Application Performance Profiling using Blocked Samples

02/18/2025

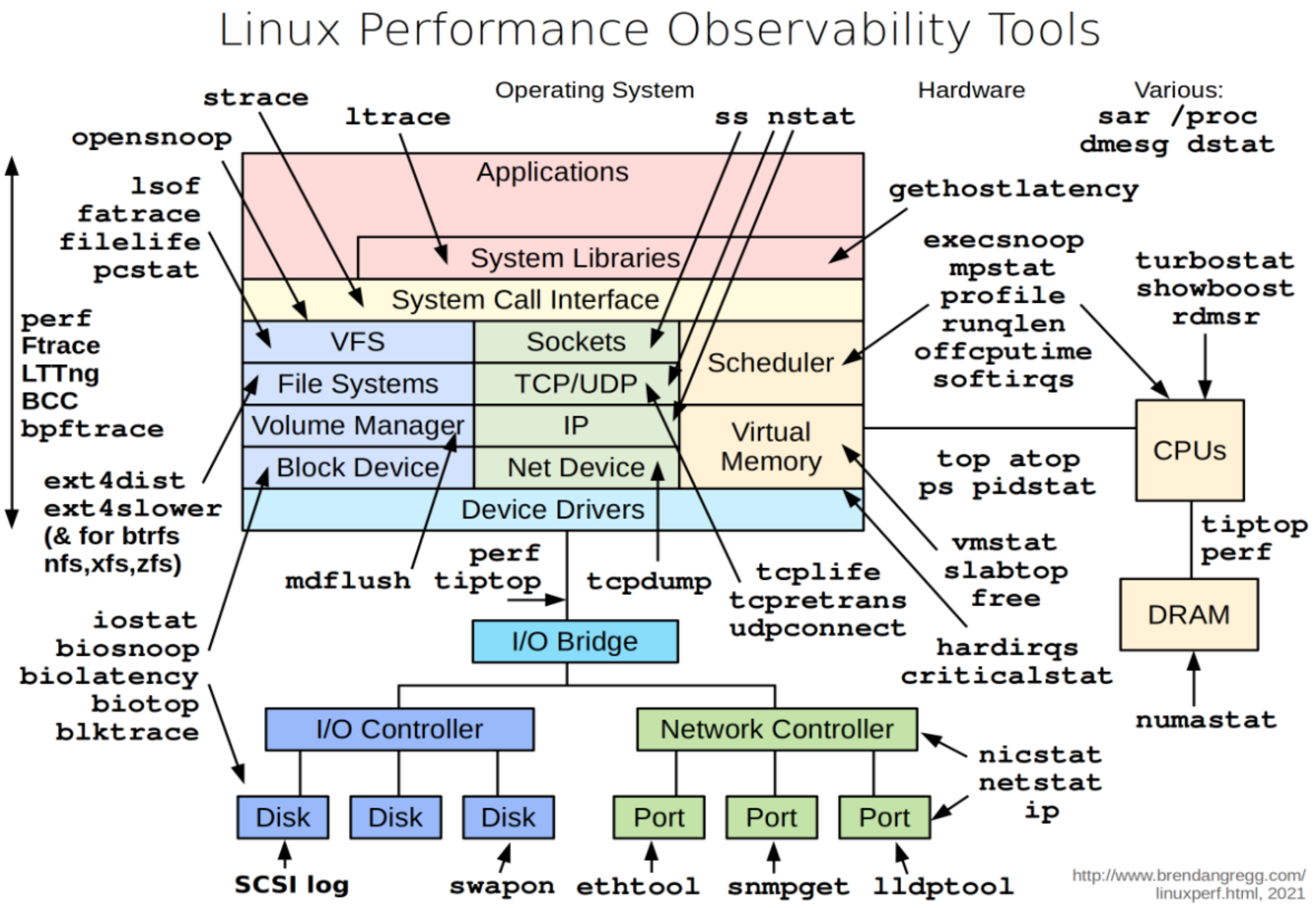
Jinkyu Jeong

Department of Computer Science and Engineering

연세대학교
YONSEI UNIVERSITY

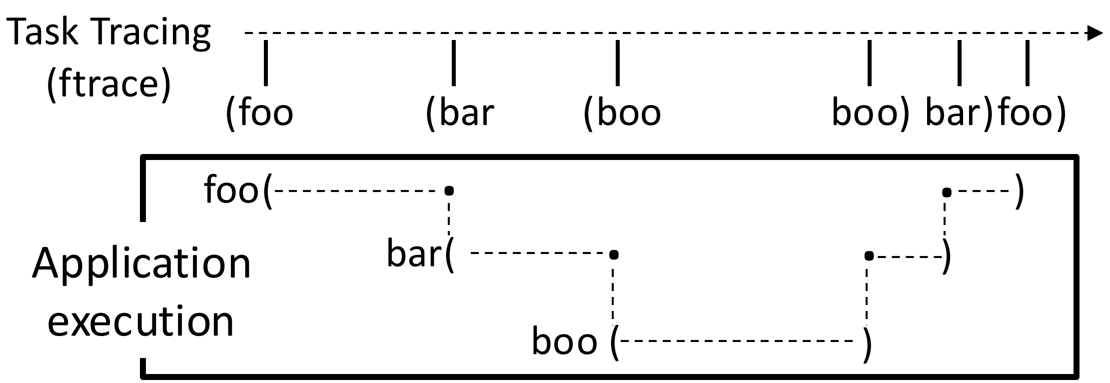
Linux Performance Analysis

- Various system tools for different system events
  - Application
  - OS
  - Blocking I/O (device ops.)
- Linux *perf* is widely (and generally) used performance profiling tool

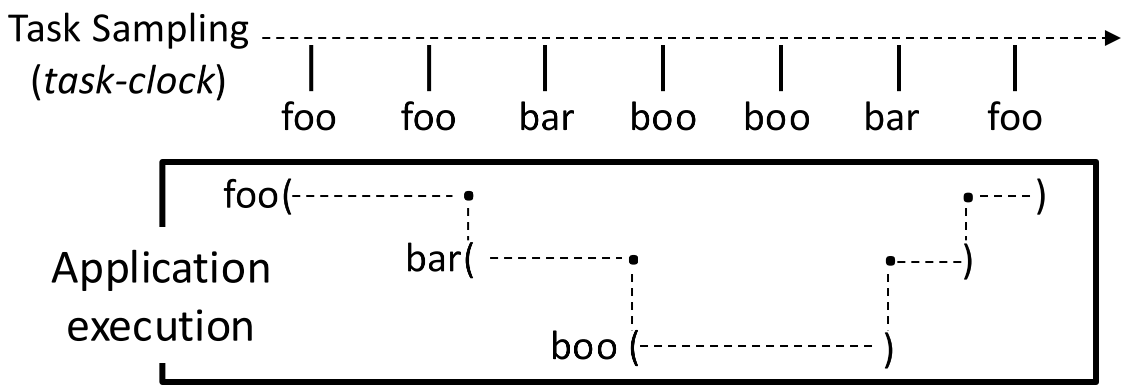


Linux perf Subsystem

- Supports performance profiling through collecting execution information
  - Collects IP and callchain
- Tracing vs. Sampling
  - Tracing(=instrument): collects <u>every events</u>
    - e.g., Linux ftrace or tracepoints
  - Sampling: collects samples of events <u>periodically</u>
    - e.g., Linux perf record task-clock



