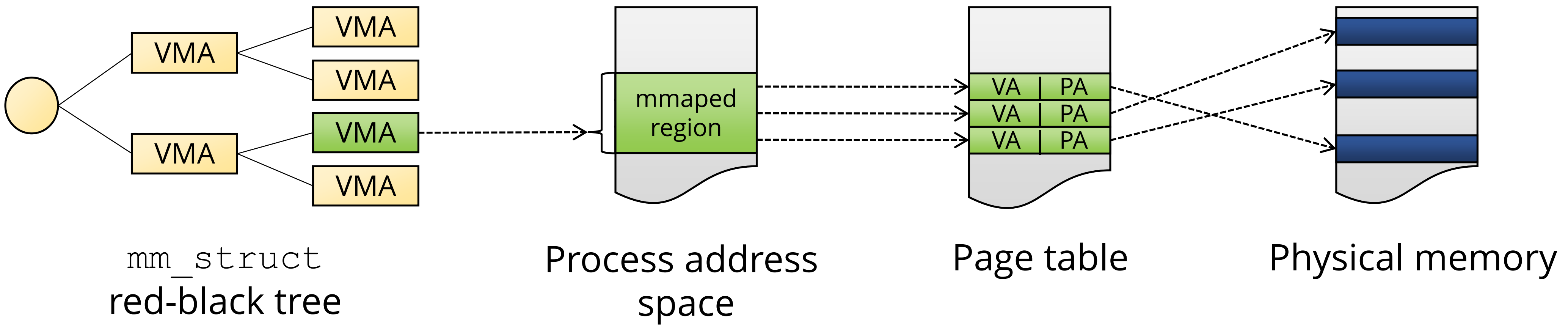
Extended `madvise` Makes User Hints Available

- Taking advantage of user hints can maximize the app's performance
  - However, existing interfaces are to optimize only paging
- Extended `madvise` makes user hints available in memory based FSs

	Existing **`madvise`**	Extended **`madvise`**
`MADV_SEQUENTIAL` `MADV_WILLNEED`	Run readahead aggressively	Run map-ahead aggressively
`MADV_RANDOM`	Stop readahead	Stop map-ahead
`MADV_DONTNEED`	Release pages in page $	Release mapping in mapping $

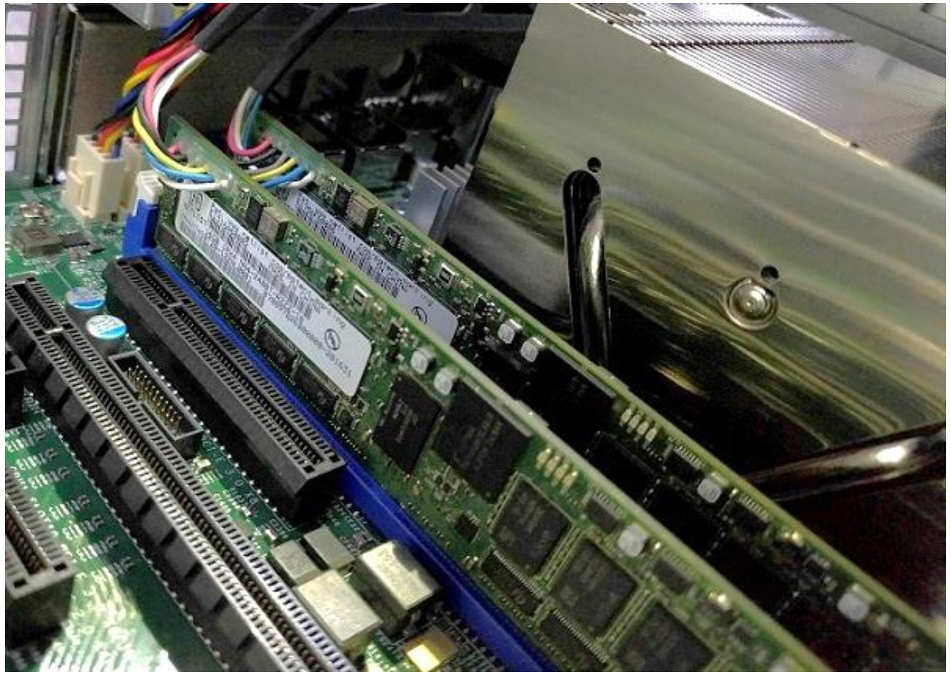
Mapping Cache Reuses Mapping Data

- When `munmap` is called, existing kernel releases the VMA and PTEs
  - In memory based FSs, mapping overhead is very large
- With mapping cache, mapping overhead can be reduced
  - VMAs and PTEs are cached in mapping cache
  - When `mmap` is called, they are reused if possible (cache hit)



