Optimizing Memory-mapped I/O for Fast Storage Devices

Anastasios Papagiannis[1,2], Giorgos Xanthakis[1,2], Giorgos Saloustros[1], Manolis Marazakis[1], and Angelos Bilas[1,2]

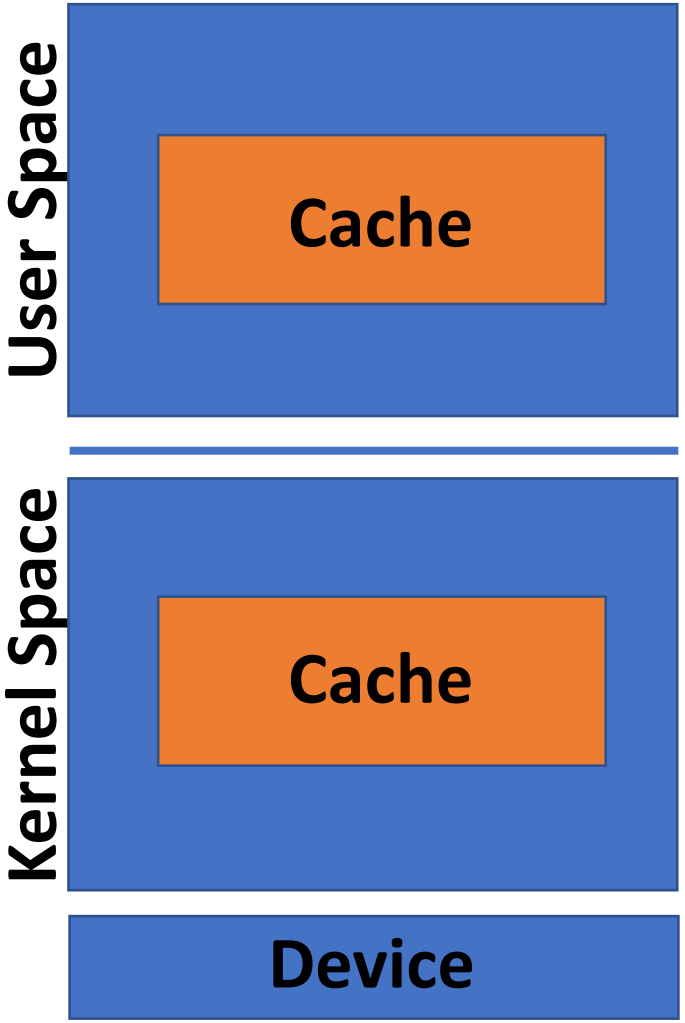
Foundation for Research and Technology – Hellas (FORTH)[1] & University of Crete[2]

Fast storage devices

- Fast storage devices → Flash, NVMe
    - Millions of IOPS
    - < 10 μs access latency
- Small I/Os are not such a big issue as in rotational disks
- Require many outstanding I/Os for peak throughput

Read/write system calls

- Read/write system calls + DRAM cache
  - Reduce accesses to the device
- Kernel-space cache
  - Requires system calls also for hits
  - Used for raw (serialized) blocks
- User-space cache
  - Lookups for hits + system calls only for misses
  - Application specific (deserialized) data
  - User-space cache removes system calls for hits
- Hit lookups in user space introduce significant overhead [SIGMOD'08]



Memory-mapped I/O

- In memory-mapped I/O (mmio) hits handled in hardware → MMU + TLB
  - Less overhead compared to cache lookup
- In mmio a file mapped to virtual address space
  - Load/store processor instructions to access data
  - Kernel fetch/evict page on-demand
- Additionally mmio removes
  - Serialization/deserialization
  - Memory copies between user and kernel