<Tracing example>



<Sampling example>

Linux perf Sampling (task-clock)

- Linux perf sampling co-operates with the periodic timer (i.e., HR timer)
  - e.g., $perf record -g -e task-clock -c 1000000 ./a.out
    - '-g': callchain, '-e task-clock': event to collect, '-c 1000000': period (=1ms)

```
void func_a() {
    while (i < 20000000)      i++;
}
void func_b() {
    while (i < 40000000)      i++;
}

int main (int argc, char *argv[]) {
    func_a();
    func_b();

    return 0;
}
```

```
a.out    37196 331011.093831:     1000000 task-clock:
            55fda5e0d13e func_a+0x15 (/home/mw/benchmarks/a.out)
            55fda5e0d17f main+0x12 (/home/mw/benchmarks/a.out)
            7fe0c1a2ddbb __libc_start_call_main+0x6b (/usr/local/lib/glibc-testing/lib/libc.so.6)
            7fe0c1a2de76 __libc_start_main@@GLIBC_2.34+0x86 (/usr/local/lib/glibc-testing/lib/libc.so.6)
            55fda5e0d065 _start+0x25 (/home/mw/benchmarks/a.out)
```

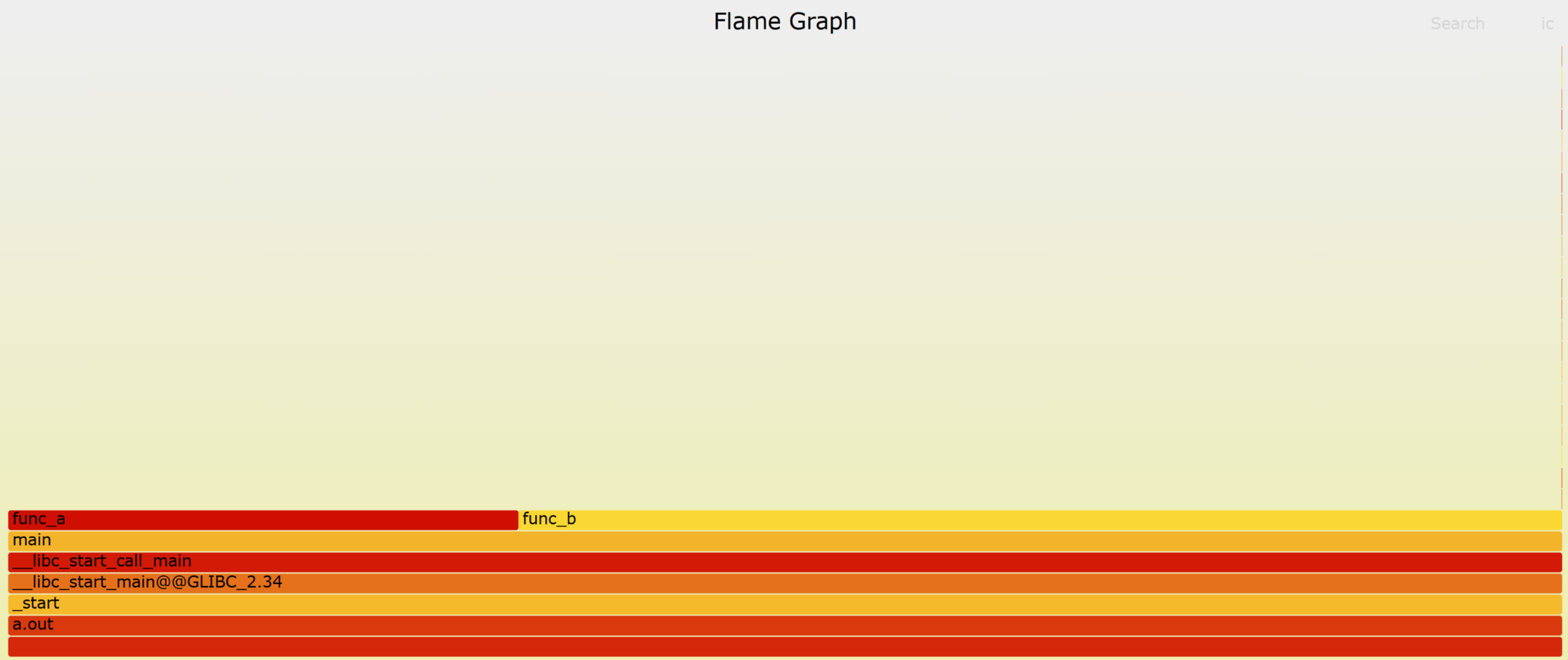
<Example of single sample ($perf script)>

```
Samples: 2K of event 'task-clock', Event count (approx.): 2920000000
  Overhead  Command  Shared Object      Symbol
-   67.12%  a.out    a.out              [.] func_b
     func_b
     main
     __libc_start_call_main
     __libc_start_main@@GLIBC_2.34
     _start
-   32.84%  a.out    a.out              [.] func_a
     func_a
     main
     __libc_start_call_main
     __libc_start_main@@GLIBC_2.34
     _start
```

<Statistical analysis result ($perf report)>

Sampling-Based Profilers (1/2)

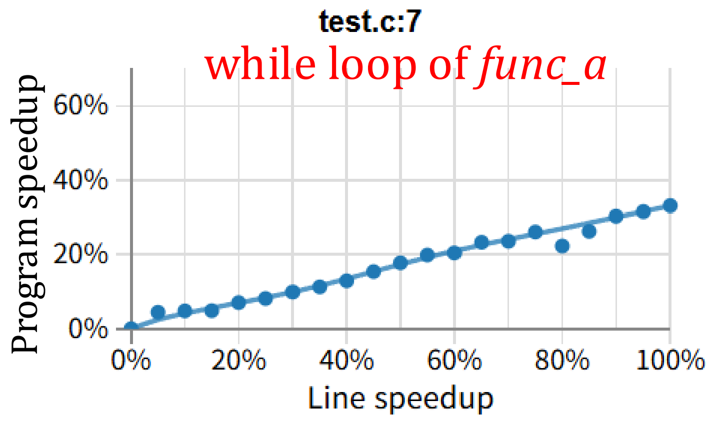
- FlameGraph [Brendan Gregg]
  - Callstack visualization of sampling results