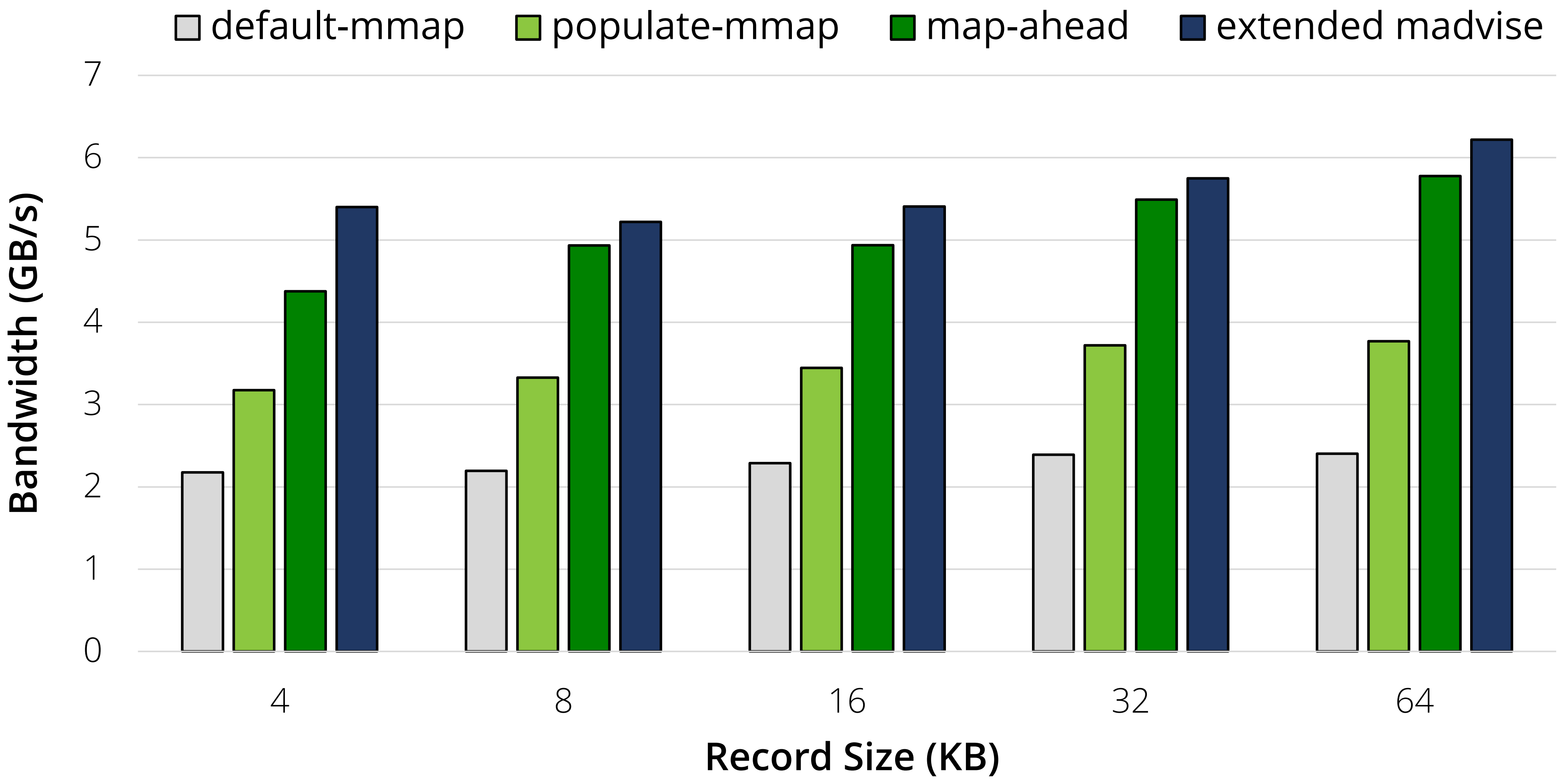
Experimental Environment

- Experimental machine
  - Intel Xeon E5-2620
  - 32GB DRAM, 32GB NVDIMM-N
  - Linux kernel 4.4
  - Ext4-DAX filesystem
- Benchmarks
  - fio : sequential & random read
  - YCSB on MongoDB : load workload
  - Apache HTTP server with httperf