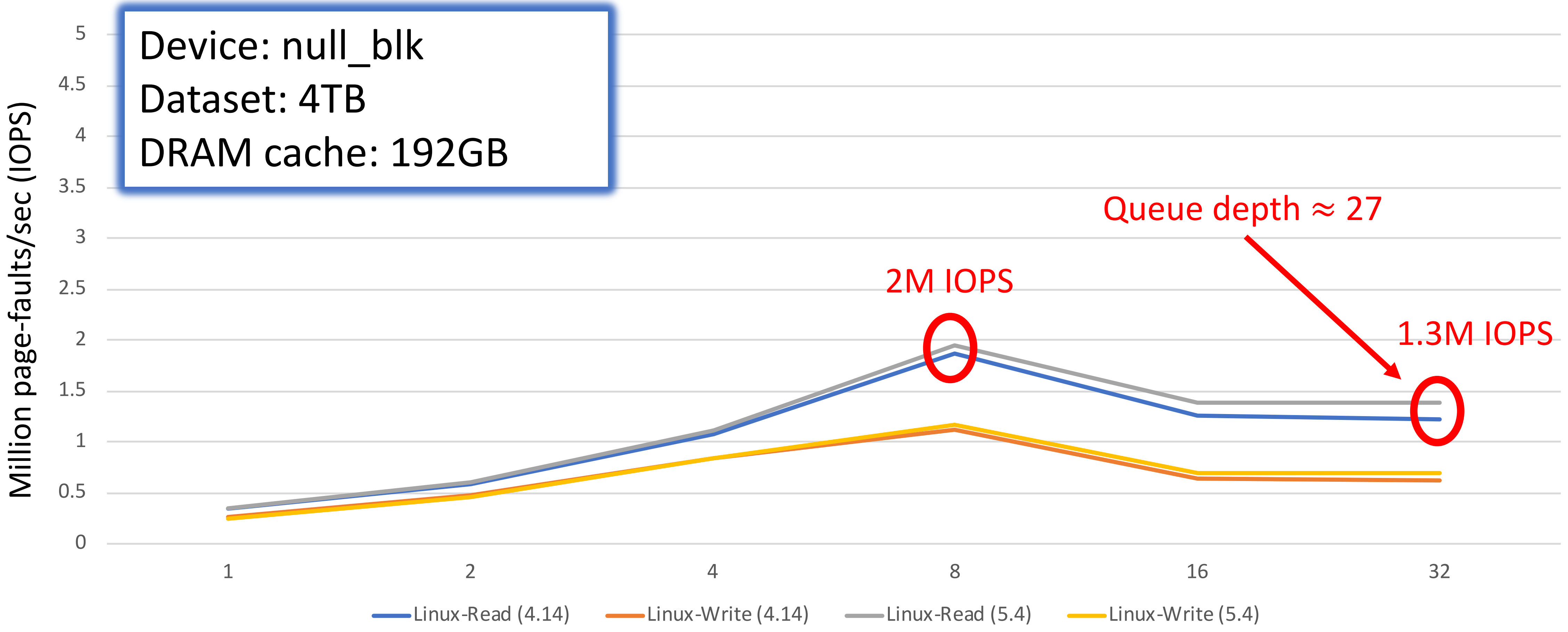
Disadvantages of mmio

- Misses require a page fault instead of a system call
- 4KB page size → Small & random I/Os
  - With fast storage devices this is not a big issue
- Linux mmio path fails to scale with #threads

Mmio path scalability

Device: null_blk
Dataset: 4TB
DRAM cache: 192GB

Mmio path scalability



Device: null_blk
Dataset: 4TB
DRAM cache: 192GB

2M IOPS

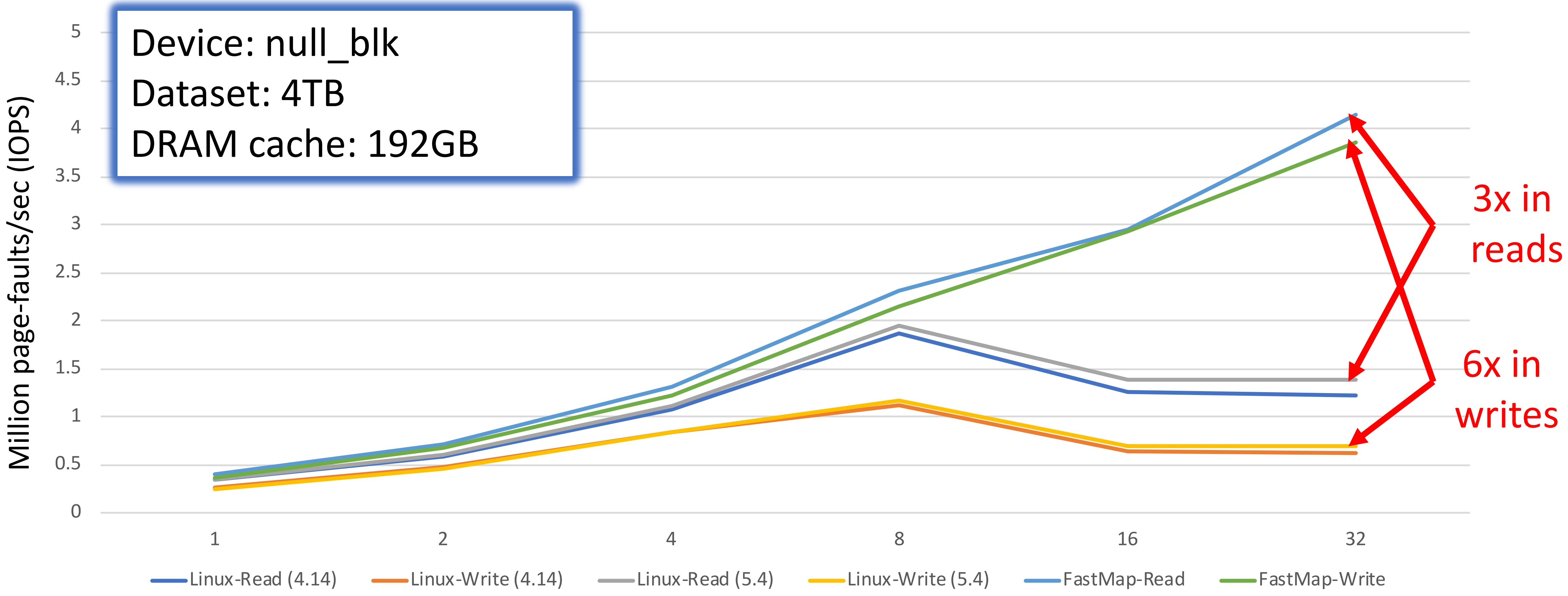
Queue depth ≈ 27

1.3M IOPS

FastMap

- A novel mmio path that achieves high scalability and I/O concurrency
  - In the Linux kernel
- Avoids all centralized contention points
- Reduces CPU processing in the common path
- Uses dedicated data structures to minimize interference