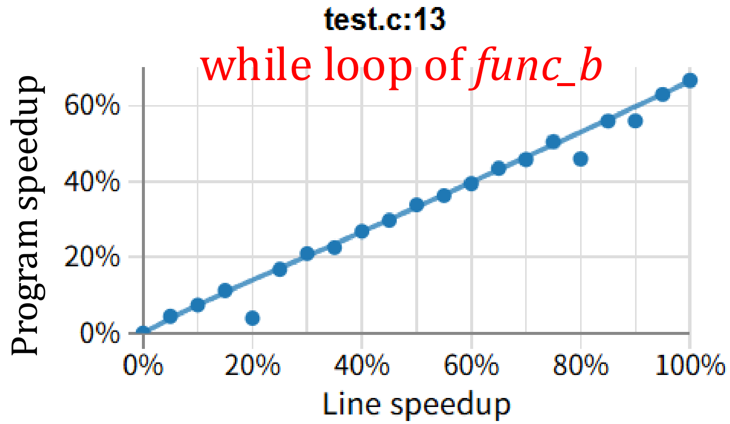


Sampling-Based Profilers (2/2)

- COZ [SOSP '15]
  - Predict the impact of optimizing <u>without actual optimization</u>

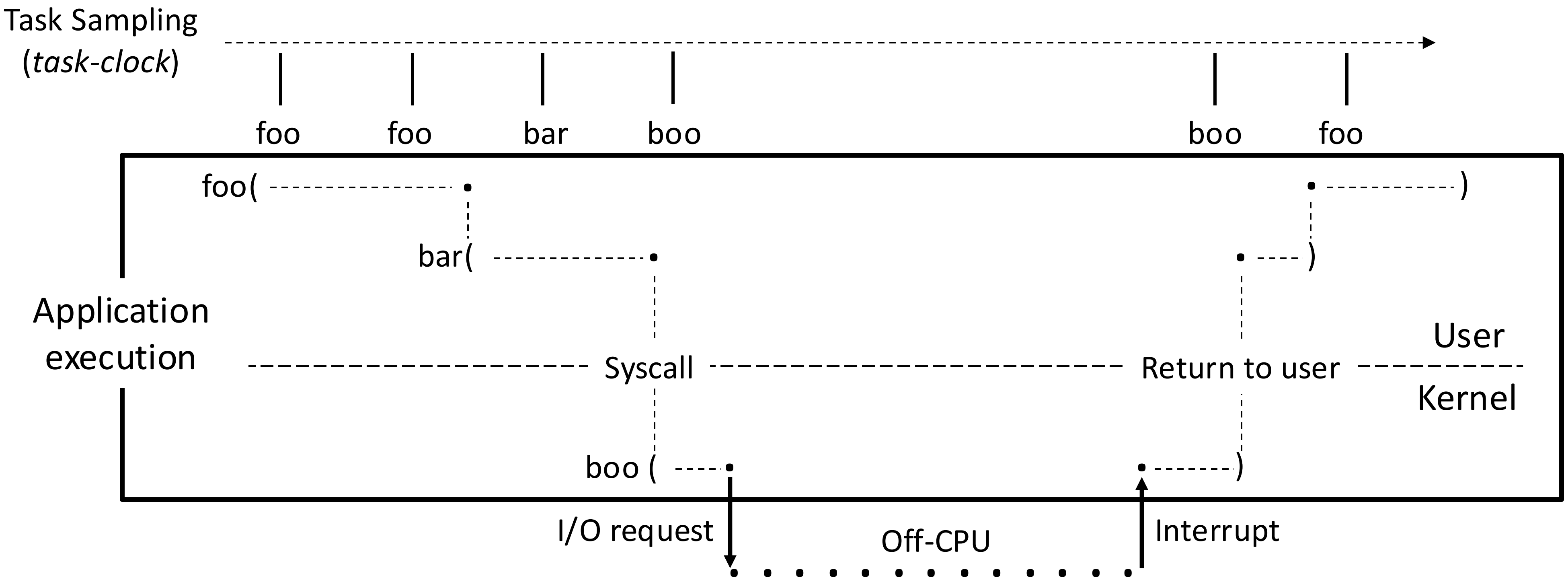
```
void func_a() {
   while (i < 20000000)       i++;
}
void func_b() {
   while (i < 40000000)       i++;
}

int main (int argc, char *argv[]) {
   func_a();
   func_b();

   return 0;
}
```