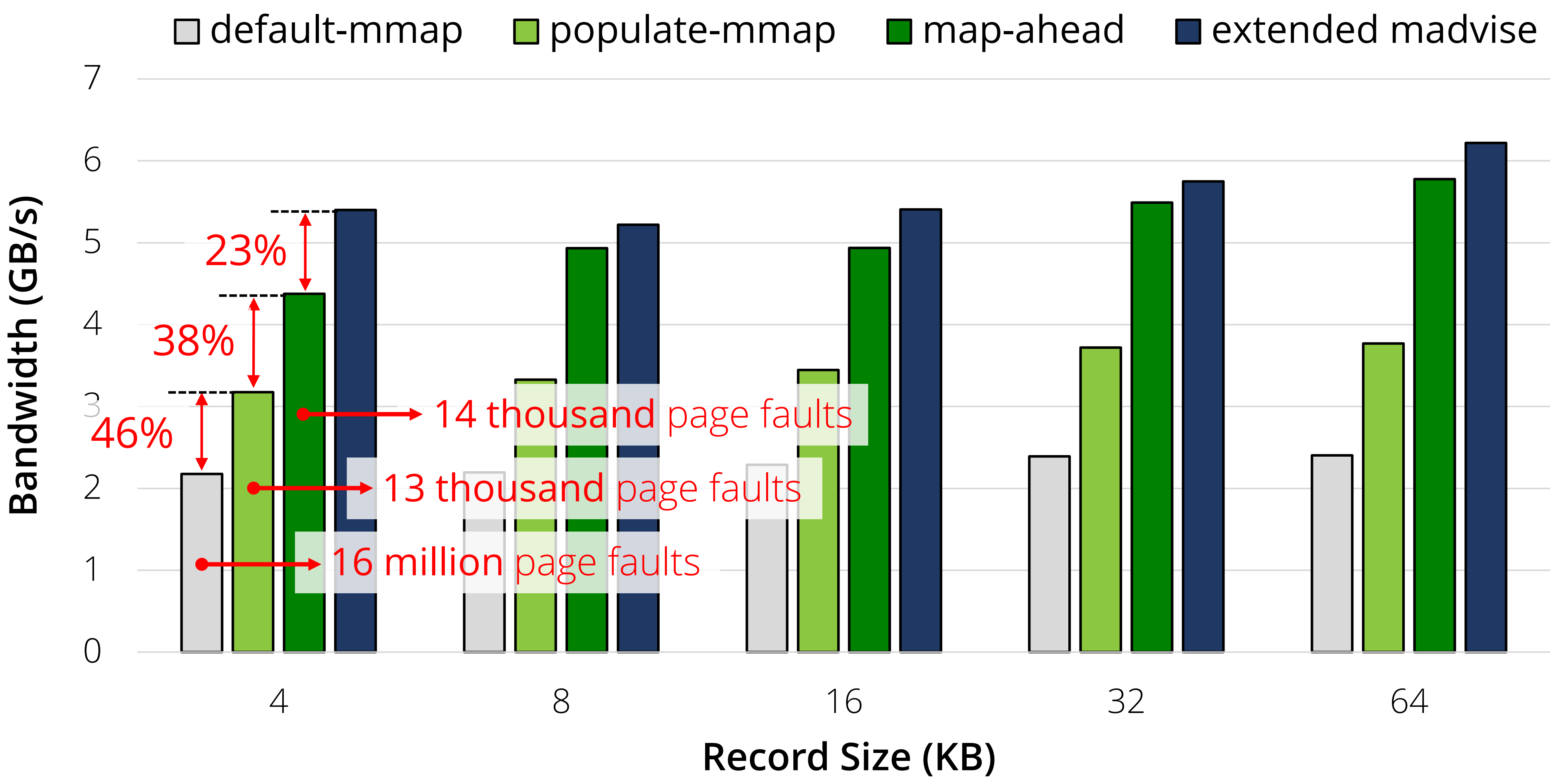


Netlist DDR4 NVDIMM-N 16GB × 2

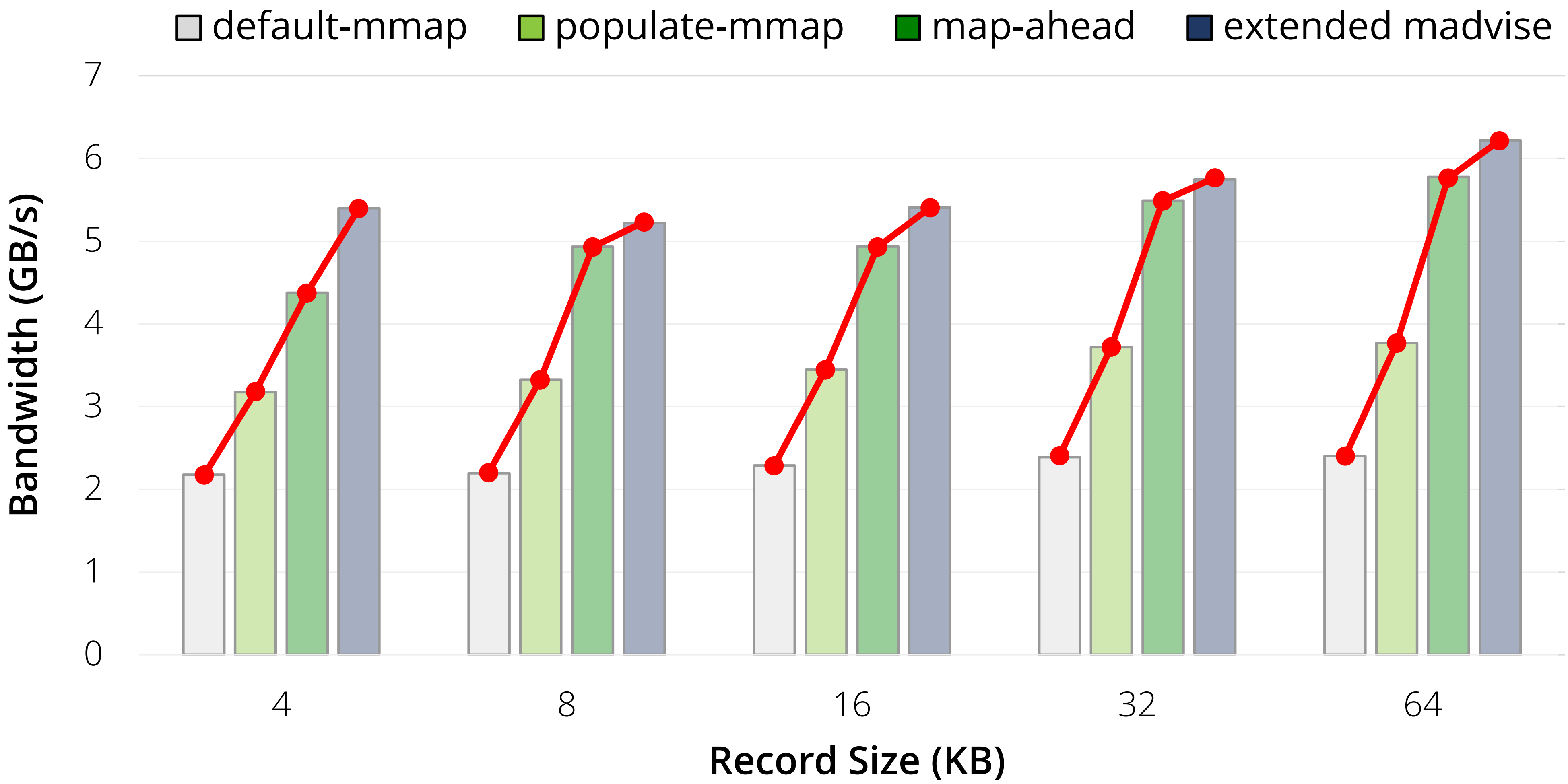
fio : Sequential Read



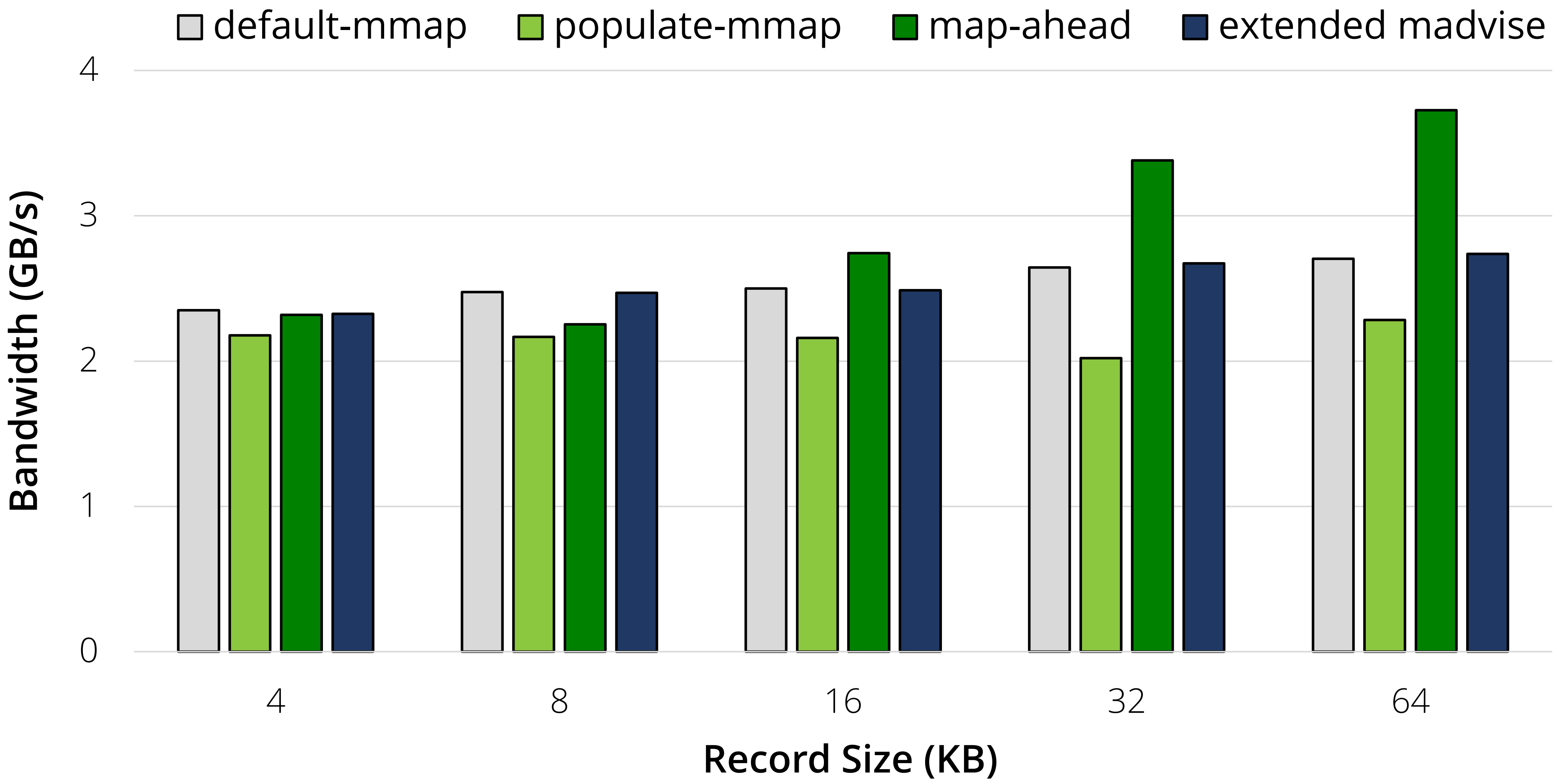
fio : Sequential Read



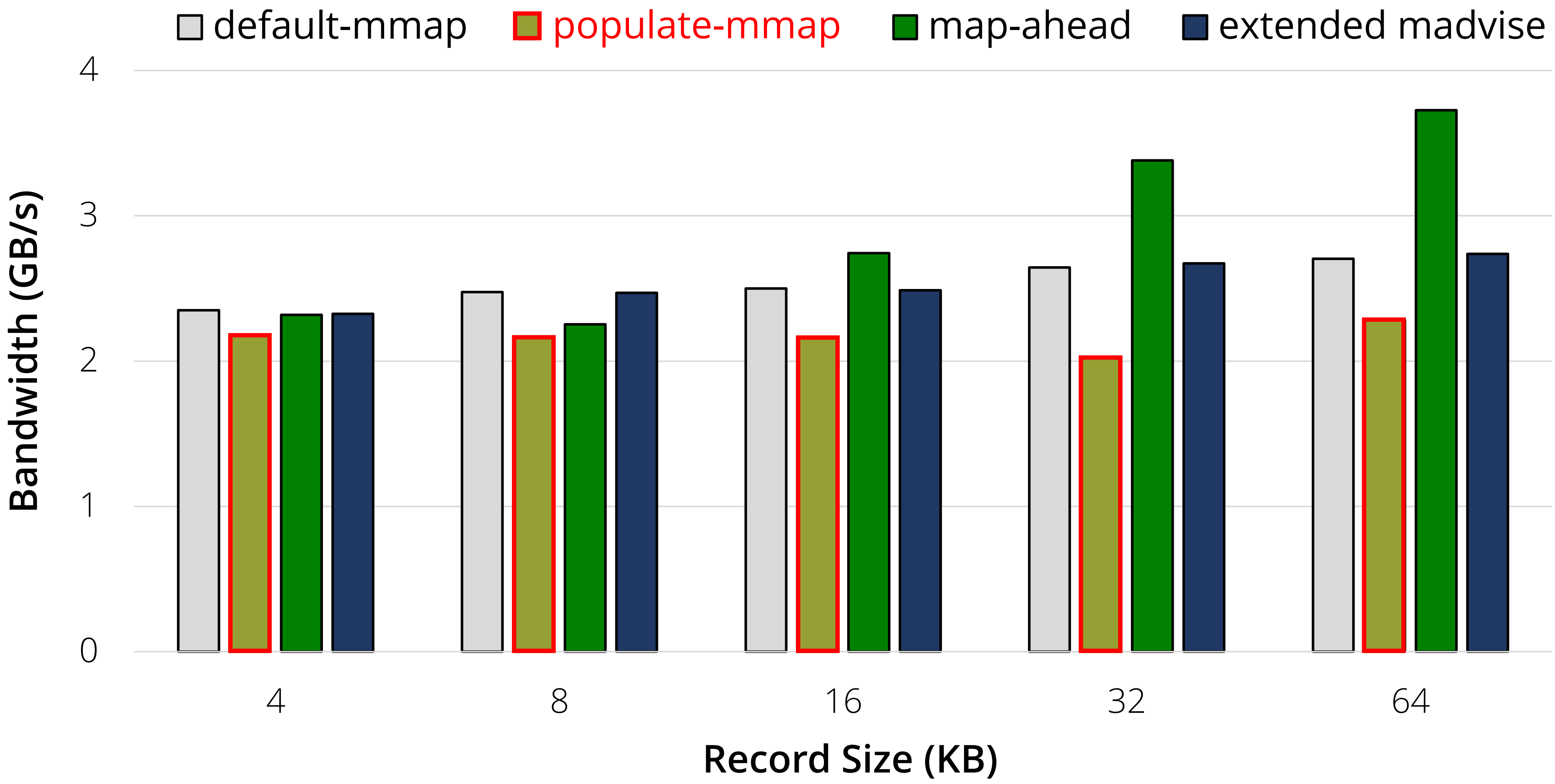
