OSDI '24

Identifying On-/Off-CPU Bottlenecks Together with Blocked Samples

Minwoo Ahn[1], Jeongmin Han[1], Youngjin Kwon[2] and Jinkyu Jeong[3]

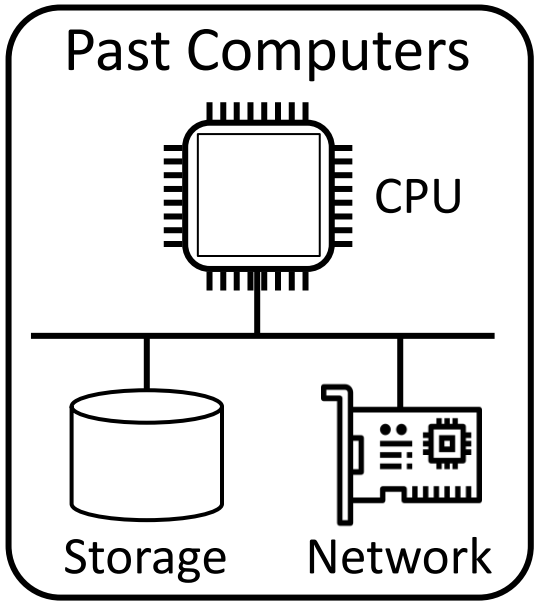
[1] Sungkyunkwan University

[2] Korea Advanced Institute of Science and Technology (KAIST)

[3] Yonsei University

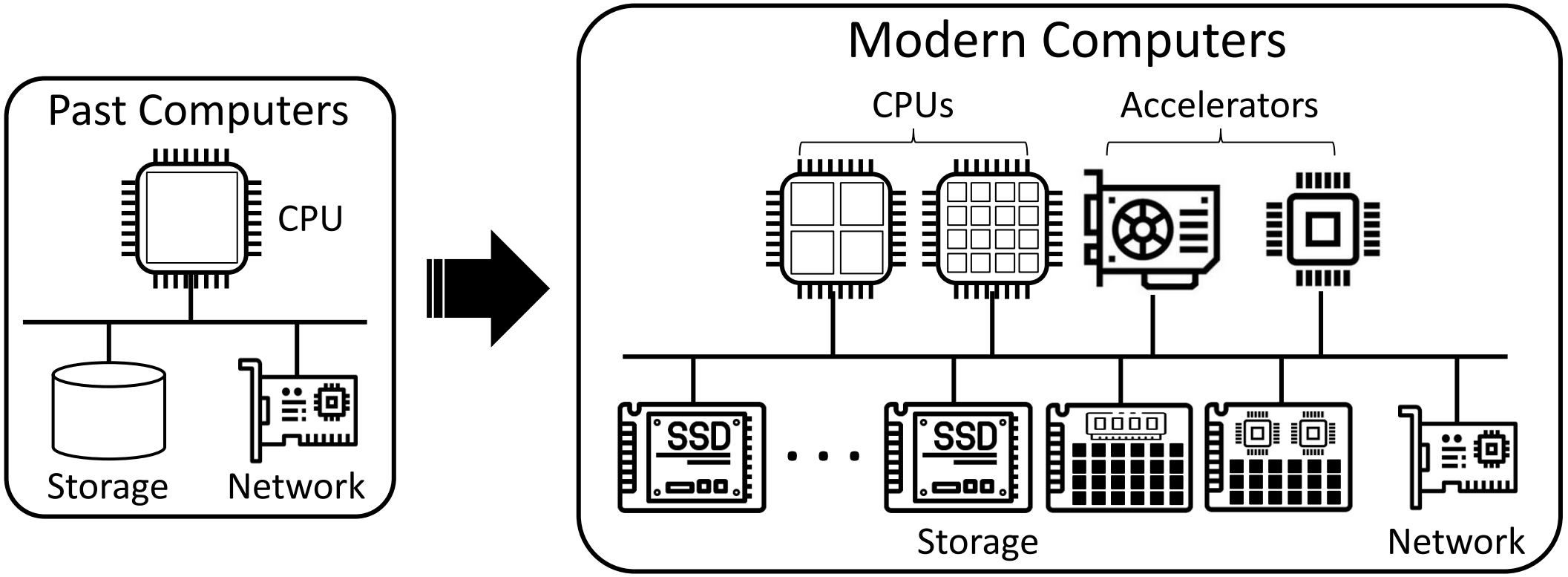
Trend of Computing Environments

- Computing environments are becoming more complex and advanced
  - Events executed outside the CPU (i.e., off-CPU) have become more diverse



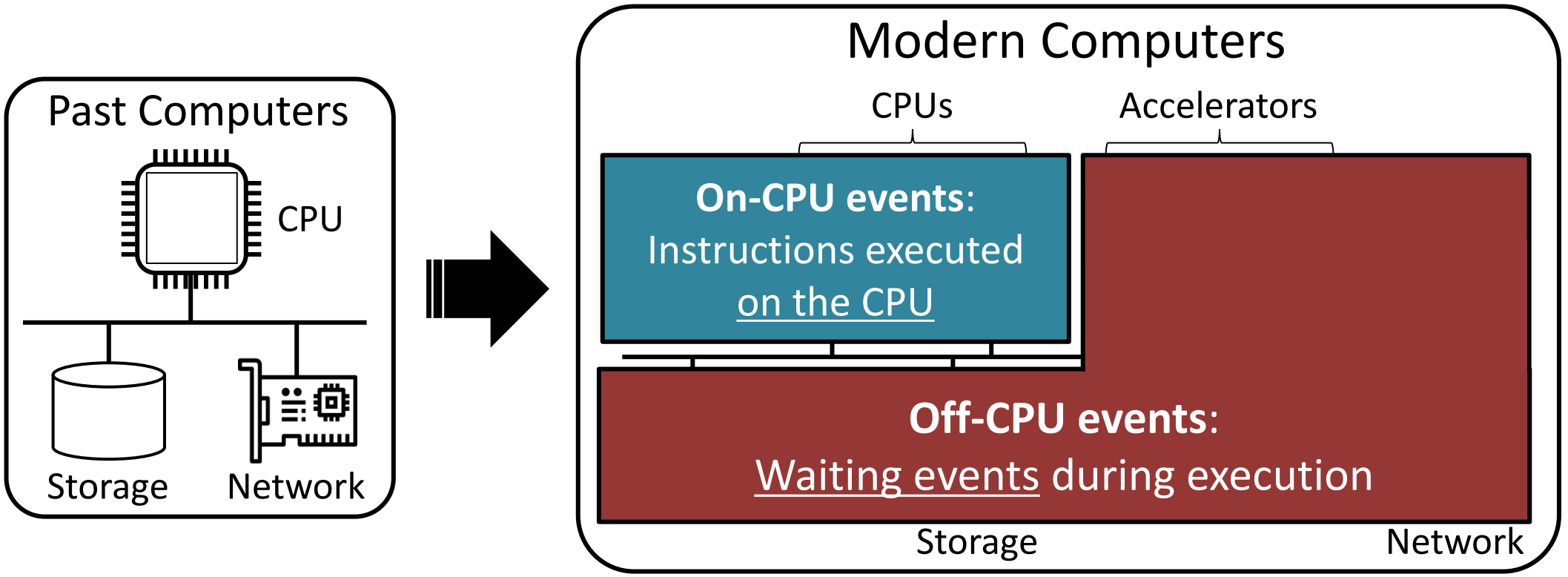
Trend of Computing Environments

- Computing environments are becoming more complex and advanced
  - Events executed outside the CPU (i.e., off-CPU) have become more diverse



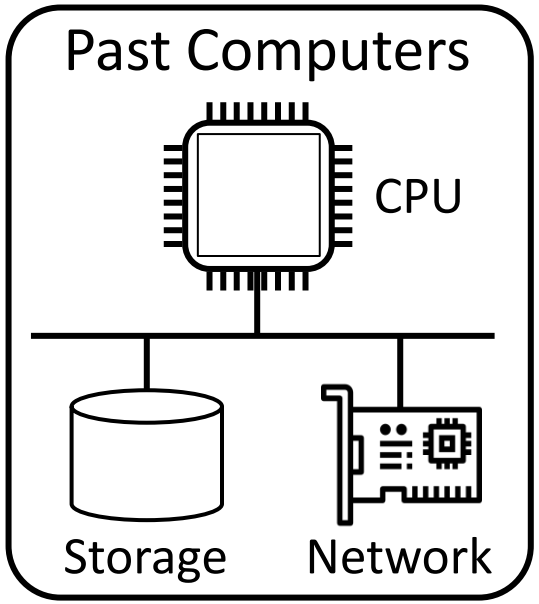
Trend of Computing Environments

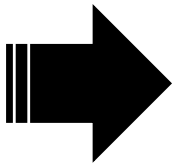
- Computing environments are becoming more complex and advanced
  - Events executed outside the CPU (i.e., off-CPU) have become more diverse

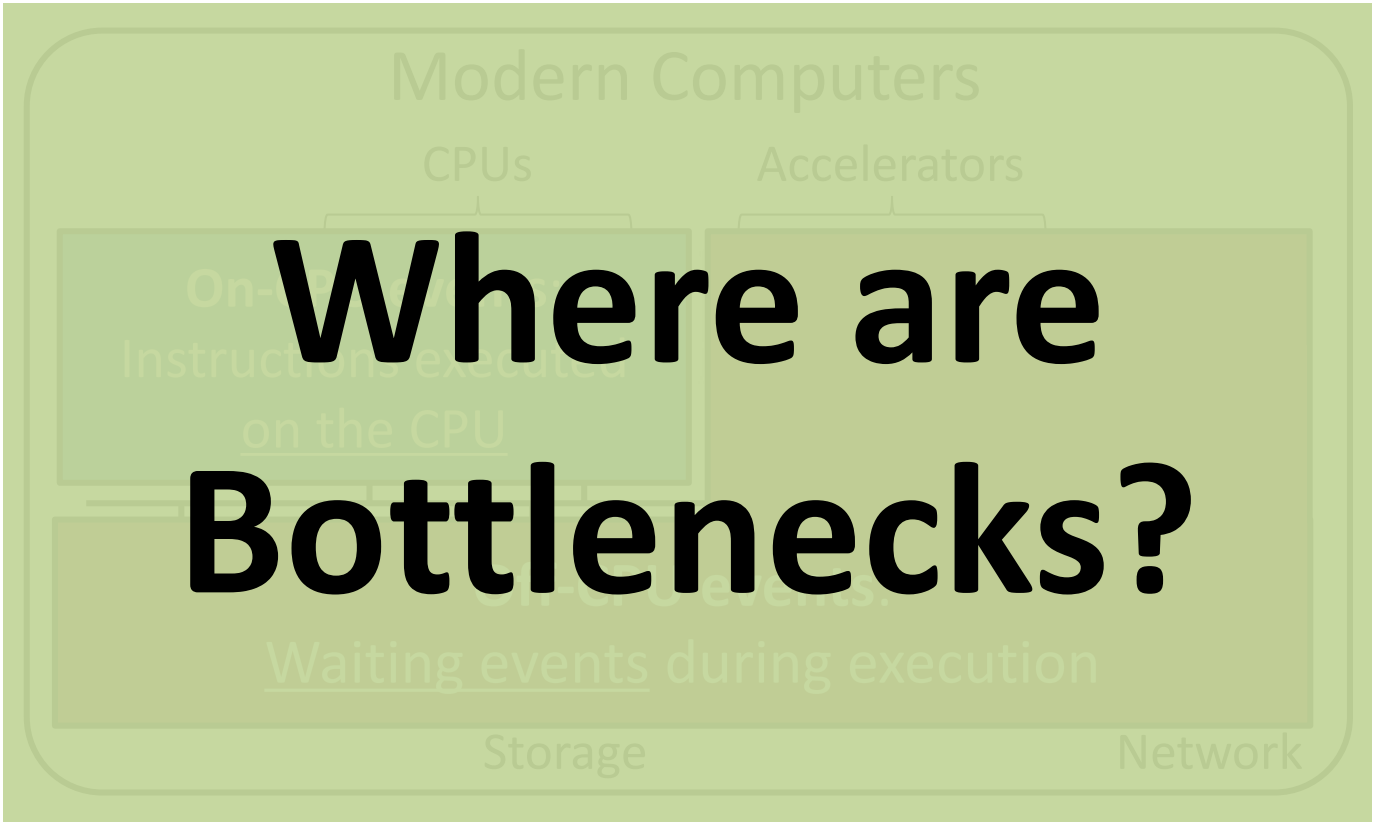


Trend of Computing Environments

- Computing environments are becoming more complex and advanced
  - Events executed outside the CPU (i.e., off-CPU) have become more diverse