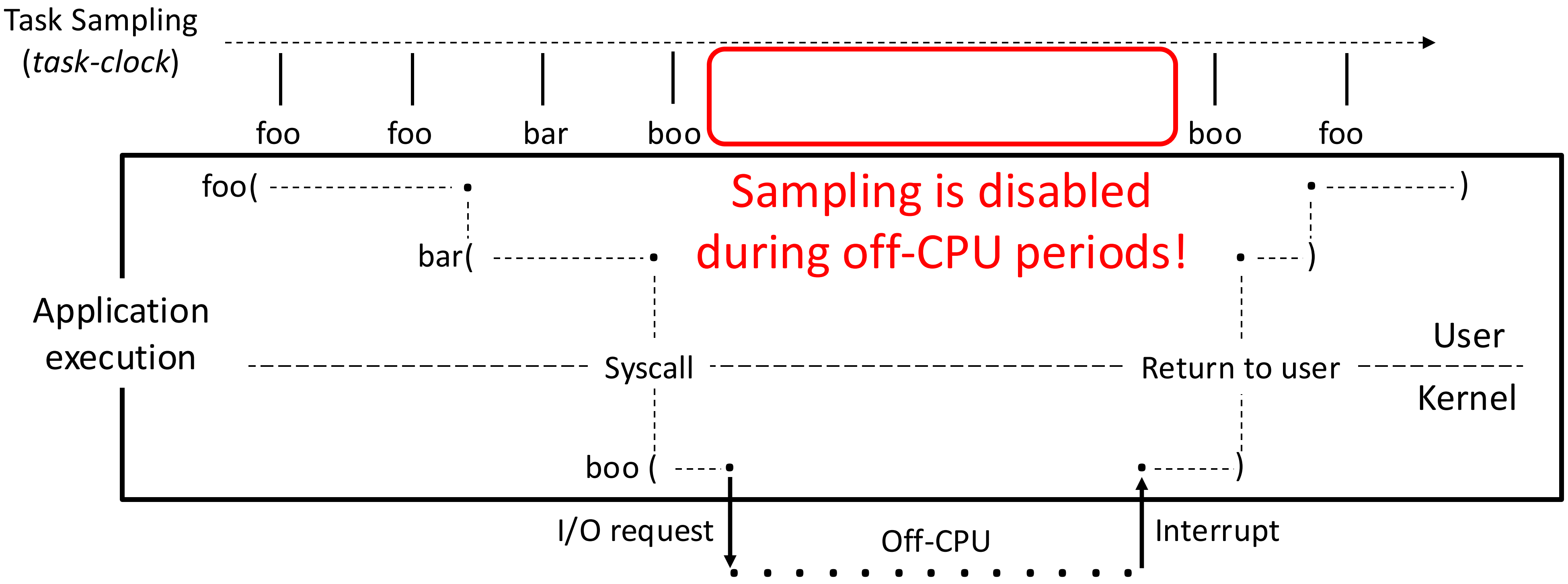
Research Question

- Is sampling effective for real-world applications?
  – Can sampling handle off-CPU events (e.g., blocking I/O, CPU scheduling, locks)?



Research Question

- Is sampling effective for real-world applications?
  - Can sampling handle off-CPU events (e.g., blocking I/O, CPU scheduling, locks)?

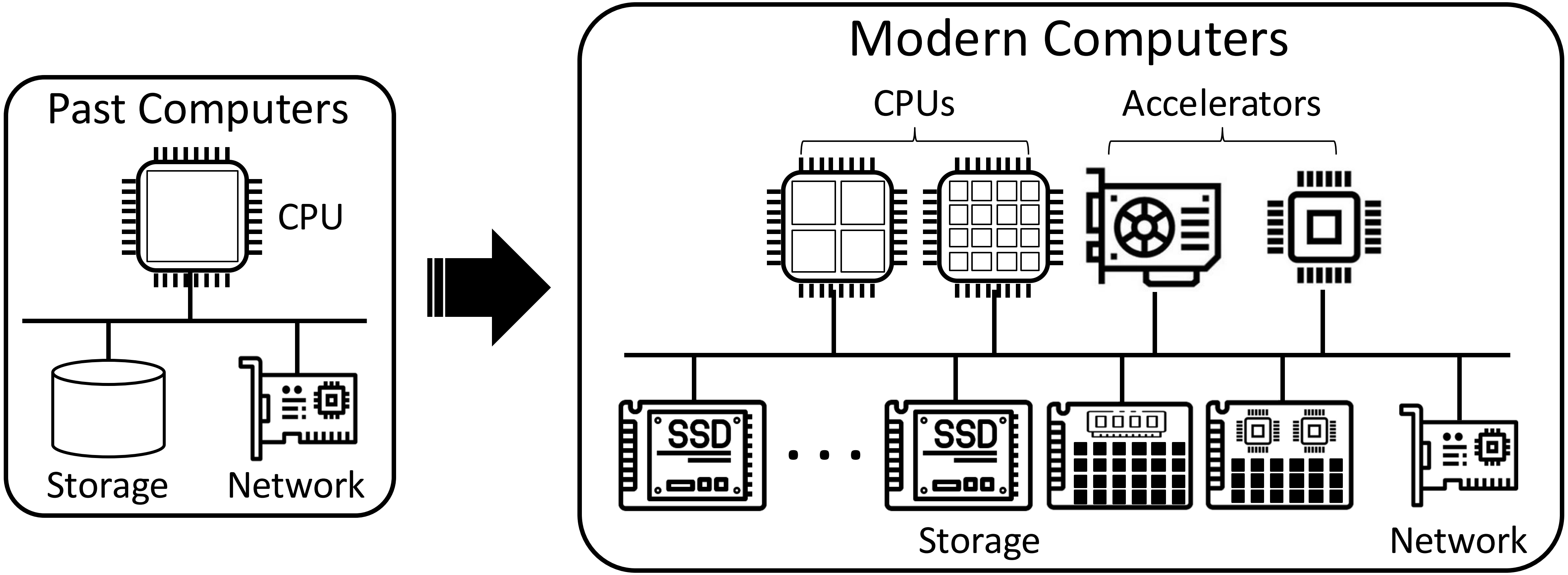


No! Sampling cannot collect off-CPU information

- Why the off-CPU analysis is important?