Mmio path scalability


Device: null_blk
Dataset: 4TB
DRAM cache: 192GB

3x in reads

6x in writes

Outline

- Introduction
- Motivation
- FastMap design
- Experimental analysis
- Conclusions

Outline

- Introduction
- Motivation
- FastMap design
- Experimental analysis
- Conclusions

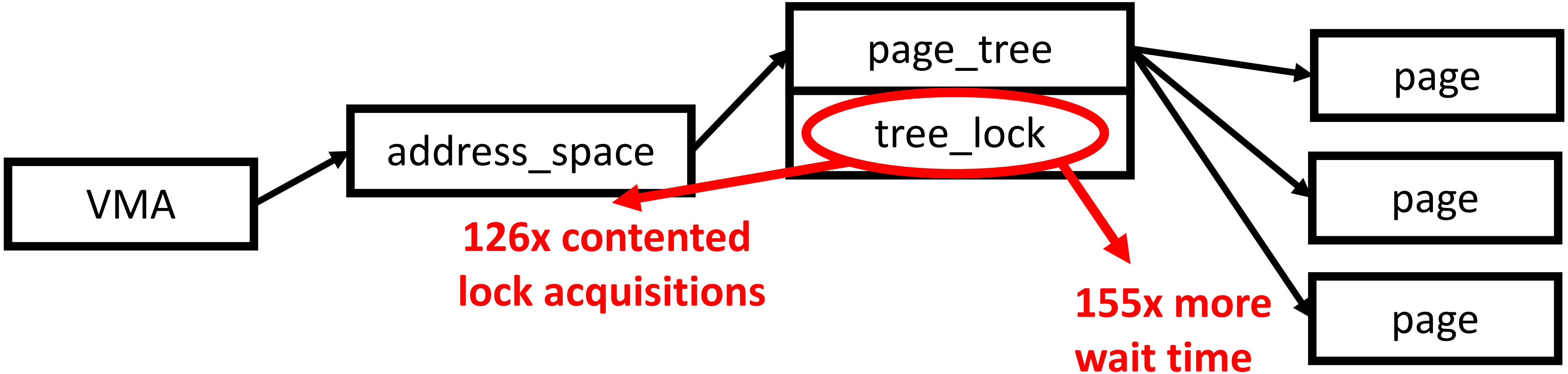
FastMap design: 3 main techniques

- Separates data structures that keep clean and dirty pages
    - Avoids all centralized contention points
- Optimizes reverse mappings
    - Reduces CPU processing in the common path
- Uses a scalable DRAM cache
    - Minimizes interference and reduce latency variability

FastMap design: 3 main techniques

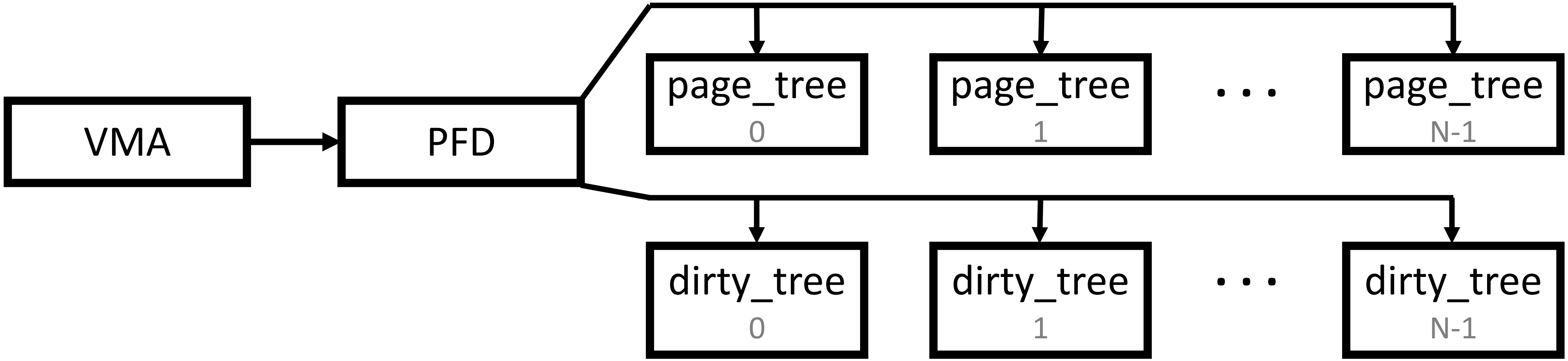
- Separates data structures that keep clean and dirty pages
  - Avoids all centralized contention points
- Optimizes reverse mappings
  - Reduces CPU processing in the common path
- Uses a scalable DRAM cache
  - Minimizes interference and reduce latency variability