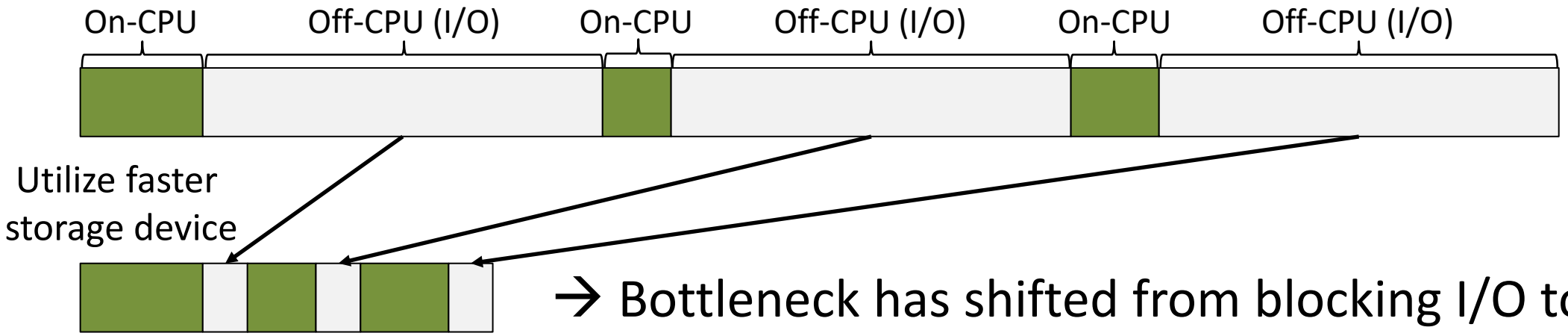






Bottlenecks of Modern Applications

- Bottlenecks of applications are diversifying
  - (I/O) Boundary between CPU-bound and I/O-bound is blurred

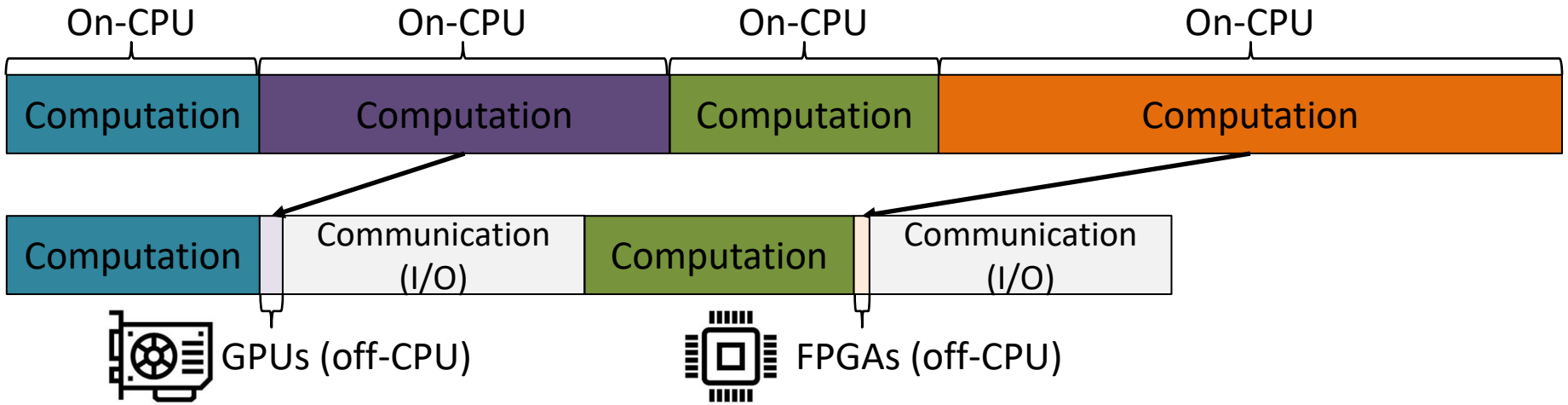


→ Bottleneck has shifted from blocking I/O to CPU

- "*kernel software is becoming the bottleneck*", XRP [OSDI '22]
- "*server CPU is becoming the bottleneck*", XSTORE [OSDI '20]
- "*Rocksdb is CPU-bound*", Kvell [SOSP '19]
- "*kernel I/O stack accounts for a large fraction*", AIOS [ATC '19]
- "*storage no longer being the bottleneck*", uDepot [FAST '19]

Bottlenecks of Modern Applications

- Bottlenecks of applications are diversifying
  - (I/O) Boundary between CPU-bound and I/O-bound is blurred
  - (Computation) Shifting away from CPU-centric computations

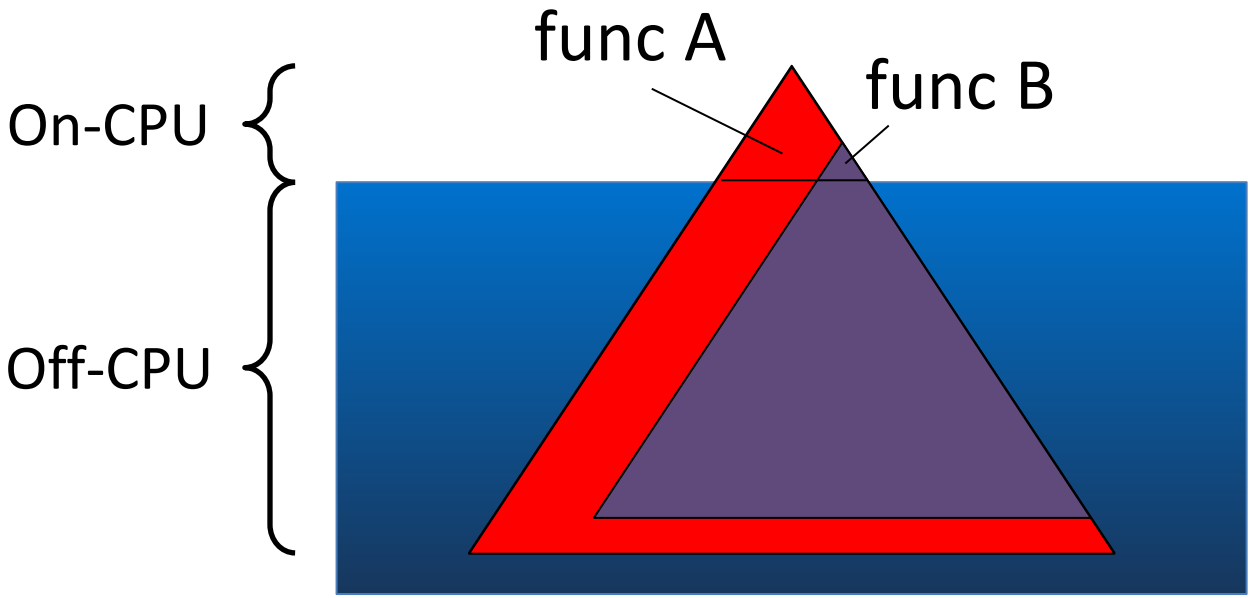


→ Bottleneck has shifted from CPU computation to I/O and communication

- "*there are spare CPU and network bandwidth*", BytePS [OSDI '20]
- "*rapid increases in GPU will shift the bottleneck towards communication*", PipeDream [SOSP '19]
- "*DNN training is not scalable, mainly due to the communication overhead*", ByteScheduler [SOSP '19]

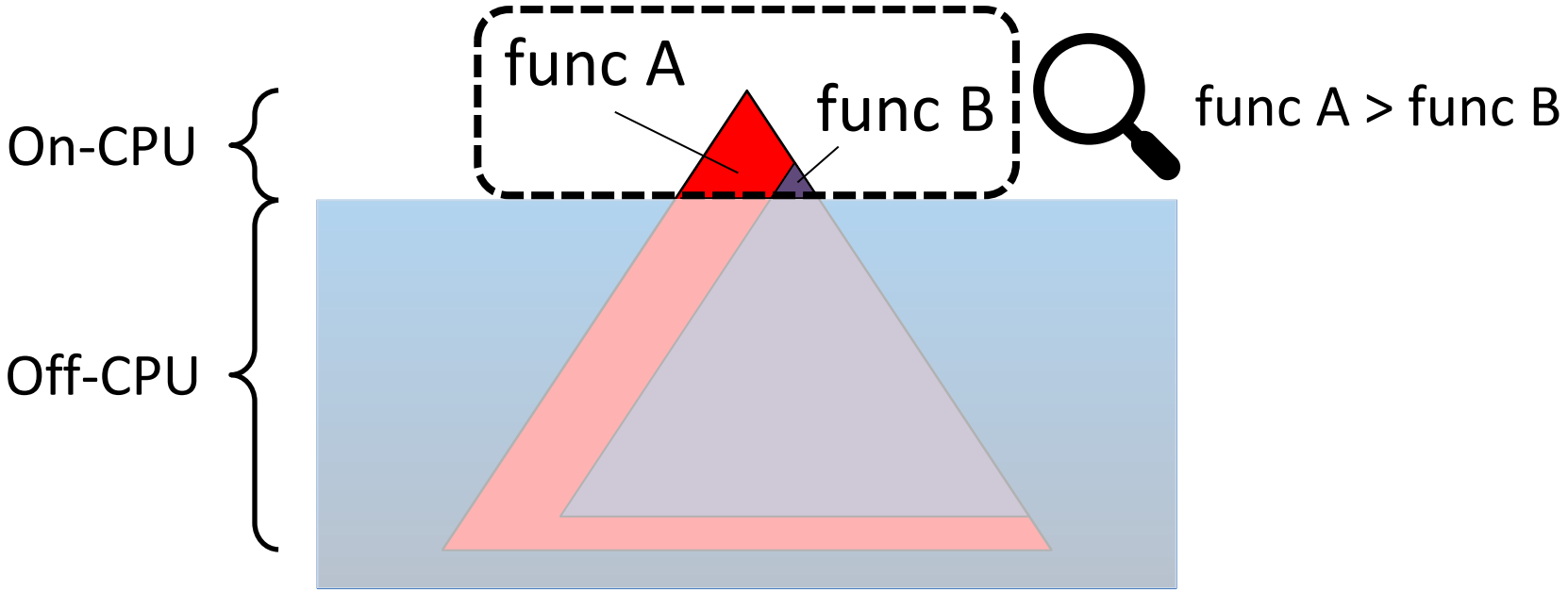
Profiling Challenge

- Both on-CPU and off-CPU events need to be considered simultaneously
  - (Challenge #1) Analysis is conducted using only partial information