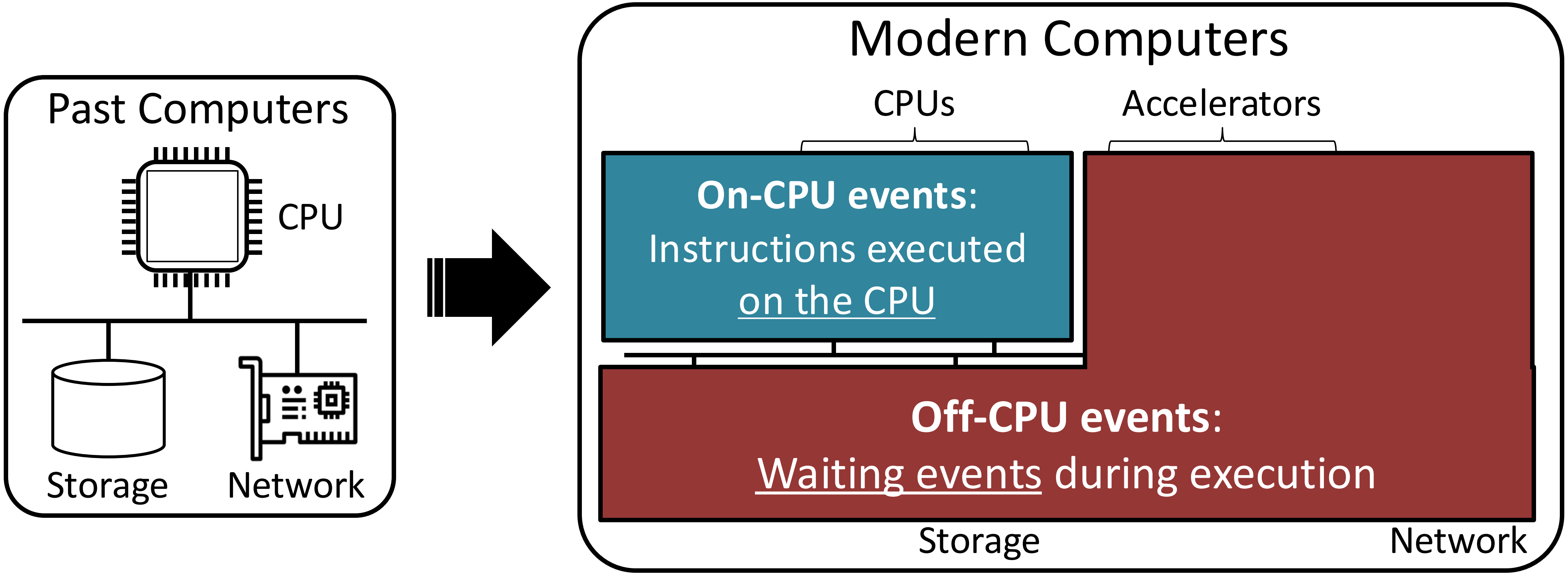
Trend of Computing Environments

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



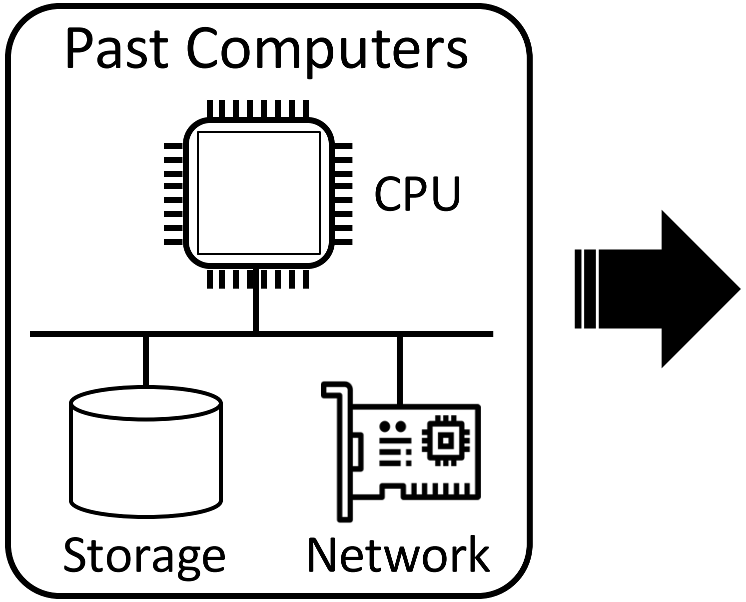
Trend of Computing Environments

- 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

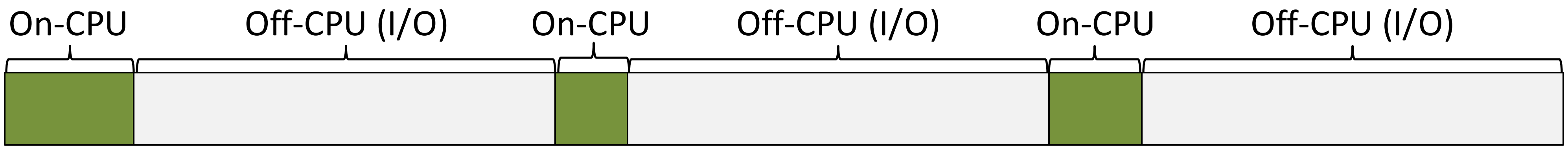
- Computing environments are becoming more complex and advanced
  - Events executed outside the CPU (i.e., off-CPU) have become more diverse



Where are Bottlenecks?

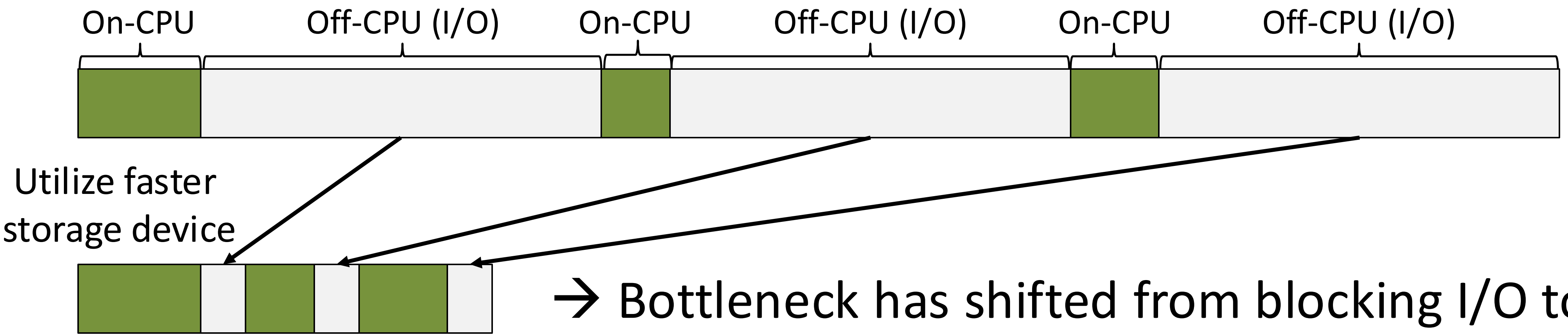
Bottlenecks of Modern Applications

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



Bottlenecks of Modern Applications

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

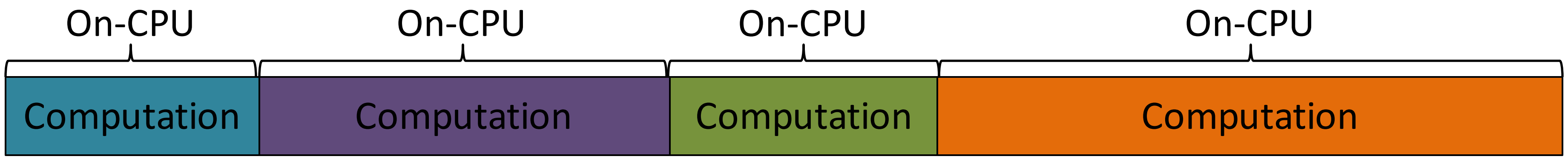


→ Bottleneck has shifted from blocking I/O to CPU

- "*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]