Linux mmio design



- tree_lock acquired for 2 main reasons
  - Insert/remove elements from page_tree & lock-free (RCU) lookups
  - Modify **tags** for a specific entry → Used to mark a page dirty

FastMap design



- Keep dirty pages on a separate data structure
- Marking a page dirty/clean does not serialize insert/remove ops
- Choose data-structure based on page_offset % num_cpus
- Radix trees to keep **ALL** cached pages → lock-free (RCU) lookups
- Red-black trees to keep **ONLY** dirty pages → sorted by device offset

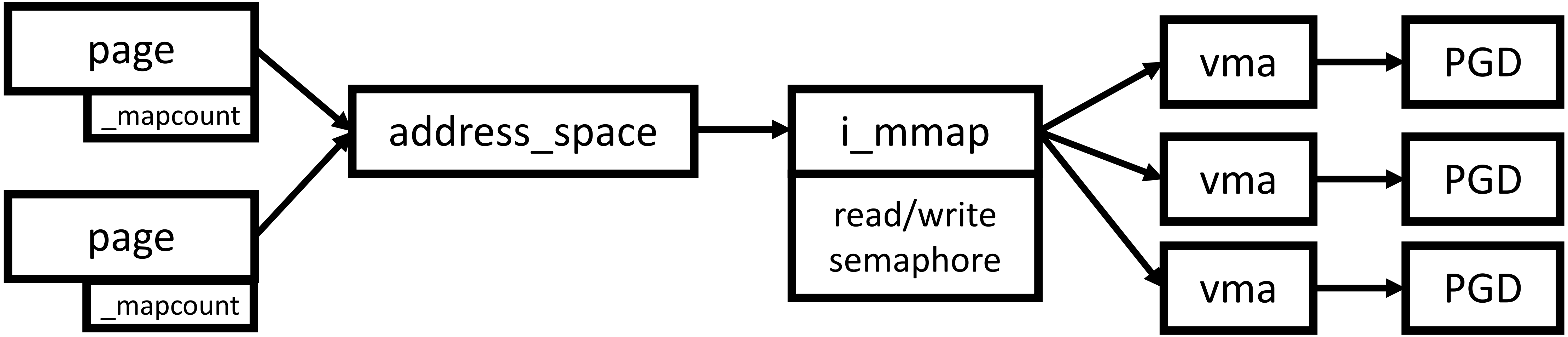
FastMap design: 3 main techniques

- Separates data structures that keep clean and dirty pages
  - Avoids all centralized contention points
- Optimizes reverse mappings
  - Reduces CPU processing in the common path
- Uses a scalable DRAM cache
  - Minimizes interference and reduce latency variability

Reverse mappings

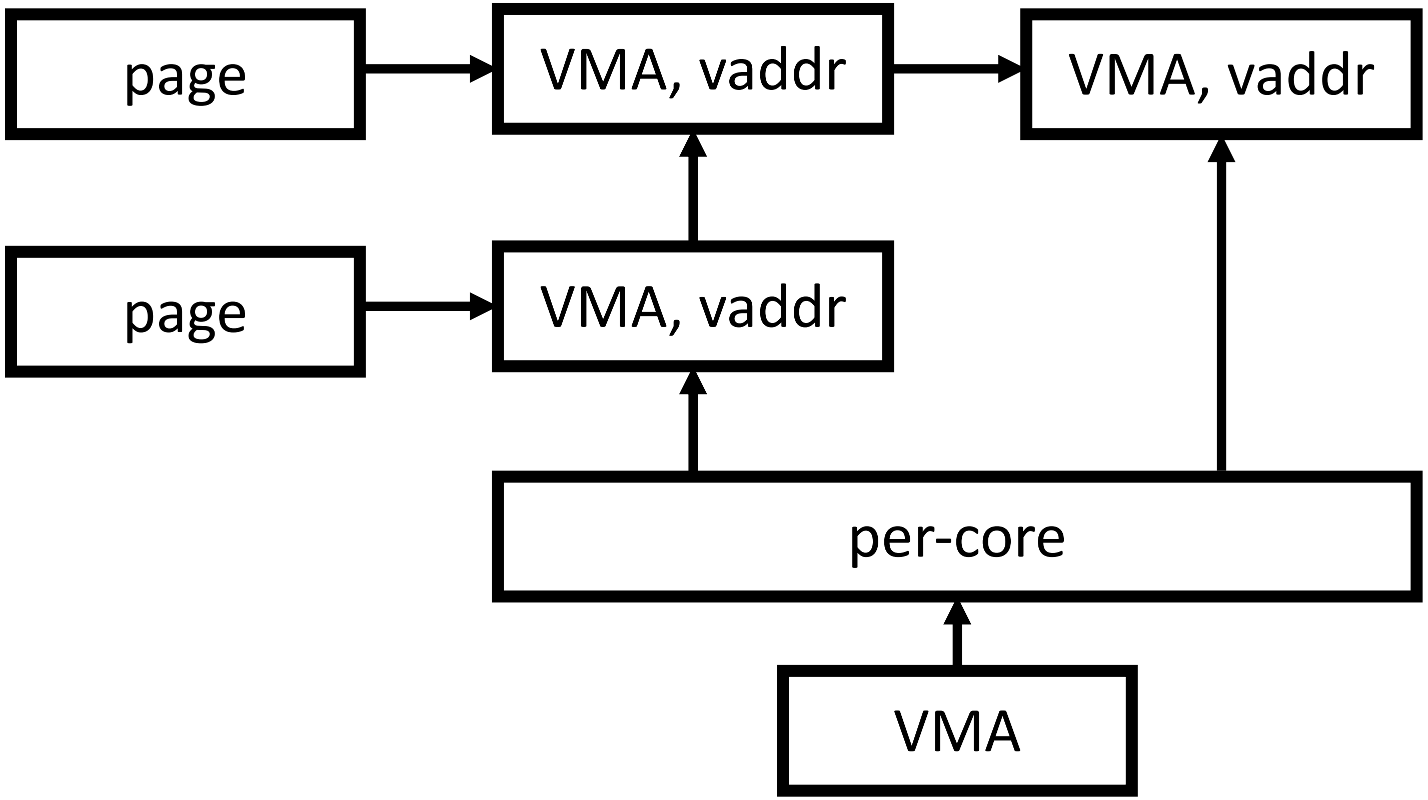
- Find out which page table entries map a specific page
  - Page eviction → Due to memory pressure or explicit writeback
  - Destroy mappings → munmap
- Linux uses object-based reverse mappings
  - Executables and libraries (e.g. libc) introduce large amount of sharing
  - Reduces DRAM consumption and housekeeping costs
- Storage applications that use memory-mapped I/O
  - Require minimal sharing
  - Can be applied selectively to certain devices or files