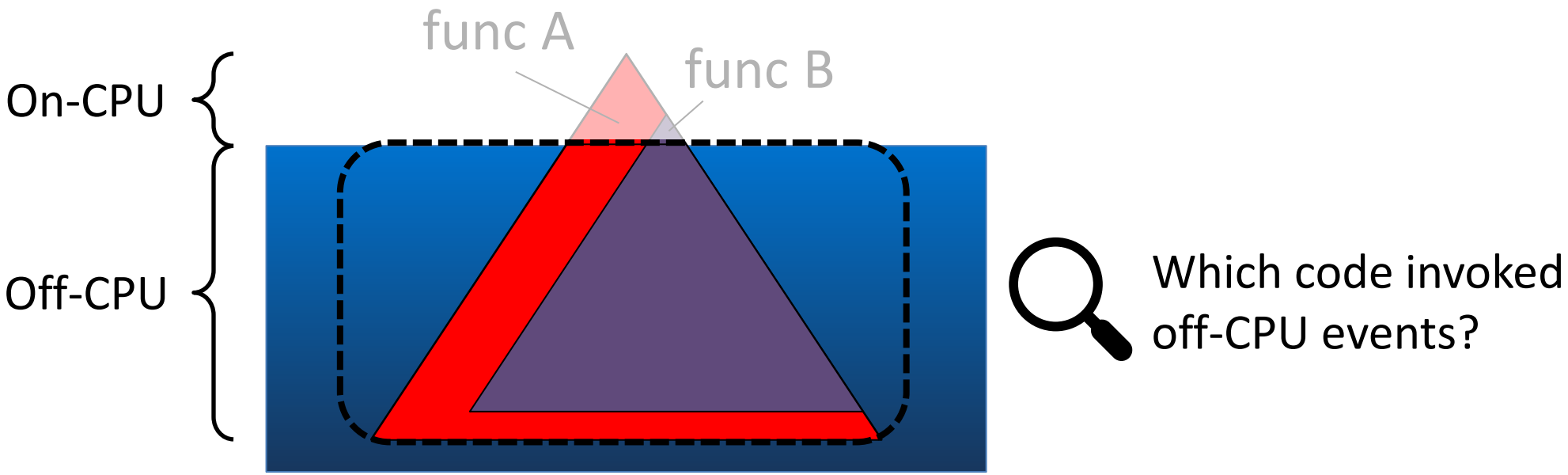
Profiling Challenge

- Both on-CPU and off-CPU events need to be considered simultaneously
  - (Challenge #1) Analysis is conducted using only partial information



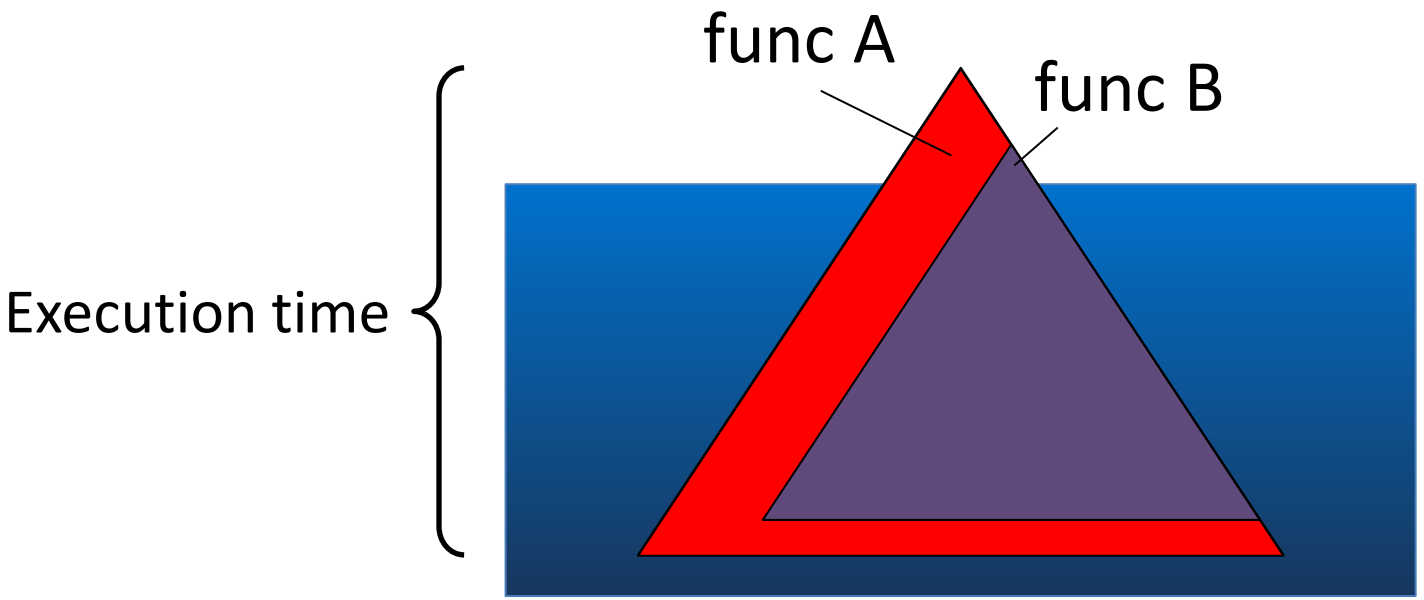
Profiling Challenge

- Both on-CPU and off-CPU events need to be considered simultaneously
  - (Challenge #1) Analysis is conducted using only partial information



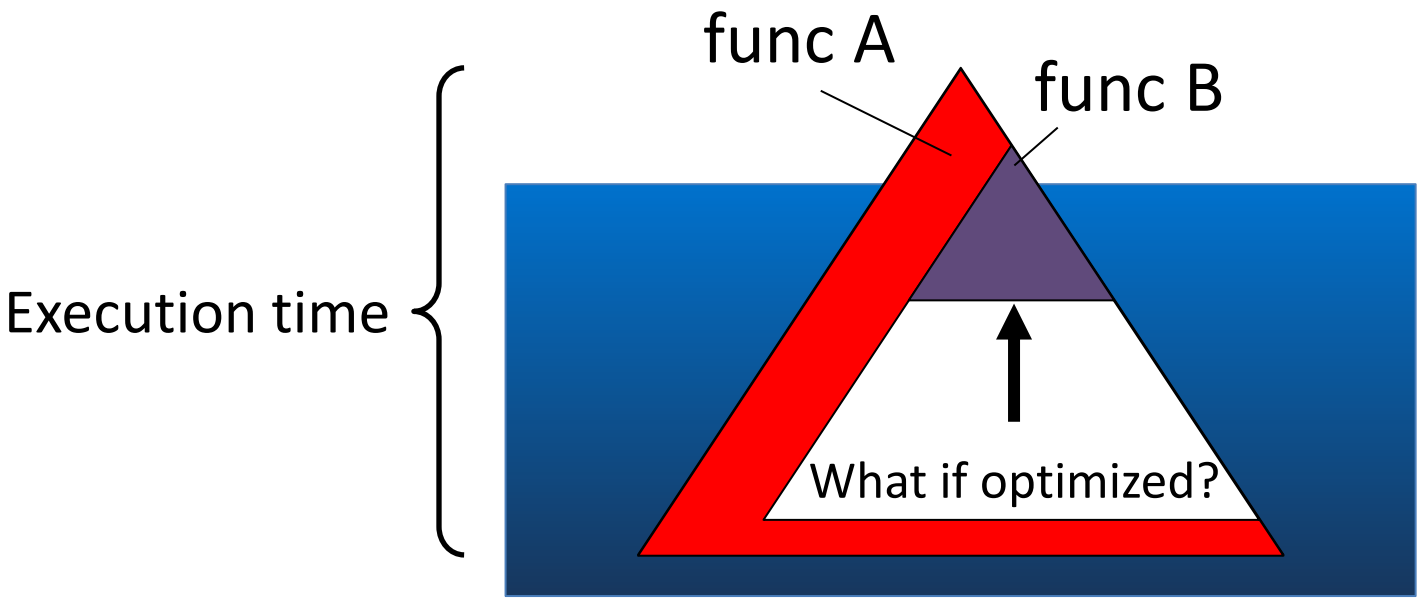
Profiling Challenge

- Both on-CPU and off-CPU events need to be considered simultaneously
  - (Challenge #1) Analysis is conducted using only partial information
  - (Challenge #2) Hard to assess the impact of optimizing off-CPU events



Profiling Challenge

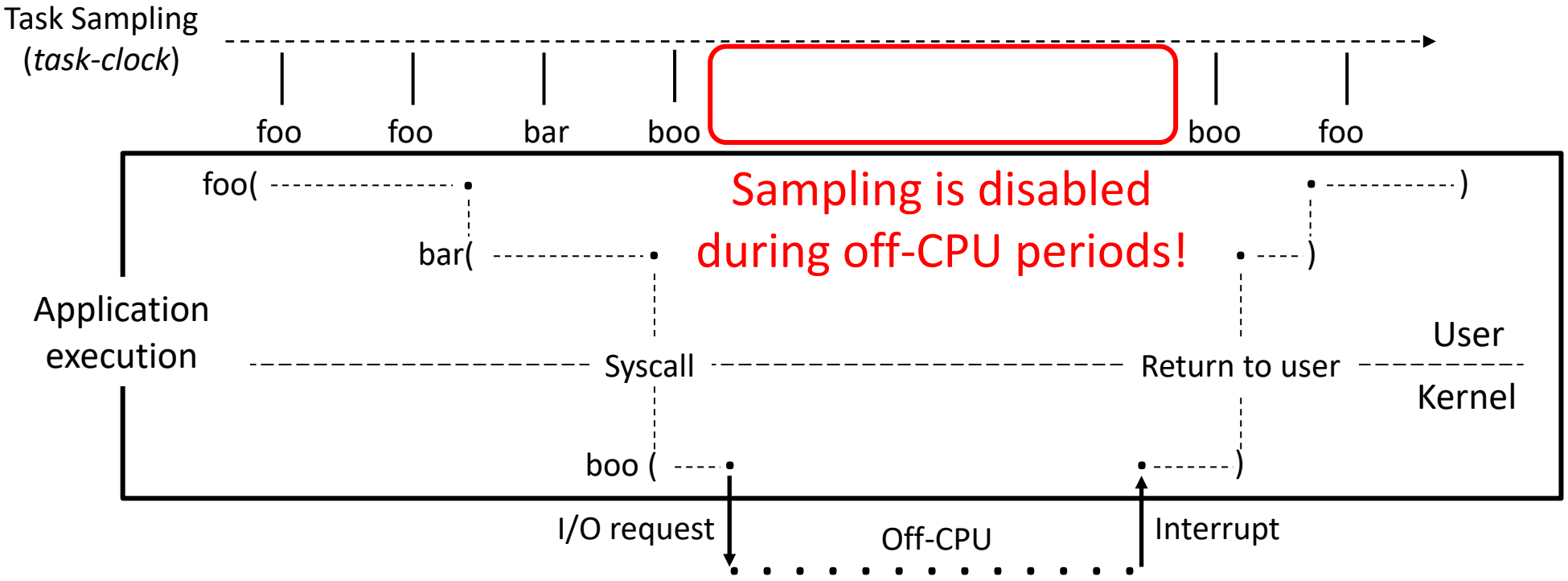
- Both on-CPU and off-CPU events need to be considered simultaneously
  - (Challenge #1) Analysis is conducted using only partial information
  - (Challenge #2) Hard to assess the impact of optimizing off-CPU events