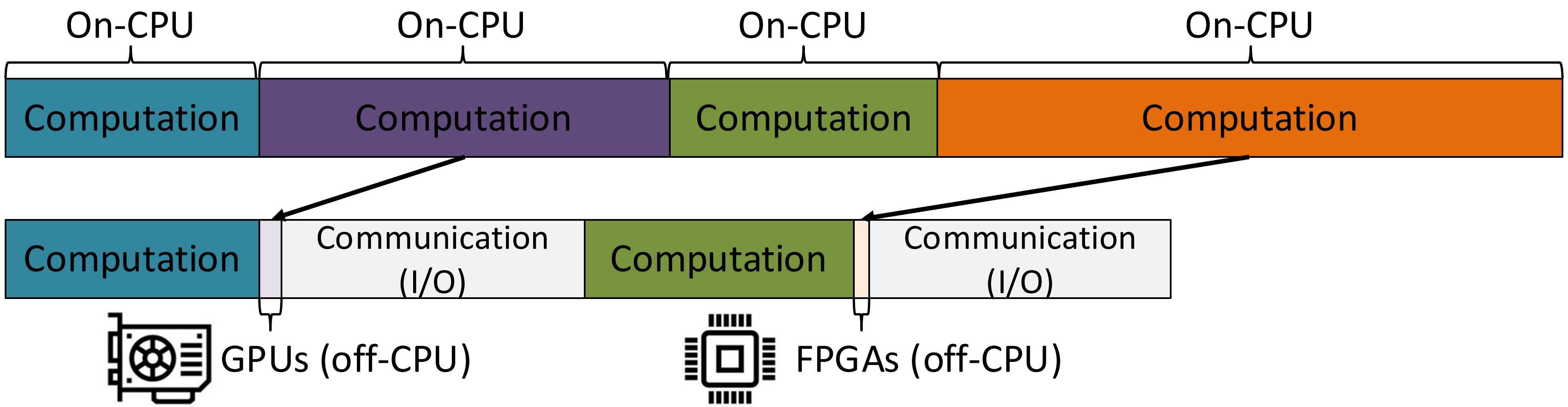
Bottlenecks of Modern Applications

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



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

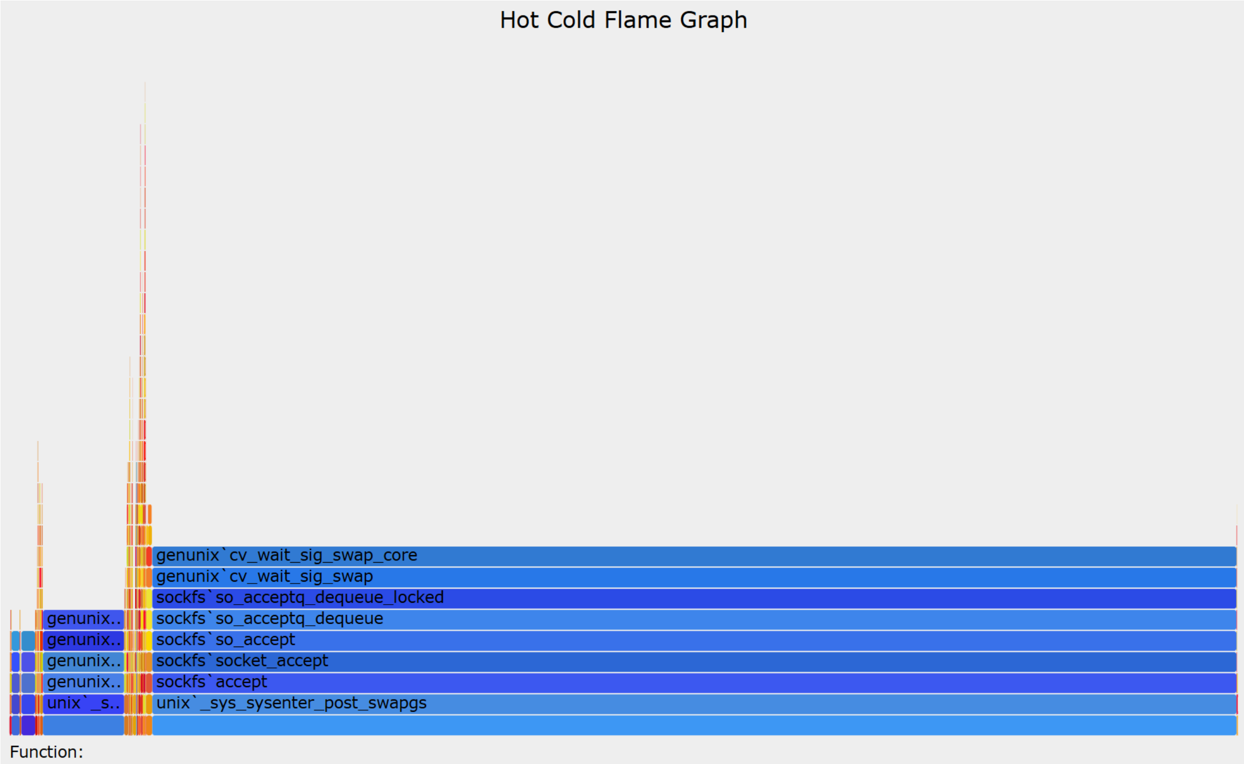


→ Bottleneck has shifted from CPU computation to I/O and communication

- "*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]

Off-CPU Analysis (1/2)

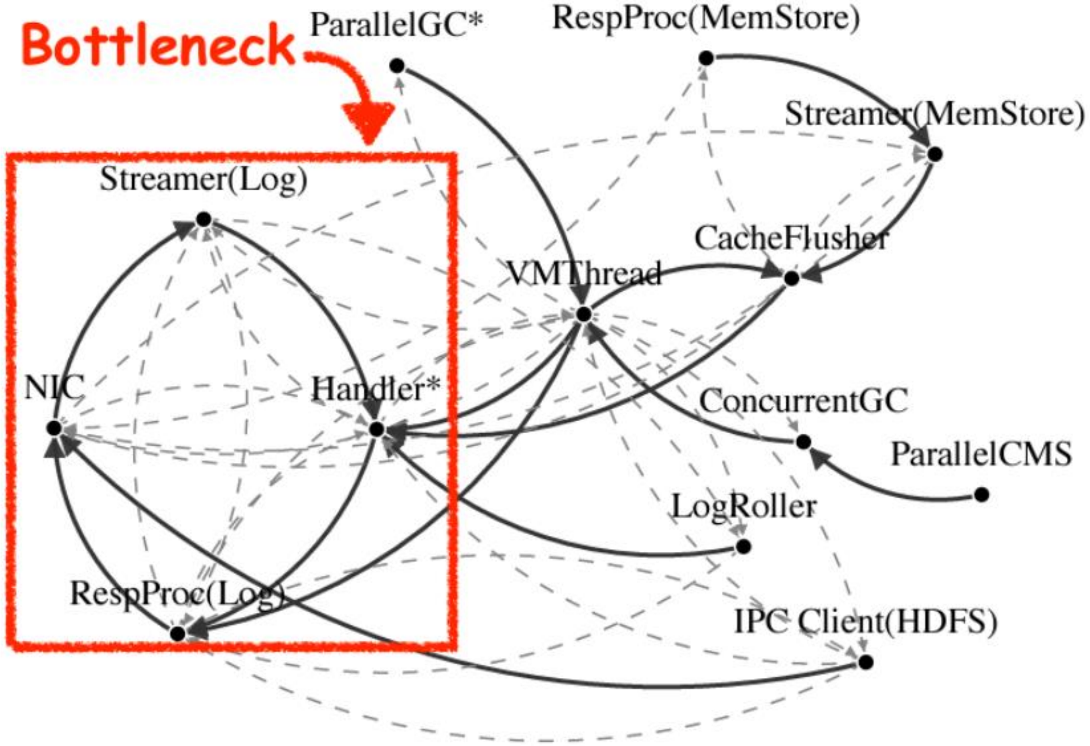
- Existing off-CPU analysis relies on tracing
  – Hot/Cold FlameGraph [Brendan Gregg]
    - Traces all blocking events (i.e., schedule-in/out) using Linux perf subsystem