fio : Sequential Read



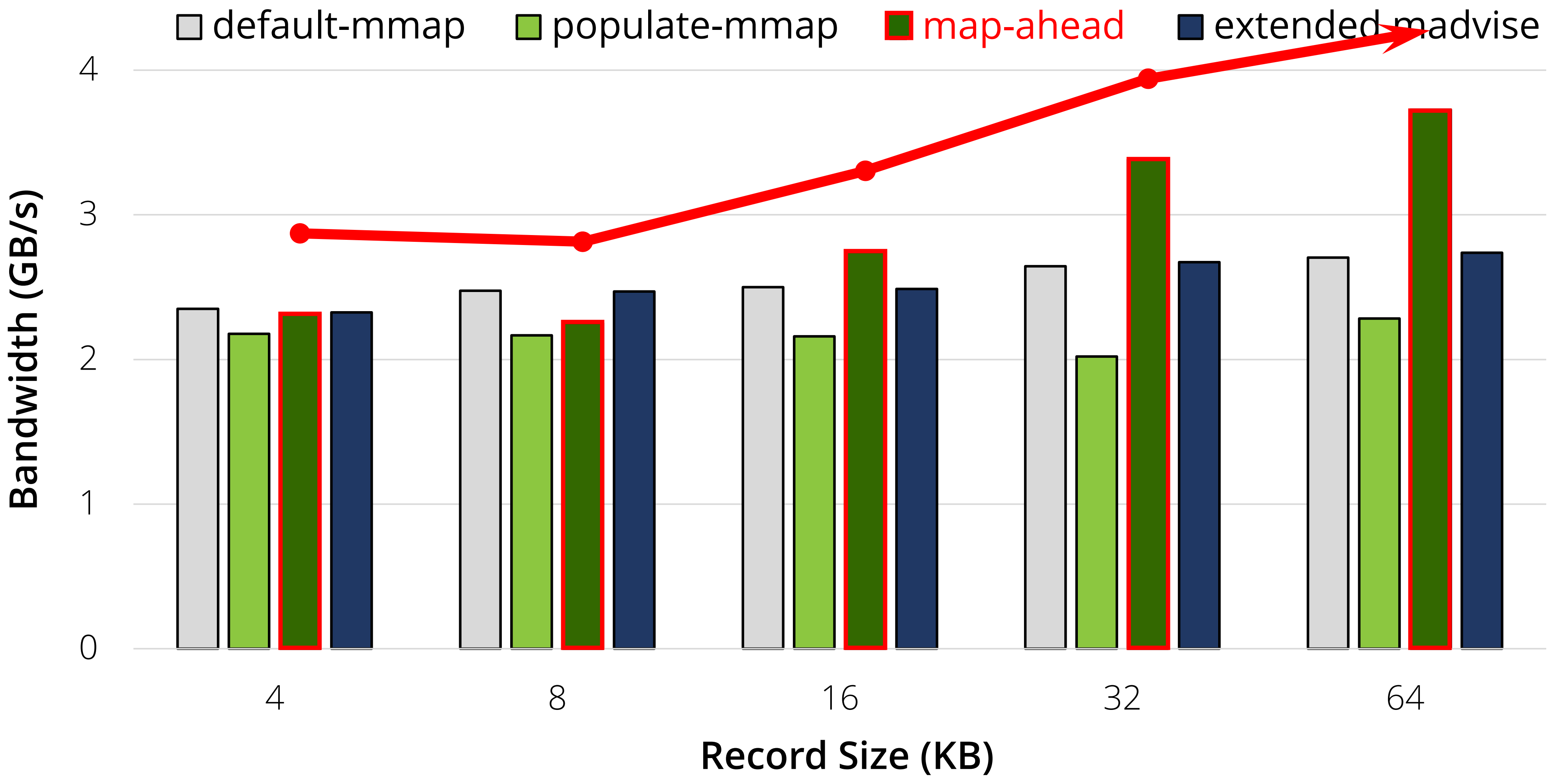
fio : Random Read



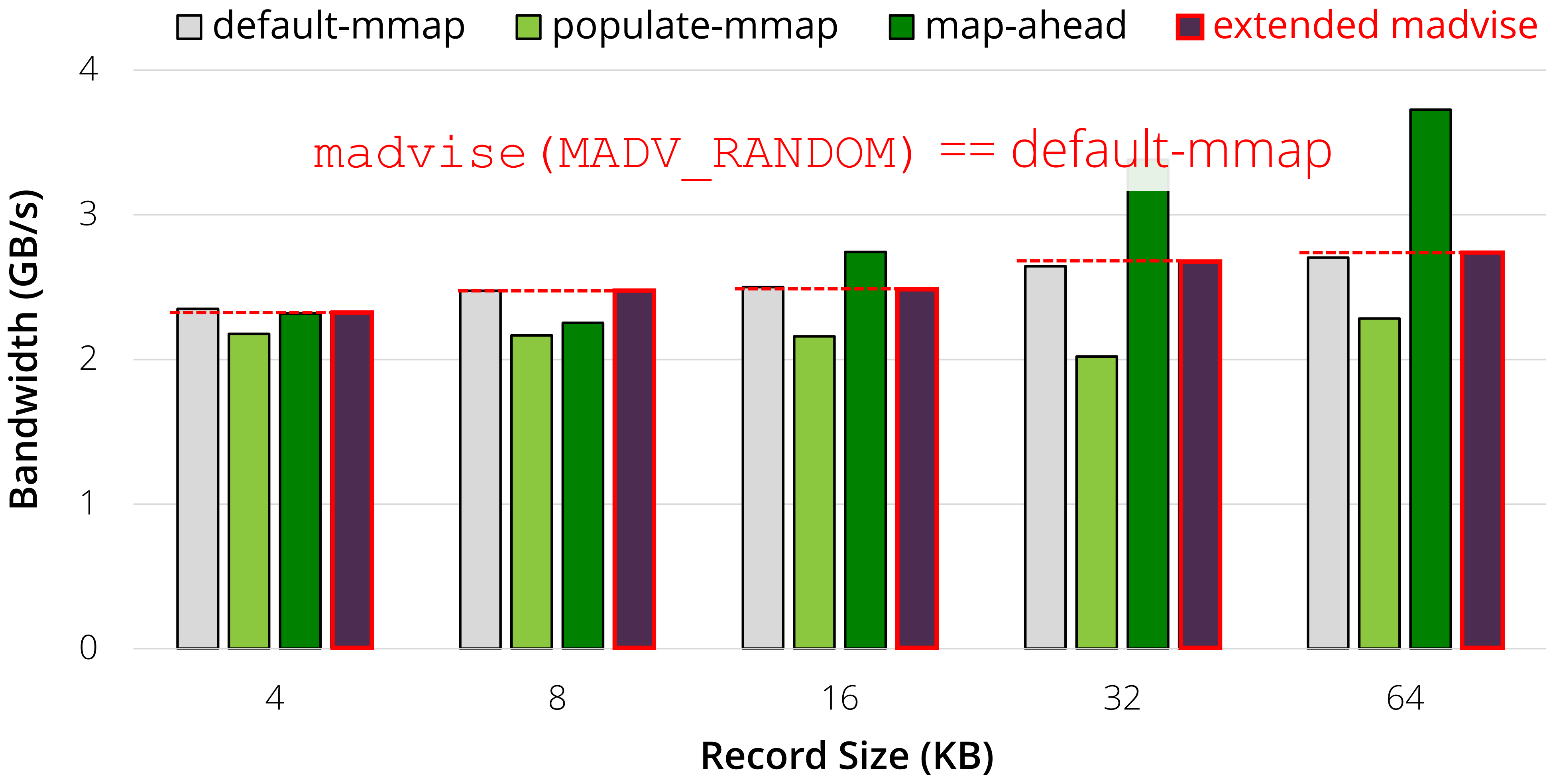
fio : Random Read



fio : Random Read

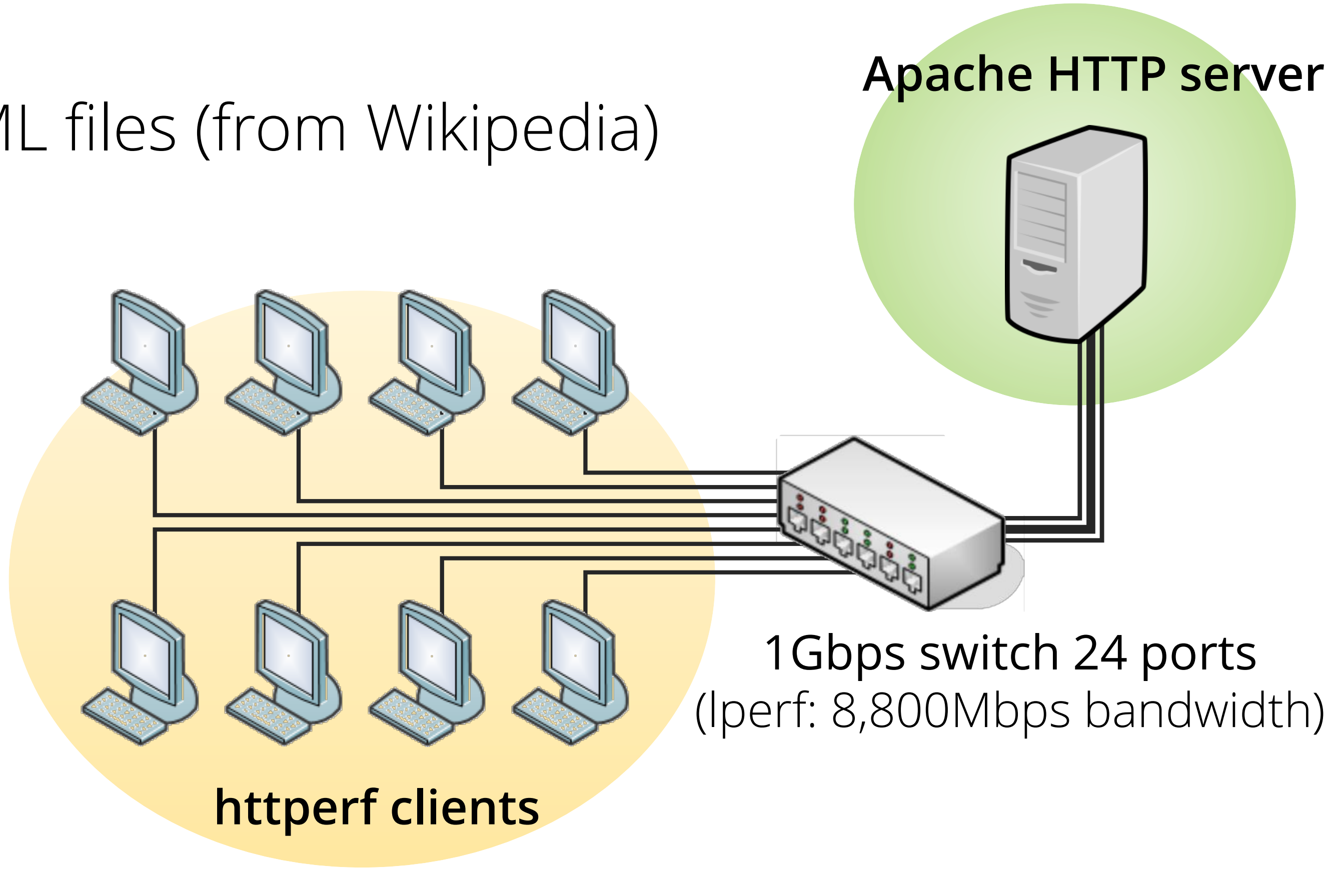