Execution time is unchanged
→ B is not on the critical path

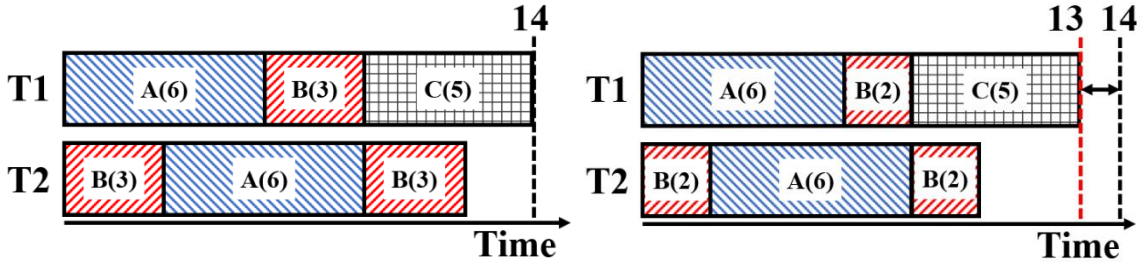
On-CPU Analysis

- Linux *perf* sampling (*task-clock*)
  - Feature in Linux kernel's perf subsystem
  - Collects profiling information (e.g., IP and callchain) periodically
  - A Low overhead, effective technique to analyze on-CPU behavior



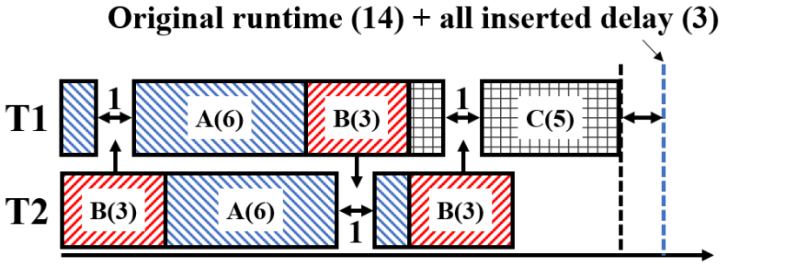
On-CPU Analysis

- COZ [SOSP '15]
  - Predict the impact of optimizing the specific code line without actual optimization
  - Virtual speedup



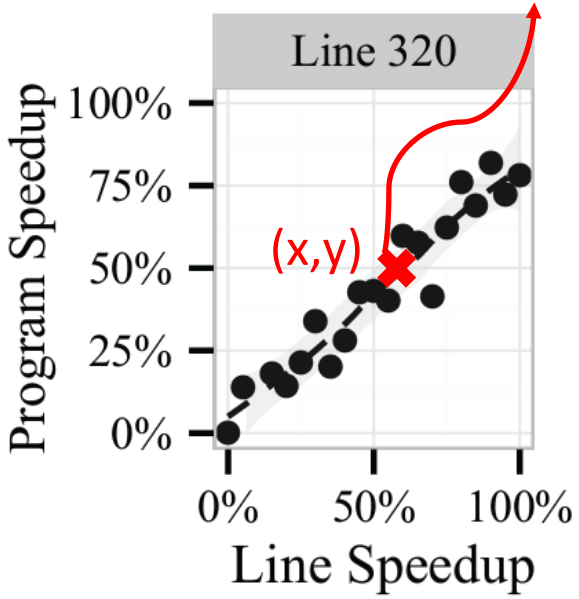
<Original application>

<Actual speedup>



<Virtual speedup>

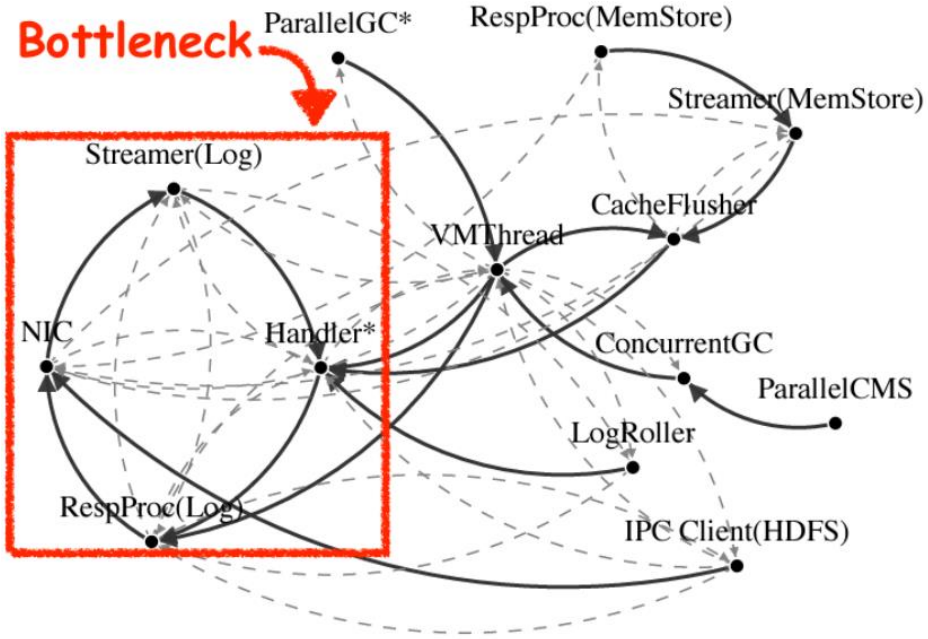
If line 320 becomes x% faster,
the program will become y% faster



COZ utilizes on-CPU sampling (Linux *perf*) → Virtual speedup is **limited to only on-CPU events**

Off-CPU Analysis

- wPerf [OSDI '18]
  - Traces all kinds of waiting events including I/O and their dependencies
  - Wait-for graph: Dependency graph of executed threads
    - Identifying closed loops (i.e., knots) through graph analysis



<Example wait-for graph>

Limitations

1) Does not provide context information of the bottleneck
   → Additional effort is needed to determine where to optimize

2) Does not provide the actual impact of optimization
   → Performance gain of the optimization could be marginal

Summary of the Limitations

→ (Limitation #1) Focuses solely on either on-CPU or off-CPU events

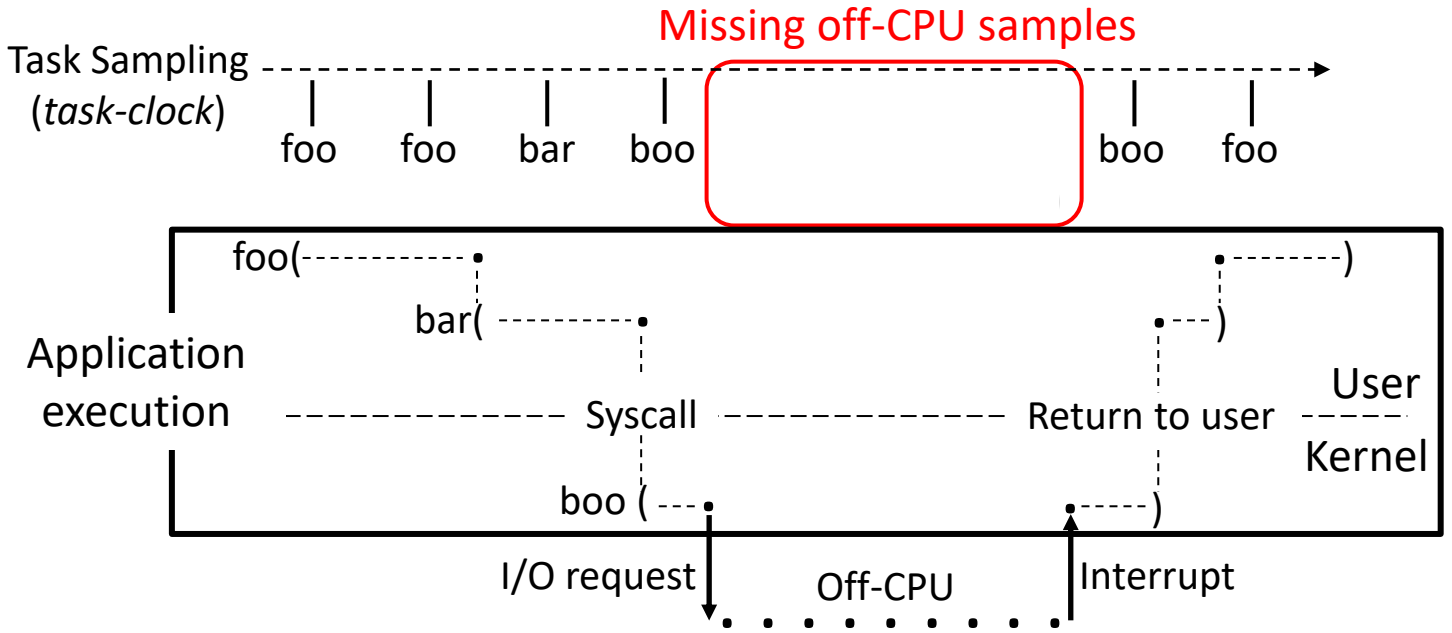
→ (Limitation #2) Causality analysis is not supported for off-CPU events

Profiler	Profiling Scope	Causality Analysis
Linux perf	On-CPU	X
COZ		△(on-CPU only)
wPerf	Off-CPU	X
Blocked Samples	Both on-/off-CPU	O