Linux object-based reverse mappings



- _mapcount can still results in useless page table traversals
- rw-semaphore acquired as read on all operations
  - Cross NUMA-node traffic
  - Spend many CPU cycles

FastMap full reverse mappings



- Full reverse mappings
  - Reduce CPU overhead
- Efficient munmap
  - No ordering required ➔ scalable updates
- More DRAM required
  - Limited by small degree of sharing in pages

FastMap design: 3 main techniques

- Separates data structures that keep clean and dirty pages
  - Avoids all centralized contention points
- Optimizes reverse mappings
  - Reduces CPU processing in the common path
- Uses a scalable DRAM cache
  - Minimizes interference and reduce latency variability

Batched TLB invalidations

- Under memory pressure FastMap evicts a batch of clean pages
  - Cache related operations
  - Page table cleanup
  - TLB invalidation
- TLB invalidation require an IPI (Inter-Processor Interrupt)
  - Limits scalability [EuroSys’13, USENIX ATC’17, EurorSys’20]
- Single TLB invalidation for the whole batch
  - Convert batch to range including unnecessary invalidations

Other optimizations in the paper

- DRAM cache
- Eviction/writeback operations
- Implementation details

Outline

- Introduction
- Motivation
- FastMap design
- Experimental analysis
- Conclusions