fio : Random Read

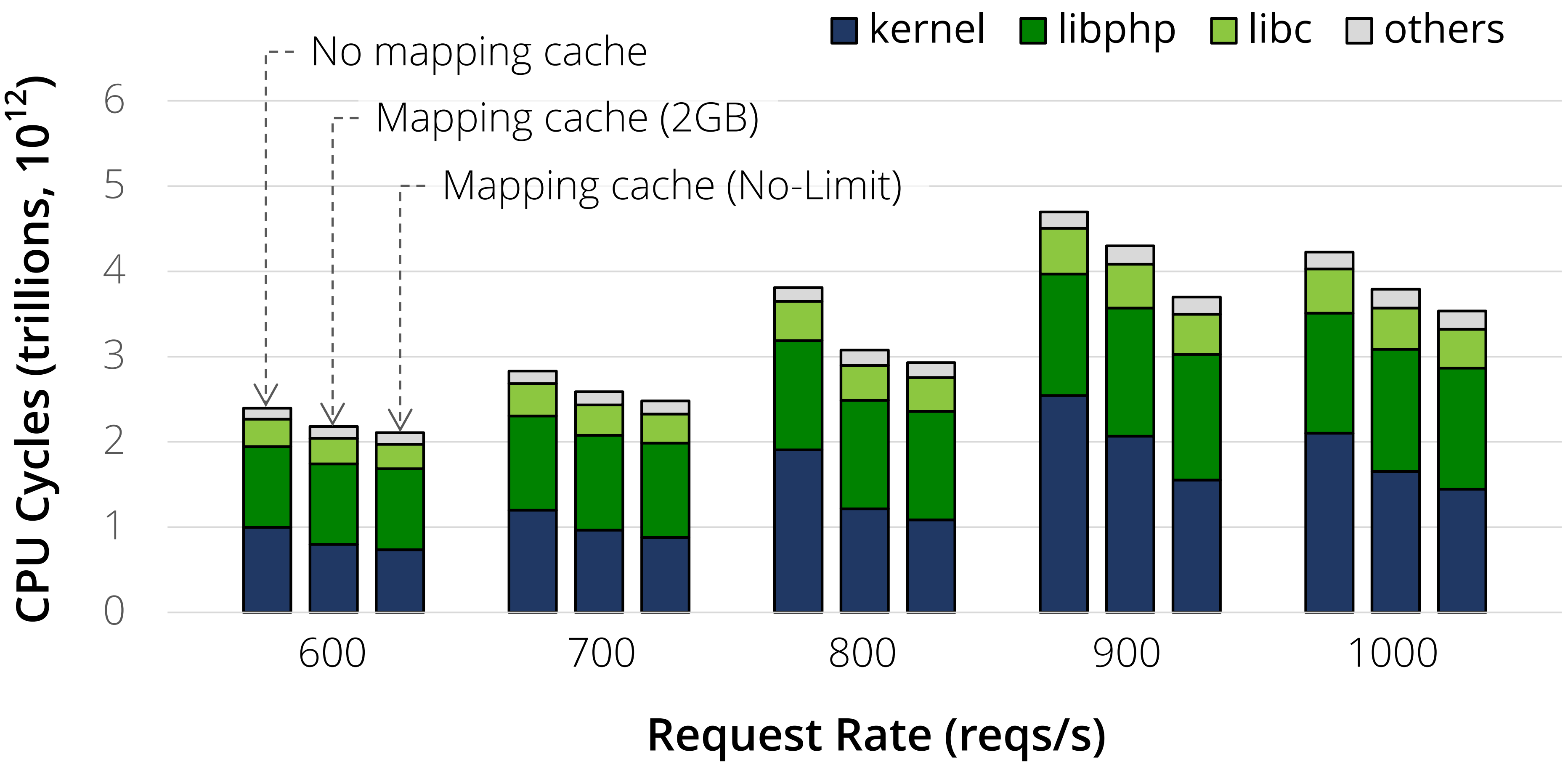


Web Server Experimental Setting

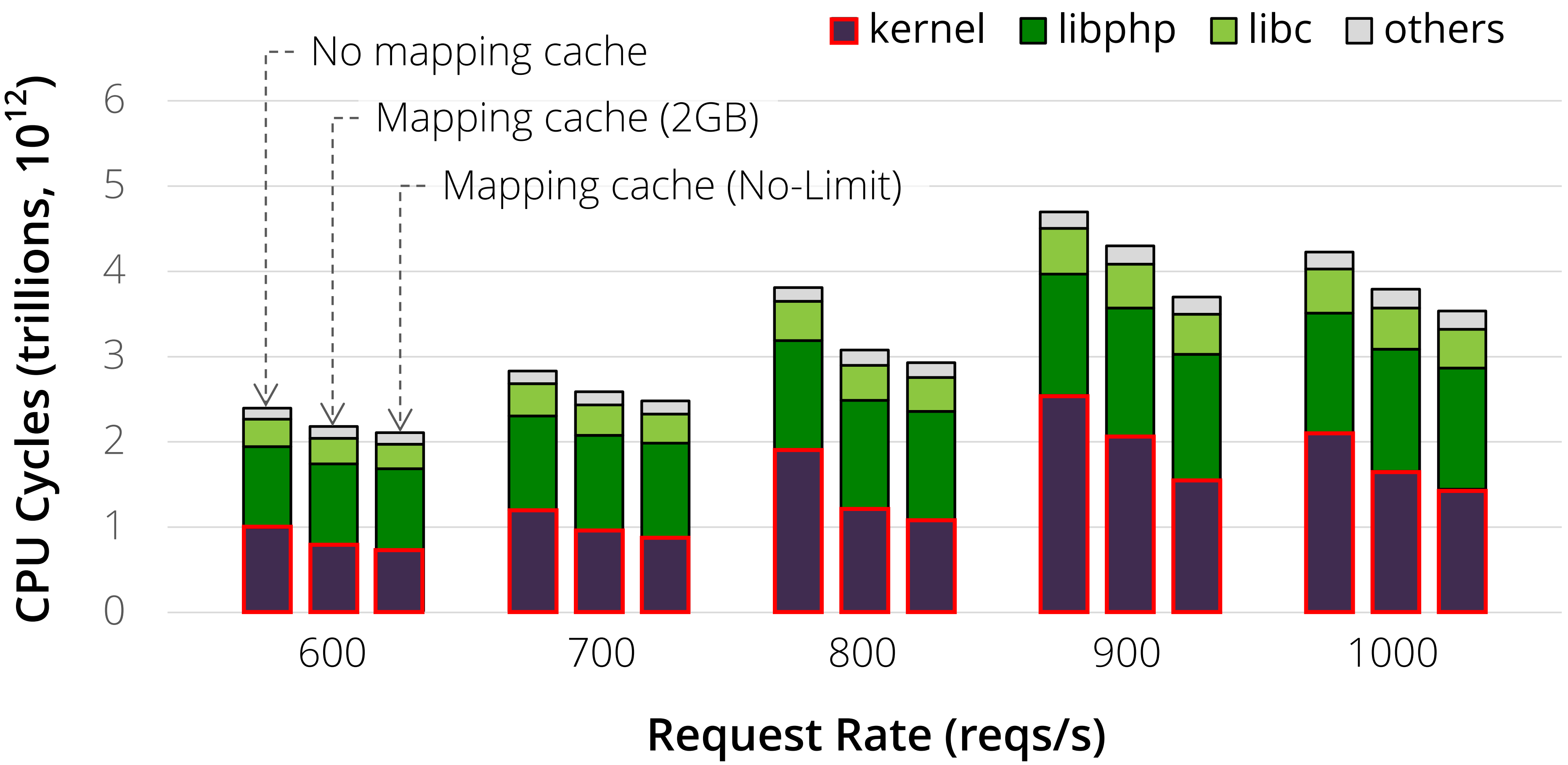
- Apache HTTP server
  - Memory mapped file I/O
  - 10 thousand 1MB-size HTML files (from Wikipedia)
  - Total size is about 10GB
- httperf clients
  - 8 additional machines
  - Zipf-like distribution



Apache HTTP Server



Apache HTTP Server



Conclusion

- SW latency is becoming bigger than the storage latency
  - Memory mapped file I/O can avoid the SW overhead
- Memory mapped file I/O still incurs expensive additional overhead
  - Page fault, TLB miss, and PTEs construction overhead
- To exploit the benefits of memory mapped I/O, we propose
  - Map-ahead, extended madvise, mapping cache
- Our techniques demonstrate good performance by mitigating the mapping overhead
  - Map-ahead : 38% ↑
  - Map-ahead + extended `madvise` : 23% ↑
  - Mapping cache : 18% ↑

QnA

chjs@skku.edu