Our Approach: Blocked Samples

- Goal: sampling on- and off-CPU events simultaneously

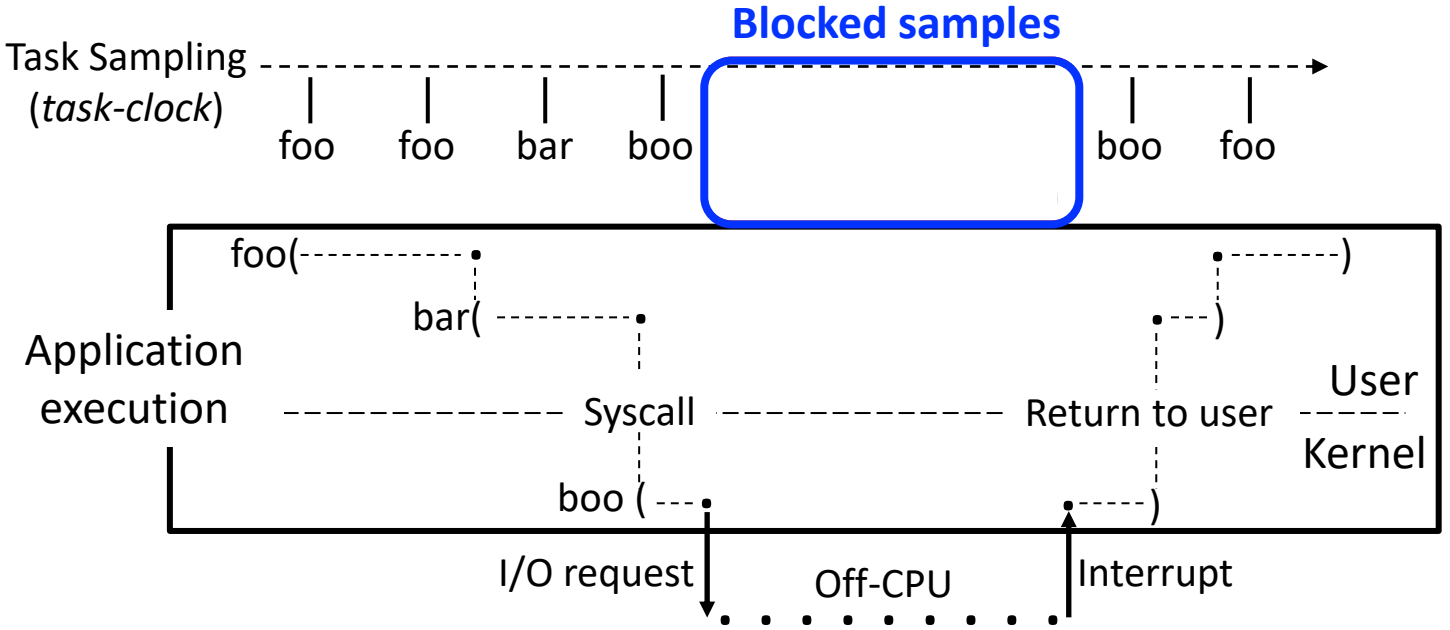
Blocked samples
(Linux perf subsystem)



Our Approach: Blocked Samples

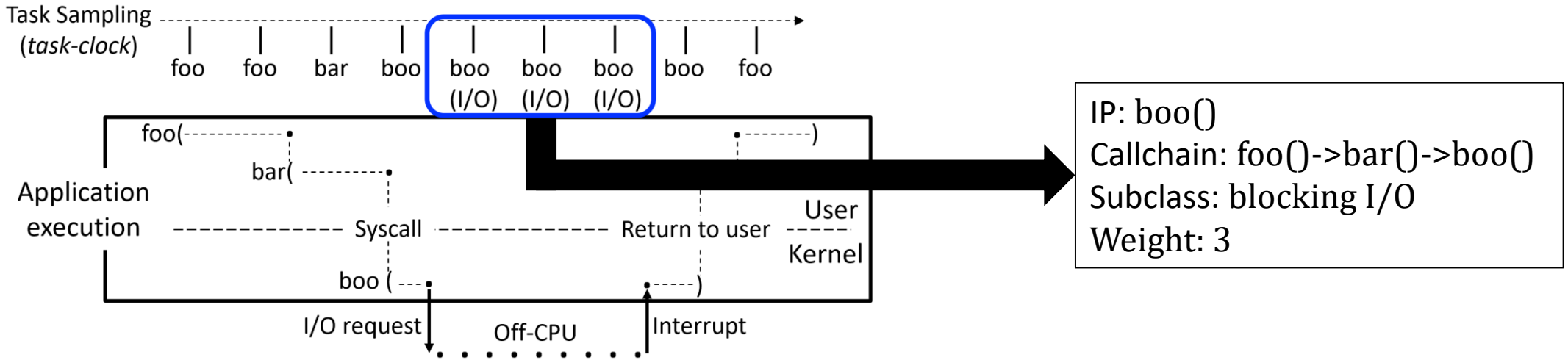
- Goal: sampling on- and off-CPU events simultaneously
  - ***Blocked samples***: sampling technique for off-CPU events
  - Proposed profilers using blocked samples
    - ***bperf***: sampling-based statistical profiler on both on-/off-CPU events
    - ***BCOZ***: causal profiler that supports virtual speedup on both on-/off-CPU events

Blocked samples
(Linux perf subsystem)



Blocked Samples

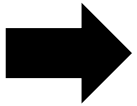
- Collected information
  - IP and callchain
  - Off-CPU subclass: reason for the blocking
    - Blocking I/O, synchronization, CPU scheduling, etc.
      - New subclasses can be defined as needed
  - Weight: # of repeats
    - Encode the number of blocked samples with the same attributes



bperf: Statistical Profiler on Both On-/Off-CPU Events

- Extension of Linux perf tool to support blocked samples
  - Sample accounting
    - bperf accounts blocked samples with on-CPU samples on the same dimension
    - bperf classifies samples considering IP, callchain, and subclasses of blocked samples
  - Result reporting
    - New symbol annotations for blocked samples
      - [I]: blocking I/O, [L]: synchronization, [S]: CPU scheduling, [B]: others
    - Both the last user-level IP and last kernel-level IP are reported for blocked samples
      - Enables an in-depth understanding of off-CPU events

```
while(N++ < 100000) {
   write();
   fsync();
}
```



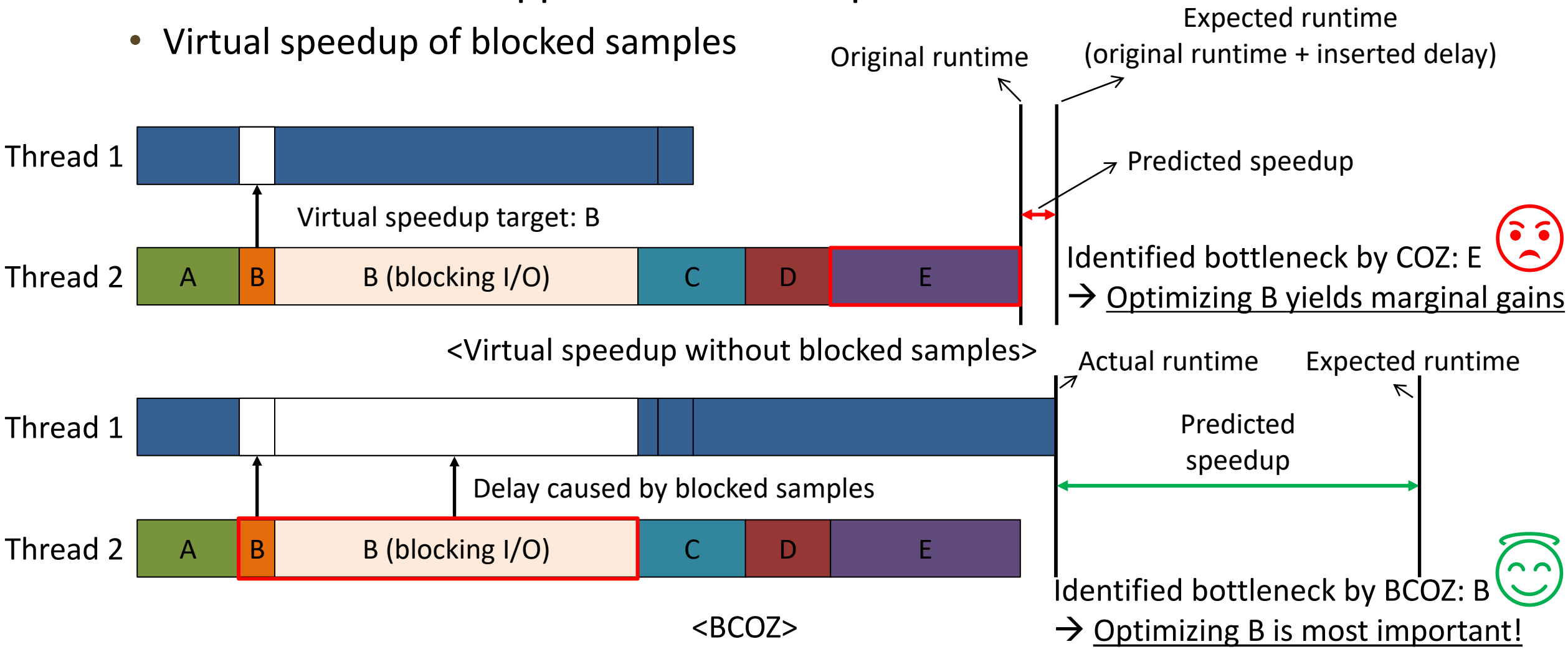
```
# Overhead  Command  Shared Object      Symbol
# ........  .......  .................  ....................................
#
    55.35%  test_io  [kernel.vmlinux]  [I] wait_on_page_bit
                                         ---[.] fsync
    27.12%  test_io  [kernel.vmlinux]  [B] jbd2_log_wait_commit
                                         ---[.] fsync
     2.78%  test_io  [kernel.vmlinux]  [k] copy_user_enhanced_fast_string
     1.74%  test_io  [kernel.vmlinux]  [k] _raw_spin_unlock_irqrestore
```

Data block write

Waiting for jbd2 thread

BCOZ: Causal Profiler on Both On-/Off-CPU Events

- Extension of COZ to support blocked samples
  - Virtual speedup of blocked samples



Features and Challenges of BCOZ

- Features
  - Sampling kernel codes
  - Virtual speedup of blocked samples
  - Subclass-level virtual speedup

- Challenges
  - Conflicts with optimization of original COZ
    - Dependency handling + batch processing of samples

→ For more details, please refer to the paper

Experimental Setup

- CPU: Intel Xeon Gold 5218 2.30GHz * 2
- OS: Ubuntu 20.04 Server (Linux kernel version: 5.3.7)
- Memory: DDR4 2933MHz, 384GB
- Storage devices: Samsung NVMe PM1735 (1,500K IOPS)
- Questions:
  - Q1) Can blocked samples identify true bottlenecks?
  - Q2) Differences from wPerf's results?
  - Q3) Profiling overhead?
    - Comparison of tracing (off-CPU only), sampling (on-CPU only), bperf (both on-/off-CPU)
    - BCOZ overhead analysis