Testbed

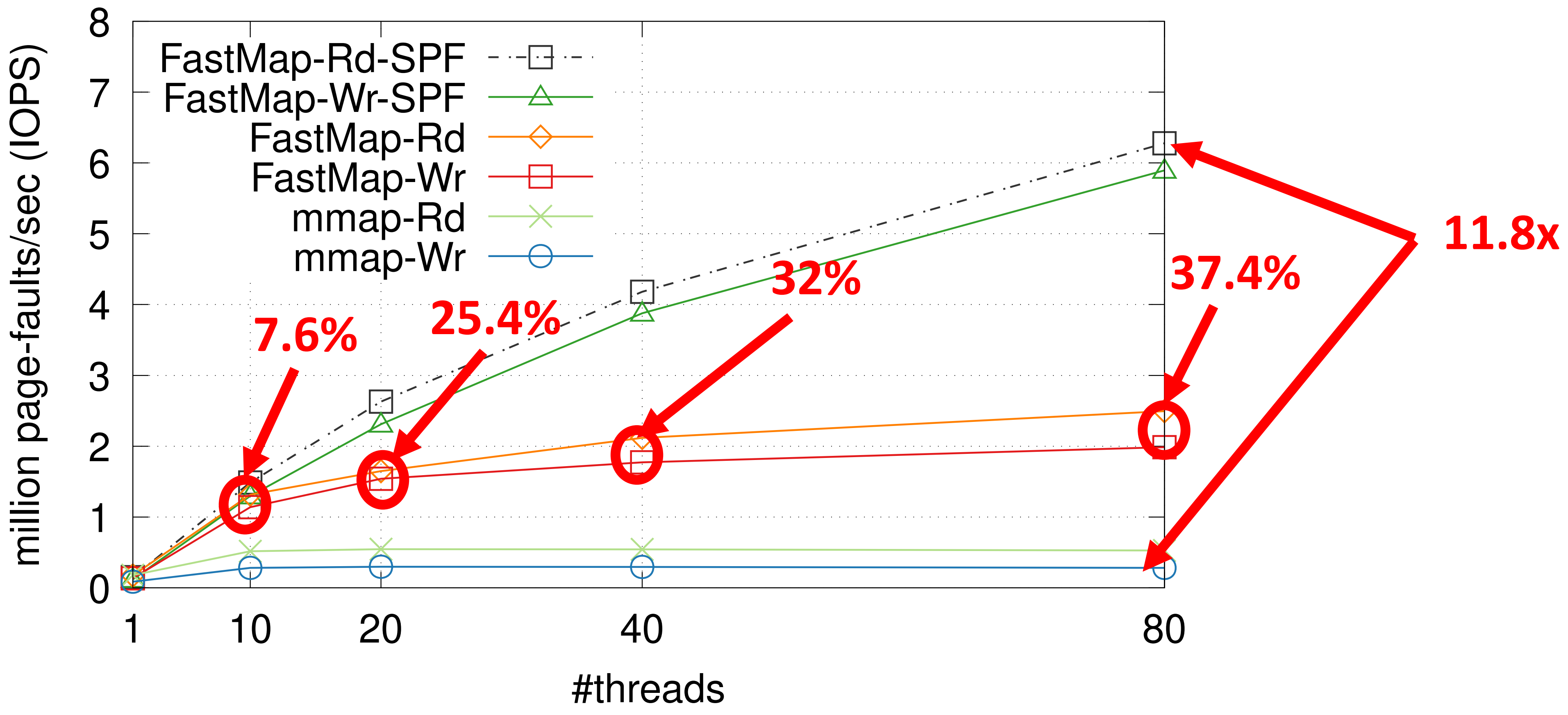
- 2x Intel Xeon CPU E5-2630 v3 CPUs (2.4GHz)
  - 32 hyper-threads
- Different devices
  - Intel Optane SSD DC P4800X (375GB) in workloads
  - null_blk in microbenchmarks
- 256 GB of DDR4 DRAM
- CentOS v7.3 with Linux 4.14.72

Workloads

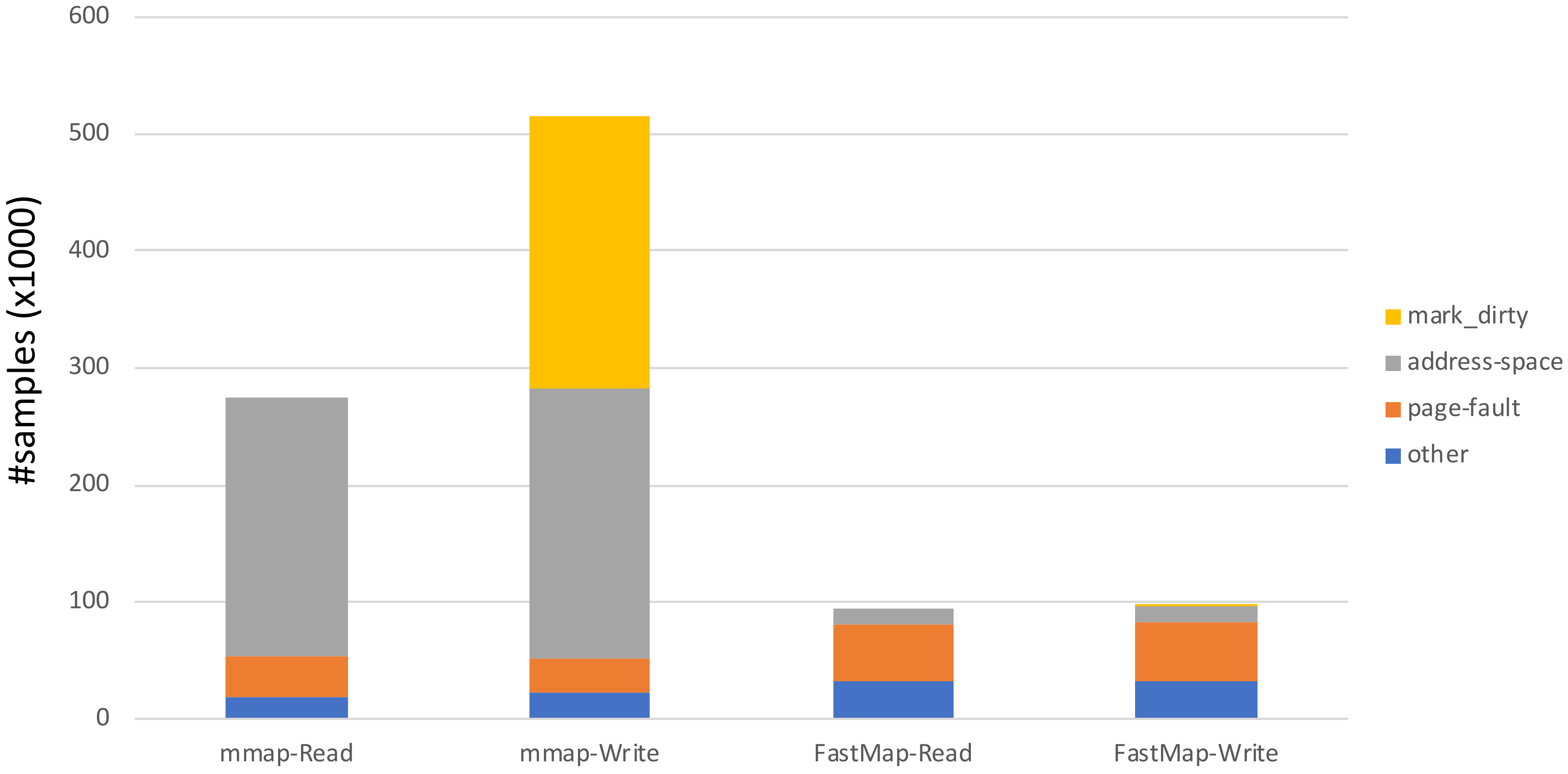
- Microbenchmarks
- Storage applications
  - Kreon [ACM SoCC’18] – persistent key-value store (YCSB)
  - MonetDB – column oriented DBMS (TPC-H)
- Extend available DRAM over fast storage devices
  - Silo [SOSP’13] – key-value store with scalable transactions (TPC-C)
  - Ligra [PPoPP’13] – graph algorithms (BFS)