<Example of hot/cold FlameGraph>

Off-CPU Analysis (2/2)

- Existing off-CPU analysis relies on tracing
  – wPerf [OSDI '18]
    - Traces all waiting events between threads with their dependency



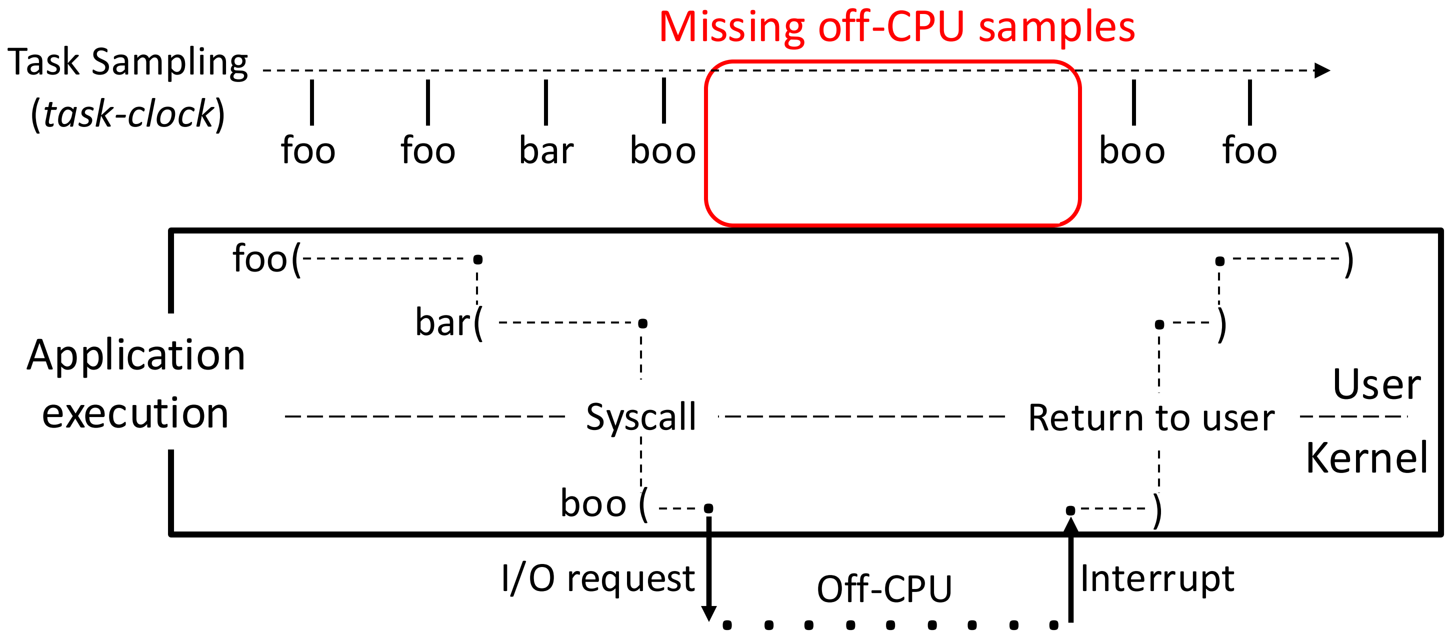
<Example of wPerf>

Limitations
#1) High overhead
- Frequent event tracing
#2) Lack of context information
- Missing code information

Our Approach: Blocked Samples

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

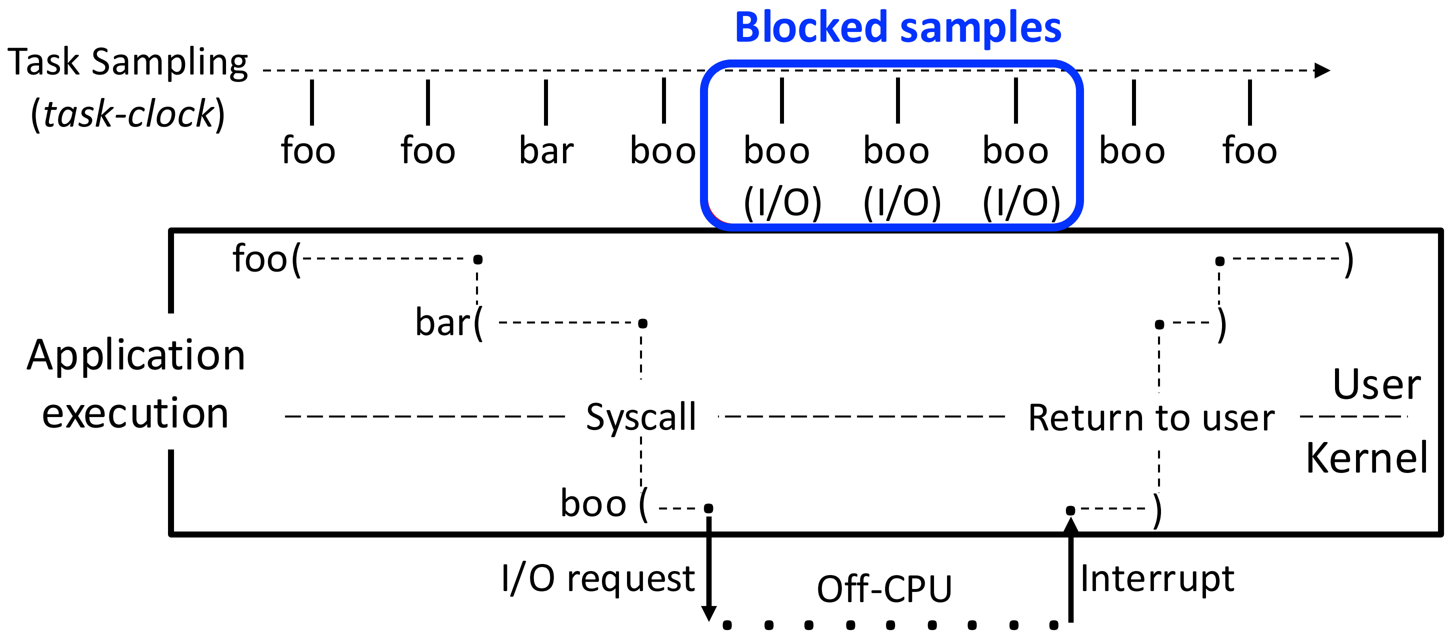
Blocked samples
(Linux perf subsystem)