  → Please refer to the paper

Summary of the Profiling Results

- Results included in the paper

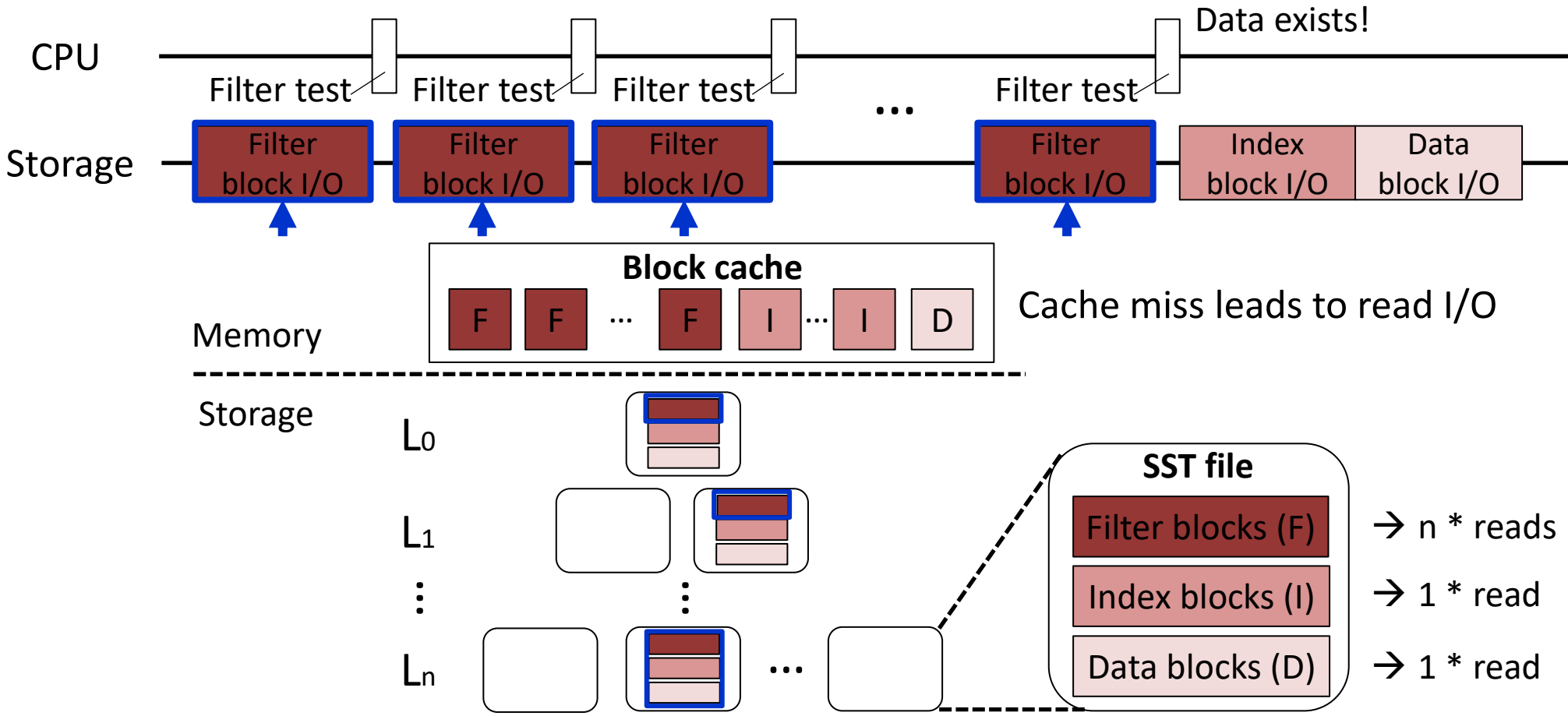
Benchmark	Workload	Identified bottlenecks	Optimization	Speedup?	Known solution?	
RocksDB	prefix_dist	Block cache contention	- Sharding	O (3.4x)	Yes	Case study 2
	allrandom	Block read I/O	- Asynchronous I/O	O (1.8x)	No	Case study 1
	fillrandom	Compaction, write stall	- No block compression - Increase the number of compaction thread - Reduce write stall	O (2.6x)	Yes	
NPB	Integer sort	CPU contention	- Allocate more CPU cores	O (16.4x)	Yes	

- Results not included in the paper (optimization is ongoing)

Benchmark	Identified Bottlenecks
HPCG	Serialized SYMGS (Symmetric Gauss Seidel) kernel
LLaMA-cpp	Blocking I/O in *ggml_vec_dot*

Case Study 1– RocksDB (Block Read Operation)

- Scenario: read-only workload (*allrandom*), small block cache (0.1% of dataset size)
- Problem: frequent block (filter, index, data) read I/Os



Case Study 1– RocksDB (Block Read Operation)

- Scenario: read-only workload (*allrandom*), small block cache (0.1% of dataset size)
- Identified bottlenecks: blocking disk I/O (filter, index, and data blocks)

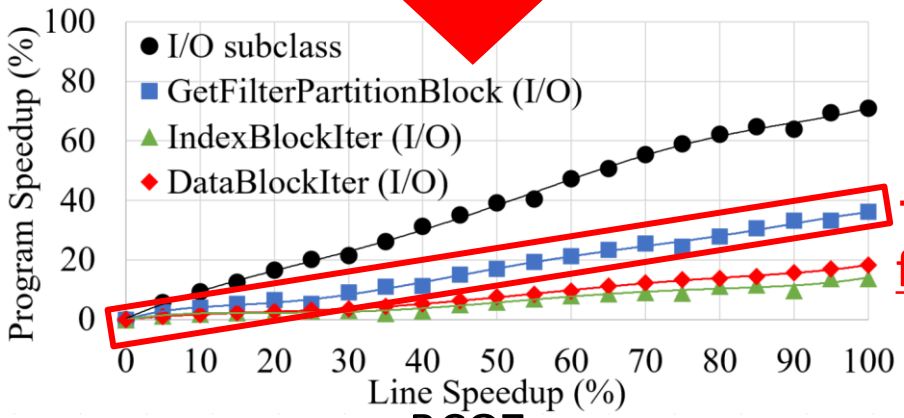
```
Samples: 1M of event 'task-clock', Event count (approx.): 1074412000000
  Overhead  Command          Shared Object       Symbol
-   85.33%  db_bench_vanill  libpthread-2.30.so  [I] __libc_pread64
   - __libc_pread64
      - rocksdb::PosixRandomAccessFile::Read
        rocksdb::RandomAccessFileReader::Read
      - rocksdb::BlockFetcher::ReadBlockContents
          - 45.09% rocksdb::BlockBasedTable::MaybeReadBlockAndLoadToCache<rocksdb::Block>
             - rocksdb::BlockBasedTable::RetrieveBlock<rocksdb::Block>
                + 23.37% rocksdb::BlockBasedTable::NewDataBlockIterator<rocksdb::DataBlockIter>
                + 21.40% rocksdb::BlockBasedTable::NewDataBlockIterator<rocksdb::IndexBlockIter>
          - 40.23% rocksdb::BlockBasedTable::MaybeReadBlockAndLoadToCache<rocksdb::ParsedFullFilterBlock>
               rocksdb::BlockBasedTable::RetrieveBlock<rocksdb::ParsedFullFilterBlock>
               rocksdb::PartitionedFilterBlockReader::GetFilterPartitionBlock
```

Blocking disk I/O

Context information

Causality analysis

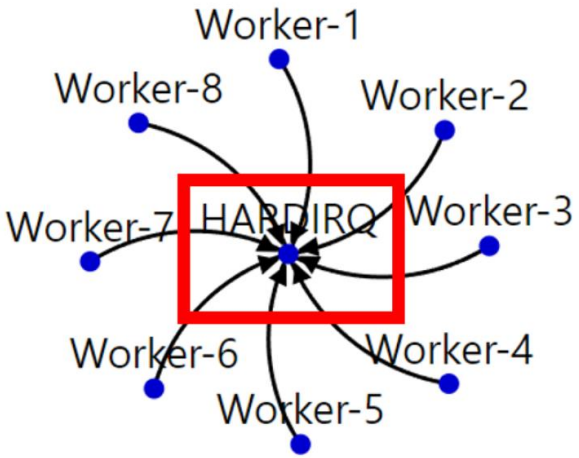
<*bperf*>



<BCOZ>

→Optimizing disk I/O of filter block is most important!

Identified bottleneck: blocking disk I/O (Worker*→HARDIRQ)



<Wait-for graph of wPerf>

→ Contexts related to disk I/Os are missing (Limitation #1)

Case Study 1– RocksDB (Block Read Operation)

- Scenario: read-only workload (*allrandom*), small block cache (0.1% of dataset size)
- Identified bottlenecks: blocking disk I/O (filter, index, and data blocks)

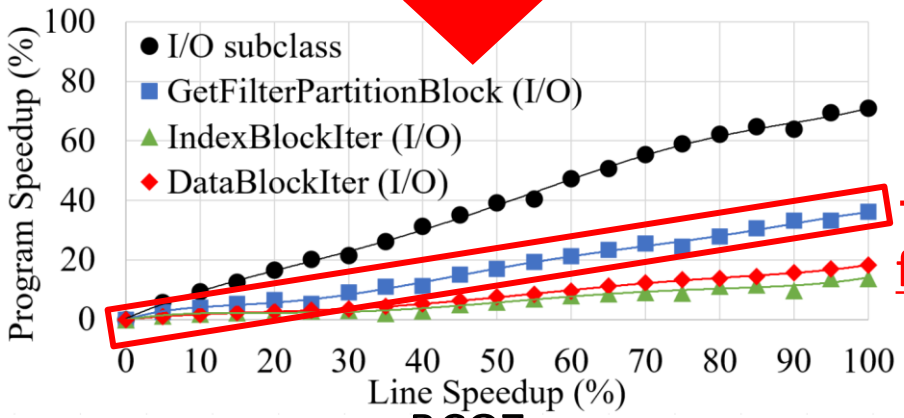
```
Samples: 1M of event 'task-clock', Event count (approx.): 1074412000000
  Overhead  Command          Shared Object       Symbol
-   85.33%  db_bench_vanill  libpthread-2.30.so  [I] __libc_pread64
   - __libc_pread64
      - rocksdb::PosixRandomAccessFile::Read
        rocksdb::RandomAccessFileReader::Read
      - rocksdb::BlockFetcher::ReadBlockContents
          - 45.09% rocksdb::BlockBasedTable::MaybeReadBlockAndLoadToCache<rocksdb::Block>
             - rocksdb::BlockBasedTable::RetrieveBlock<rocksdb::Block>
                + 23.37% rocksdb::BlockBasedTable::NewDataBlockIterator<rocksdb::DataBlockIter>
                + 21.40% rocksdb::BlockBasedTable::NewDataBlockIterator<rocksdb::IndexBlockIter>
          - 40.23% rocksdb::BlockBasedTable::MaybeReadBlockAndLoadToCache<rocksdb::ParsedFullFilterBlock>
               rocksdb::BlockBasedTable::RetrieveBlock<rocksdb::ParsedFullFilterBlock>
               rocksdb::PartitionedFilterBlockReader::GetFilterPartitionBlock
```