FastMap Scalability

4x Intel Xeon CPU E5-4610 v3 CPUs (1.7 GHz)
80 hyper-threads



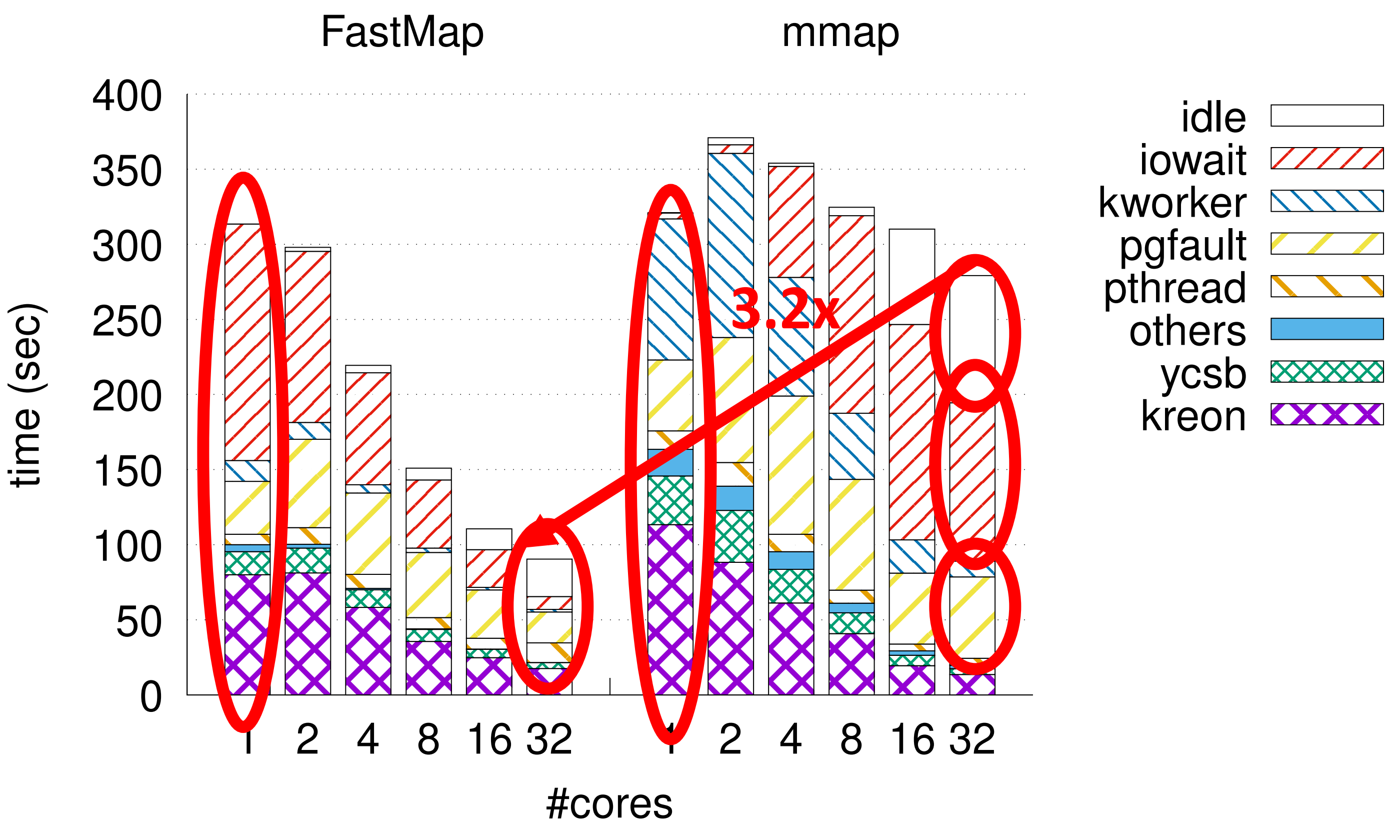
FastMap execution time breakdown



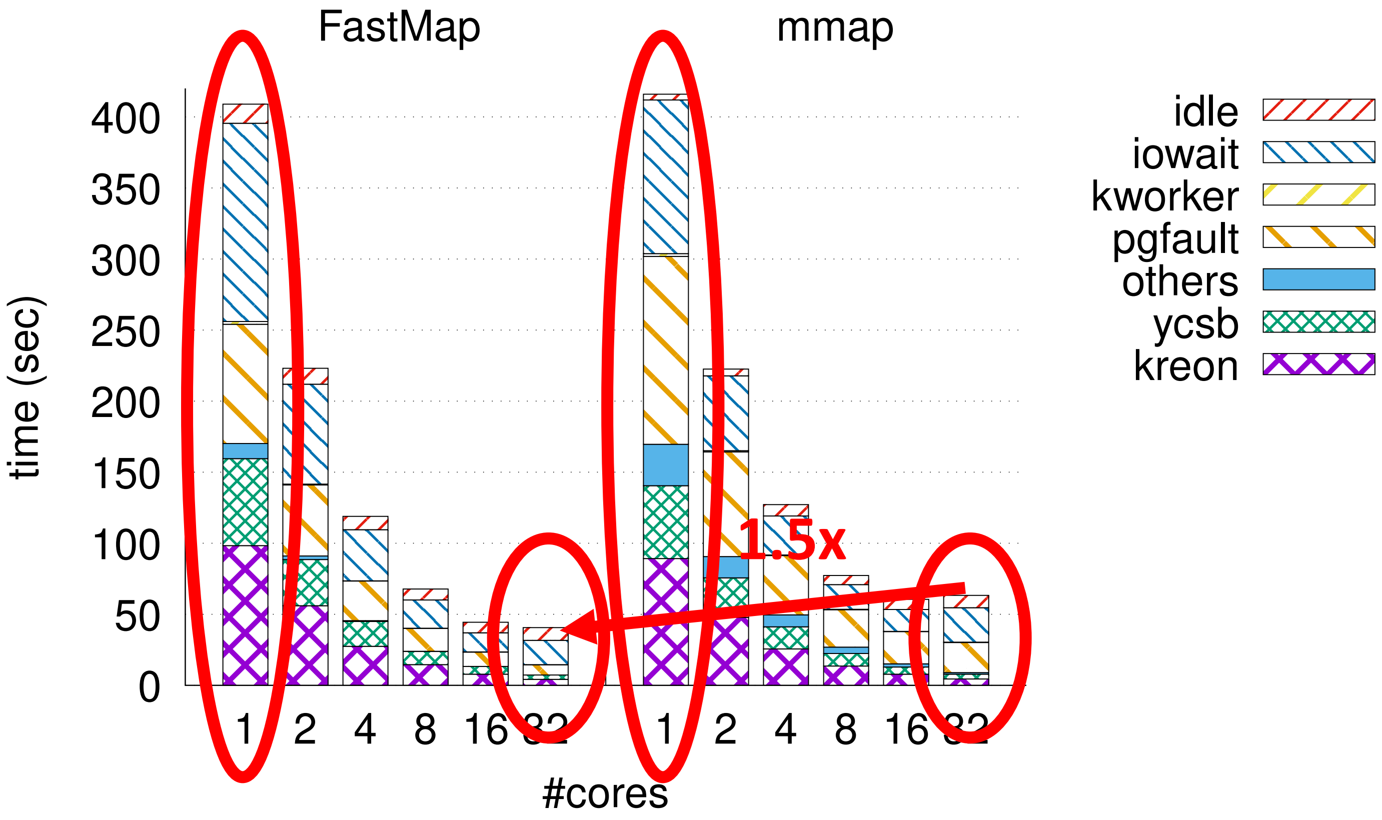
Kreon key-value store

- Persistent key-value store based on LSM-tree
- Designed to use memory-mapped I/O in the common path
- YCSB with 80M records
  - 80GB dataset
  - 16GB DRAM

Kreon – 100% inserts



Kreon – 100% lookups



Batched TLB invalidations

- TLB batching results in 25.5% more TLB misses
- Improvement due to fewer IPIs
  - 24% higher throughput
  - 23.8% lower average latency
- Less time in flush_tlb_mm_range()
  - 20.3% → 0.1%

Silo key-value store
&
TPC-C

Conclusions

- FastMap, an optimized mmio path in Linux
    - Scalable with number of threads & low CPU overhead
- FastMap has significant benefits for data-intensive applications
    - Fast storage devices
    - Multi-core servers
- Up to 11.8x more IOPS with 80 cores and null_blk
- Up to 5.2x more IOPS with 32 cores and Intel Optane SSD

Optimizing Memory-mapped I/O for Fast Storage Devices

Anastasios Papagiannis

Foundation for Research and Technology Hellas (FORTH) & University of Crete

email: apapag@ics.forth.gr