Blocking disk I/O

Context information

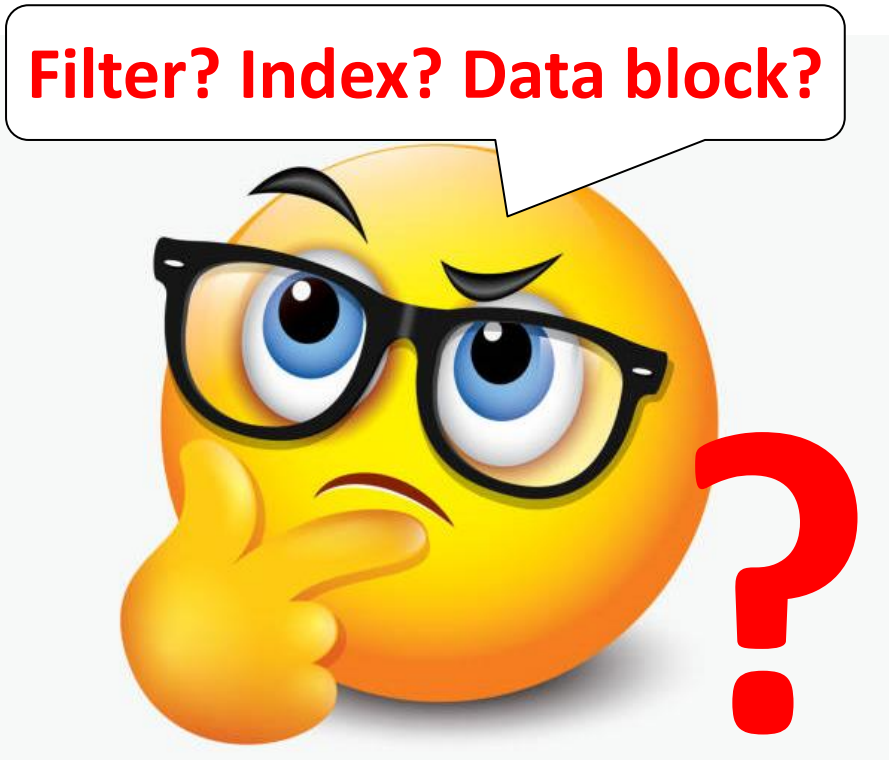
Causality analysis

<*bperf*>



<BCOZ>

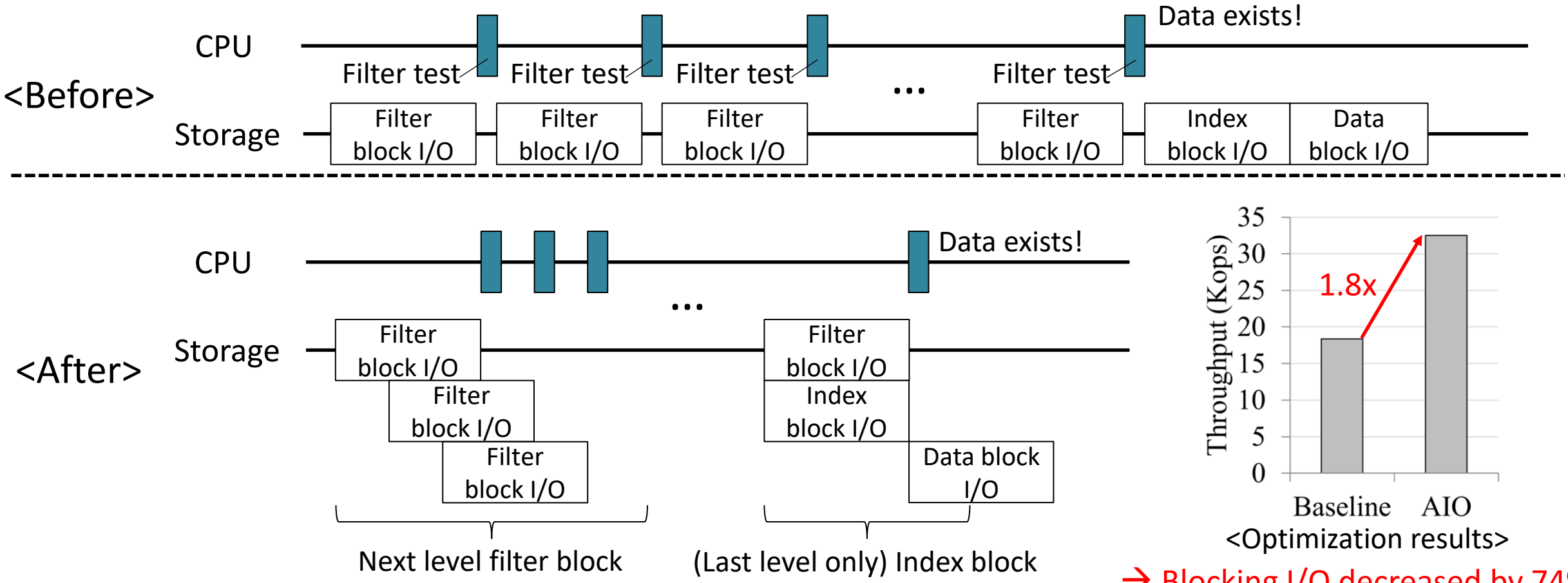
→Optimizing disk I/O of filter block is most important!



→ Contexts related to disk I/Os are missing (Limitation #1)

Case Study 1– RocksDB (Block Read Operation)

- Scenario: read-only workload (*allrandom*), small block cache (0.1% of dataset size)
- Optimization: asynchronous I/O for filter and index blocks

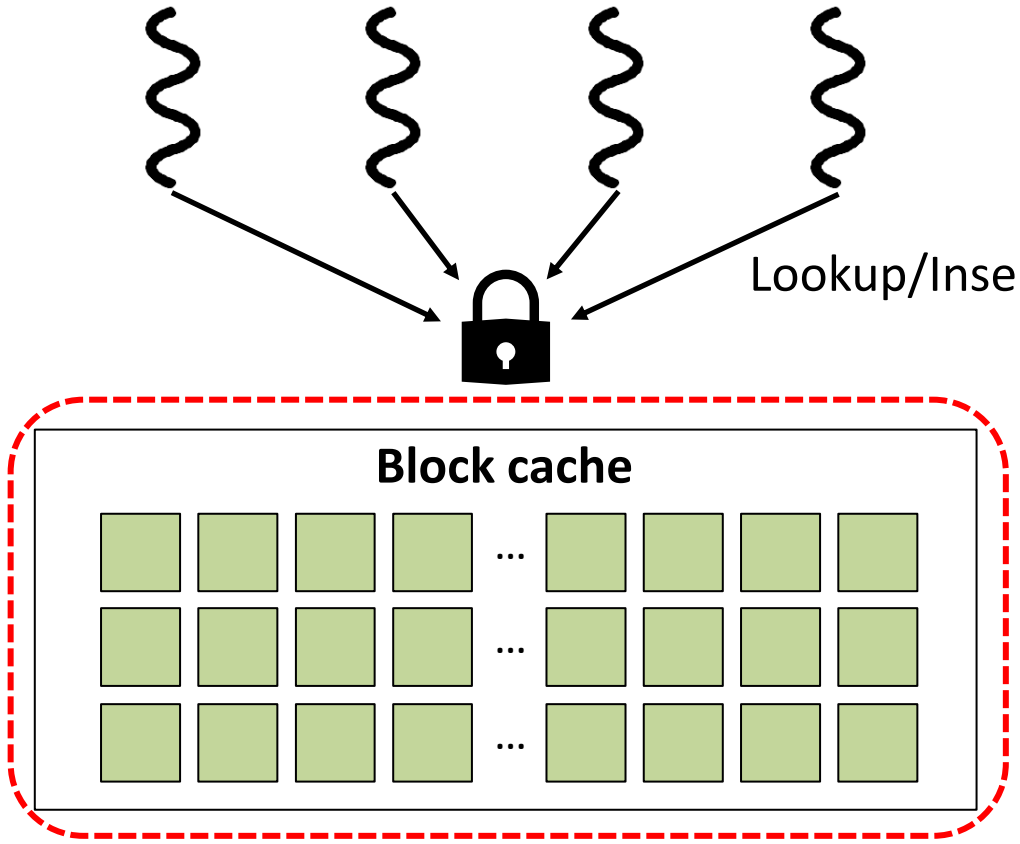


<Optimization results>

→ Blocking I/O decreased by 74%

Use Case 2 – RocksDB (Block Cache Contention)

- Scenario: read-only workload (*prefix_dist*), large block cache (10% of dataset size)
- Problem: block cache lock contention



Frequent block cache access leads to lock contention

Use Case 2 – RocksDB (Block Cache Contention)

- Scenario: read-only workload (*prefix_dist*), large block cache (10% of dataset size)
- Identified bottlenecks: lock-waiting

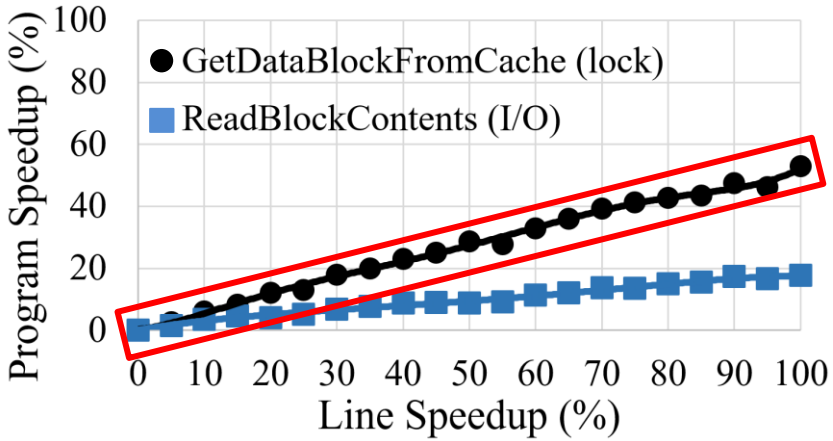
```
Samples: 1M of event 'task-clock', Event count (approx.): 1097249000000
  Overhead  Command          Shared Object       Symbol
+   25.27%  db_bench_vanill  [kernel.vmlinux]    [k] native_queued_spin_lock_slowpath
-   24.16%  db_bench_vanill  libpthread-2.30.so  [L] __lll_lock_wait
   - 24.09% __lll_lock_wait
      - __pthread_mutex_lock
      - rocksdb::port::Mutex::Lock
         - 12.51% rocksdb::LRUCacheShard::Lookup
              rocksdb::ShardedCache::Lookup
            - rocksdb::BlockBasedTable::GetEntryFromCache
               + 8.05% rocksdb::BlockBasedTable::GetDataBlockFromCache<rocksdb::Block>
               + 4.46% rocksdb::BlockBasedTable::GetDataBlockFromCache<rocksdb::ParsedFullFilterBlock>
         + 11.55% rocksdb::LRUCacheShard::Release
+    6.24%  db_bench_vanill  [kernel.vmlinux]    [k] _raw_spin_unlock_irqrestore
                    • • •
+    1.01%  db_bench_vanill  libpthread-2.30.so  [I] __libc_pread64
```

Lock-waiting

Causality analysis

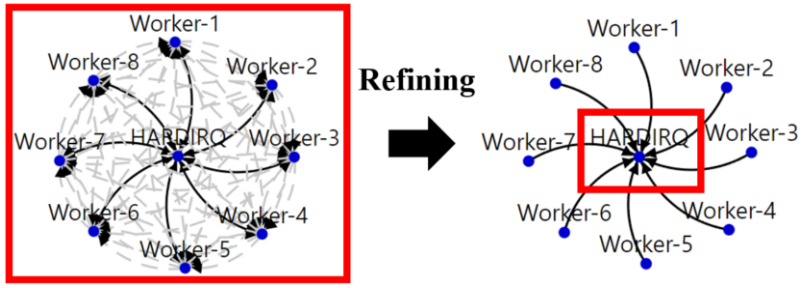
Context information

Blocking I/O



→ Optimizing lock-contention is more important than disk I/O

Identified bottleneck: blocking disk I/O, lock-waiting
(Worker*→HARDIRQ, Worker*←→Worker*)



(Limitation #1)
→ Codes that invoke lock-contention are missing

(Limitation #2)
→ Actual impact of optimizing blocking disk I/O is missing

Use Case 2 – RocksDB (Block Cache Contention)

- Scenario: read-only workload (*prefix_dist*), large block cache (10% of dataset size)
- Identified bottlenecks: lock-waiting

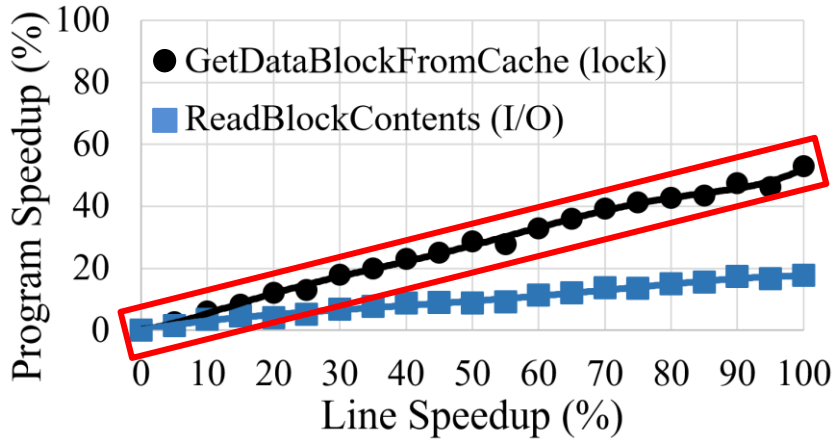
```
Samples: 1M of event 'task-clock', Event count (approx.): 1097249000000
  Overhead  Command          Shared Object       Symbol
+   25.27%  db_bench_vanill  [kernel.vmlinux]    [k] native_queued_spin_lock_slowpath
-   24.16%  db_bench_vanill  libpthread-2.30.so  [L] __lll_lock_wait
   - 24.09% __lll_lock_wait
      - __pthread_mutex_lock
      - rocksdb::port::Mutex::Lock
         - 12.51% rocksdb::LRUCacheShard::Lookup
              rocksdb::ShardedCache::Lookup
            - rocksdb::BlockBasedTable::GetEntryFromCache
               + 8.05% rocksdb::BlockBasedTable::GetDataBlockFromCache<rocksdb::Block>
               + 4.46% rocksdb::BlockBasedTable::GetDataBlockFromCache<rocksdb::ParsedFullFilterBlock>
         + 11.55% rocksdb::LRUCacheShard::Release
+    6.24%  db_bench_vanill  [kernel.vmlinux]    [k] _raw_spin_unlock_irqrestore
                                                  • • •
+    1.01%  db_bench_vanill  libpthread-2.30.so  [I] __libc_pread64
```

Lock-waiting

Causality analysis

Context information

Blocking I/O



→ Optimizing lock-contention is more important than disk I/O

Lock or I/O?

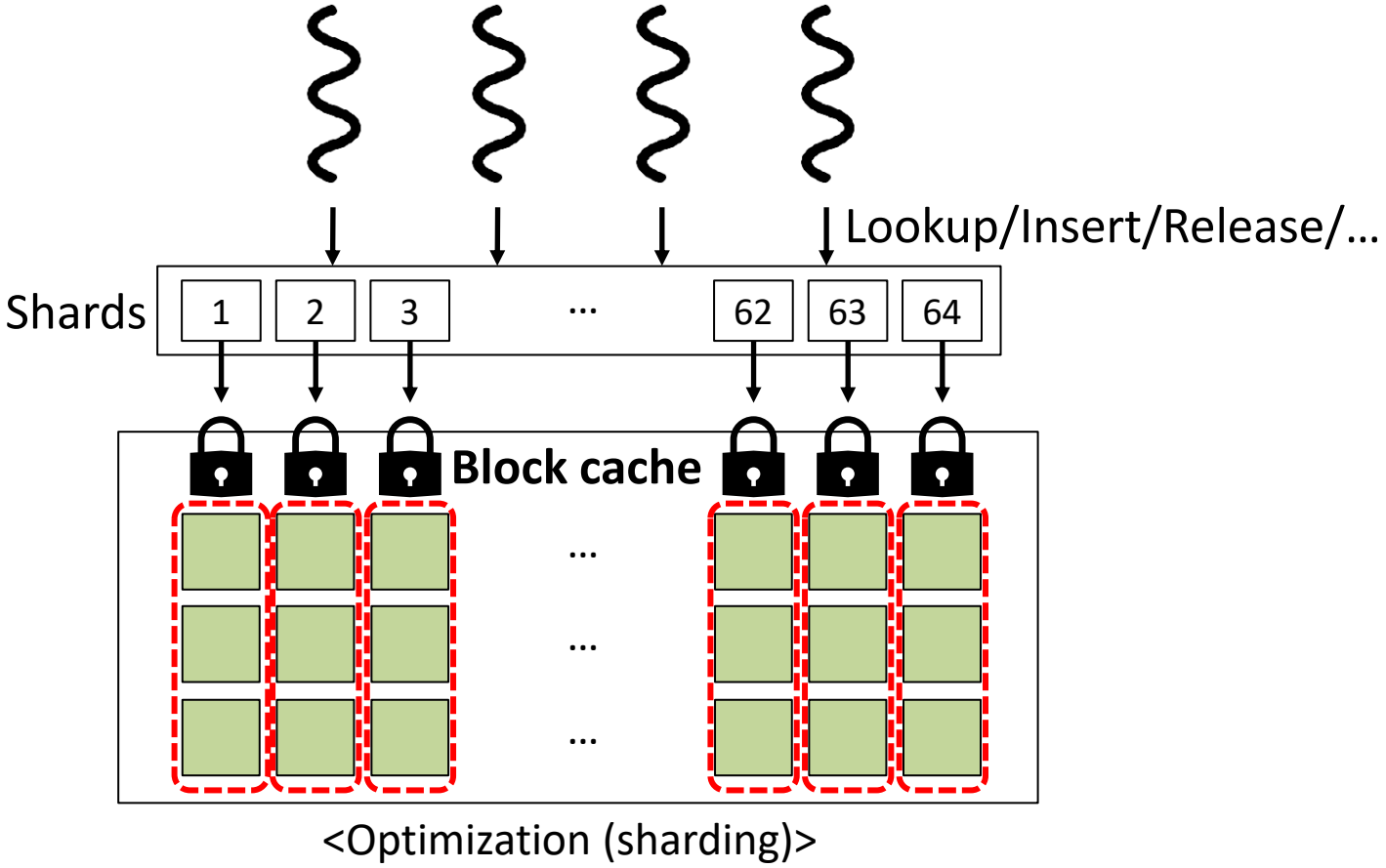
Lookup? Insert? Release?

(Limitation #1)
→ Codes that invoke lock-contention are missing

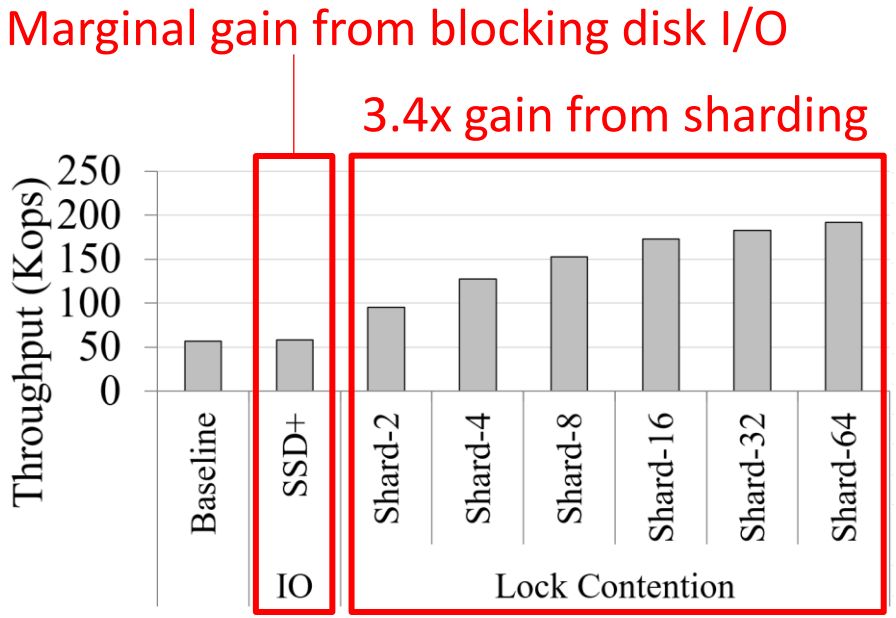
(Limitation #2)
→ Actual impact of optimizing blocking disk I/O is missing

Use Case 2 – RocksDB (Block Cache Contention)

- Scenario: read-only workload (*prefix_dist*), large block cache (10% of dataset size)
- Optimization: apply sharding



<Optimization (sharding)>



→ Lock-contention decreased by 97%

<Optimization results>

Conclusion

- Profiling modern applications has become more challenging
- ***Blocked samples*** collects off-CPU events information
  - ***bperf***, provides statistical profiling of both on-/off-CPU events
  - ***BCOZ***, provides virtual speedup of both on-/off-CPU events
- Blocked samples, a general solution for off-CPU sampling
  - Planning on enriching blocked samples with off-CPU information details (device-internal ops.)

Blocked samples is available at:

https://github.com/s3yonsei/blocked_samples

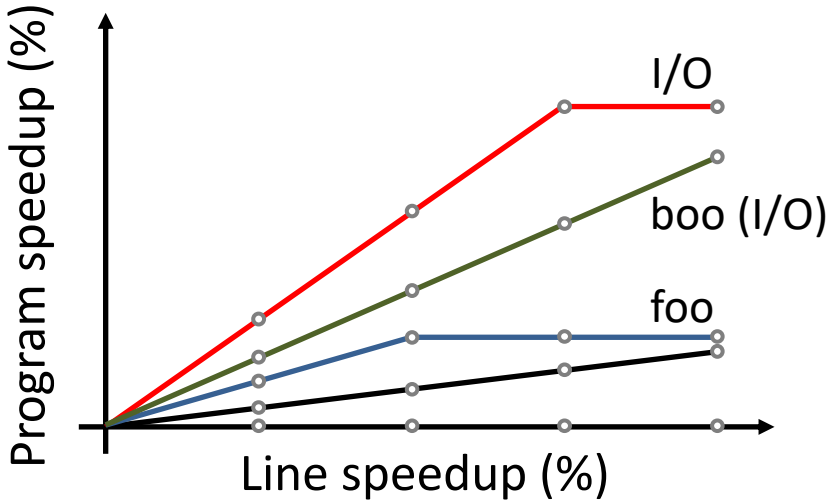
Thank you!

Q&A

bperf

# Overhead	Symbol
40.00%	[I/O] fsync
	- 25.00% [.] foo
	- 15.00% [.] boo
20.00%	[LOCK] mutex_lock
	--- [.] bar
15.00%	[.] compute
	. . .

BCOZ



Blocked samples (Linux perf subsystem)