Our Approach: Blocked Samples

- Goal: sampling on- and off-CPU events simultaneously
  - ***Blocked samples***: sampling technique for off-CPU events (*task-clock-plus*)

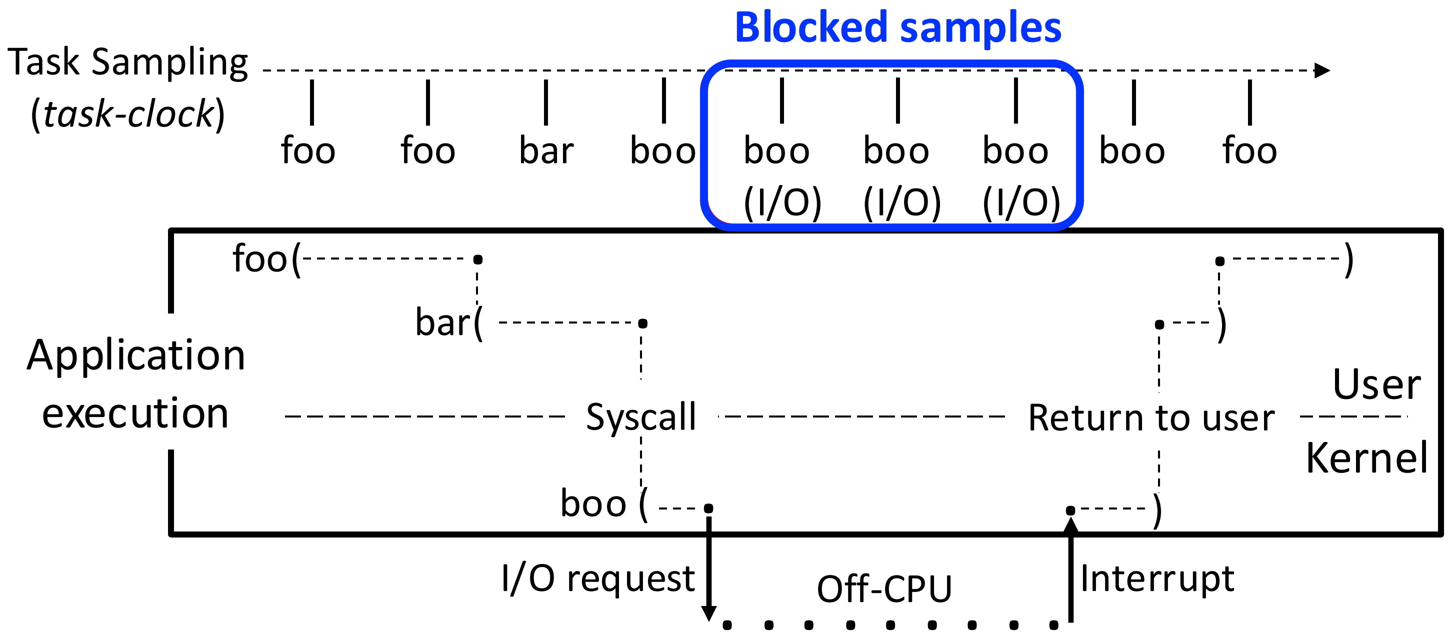
Blocked samples (Linux perf subsystem)



Our Approach: Blocked Samples

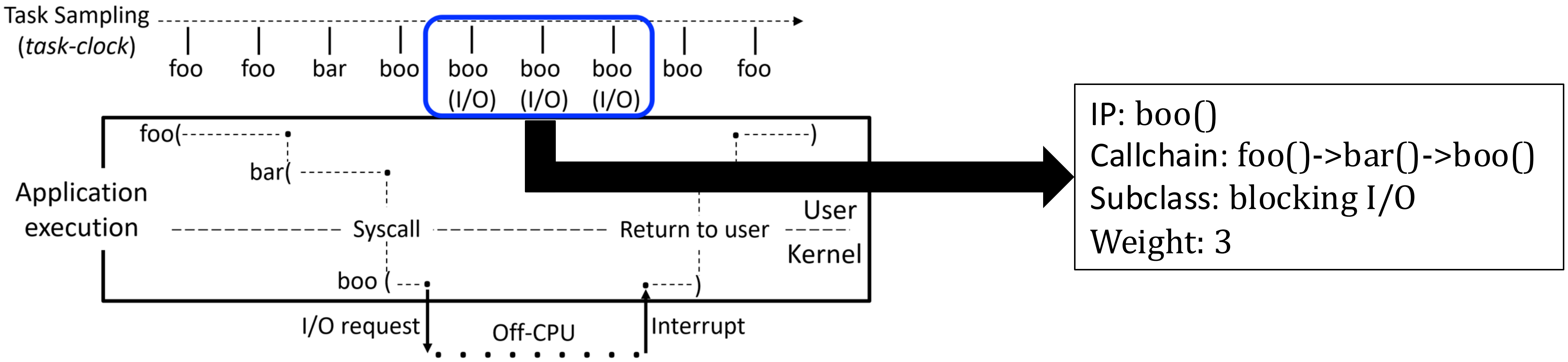
- Goal: sampling on- and off-CPU events simultaneously
  - ***Blocked samples***: sampling technique for off-CPU events (***task-clock-plus***)
  - Proposed profilers using blocked samples
    - ***bperf***: sampling-based statistical profiler on both on-/off-CPU events
    - ***BCOZ***: causal profiler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

Blocked samples (Linux perf subsystem)



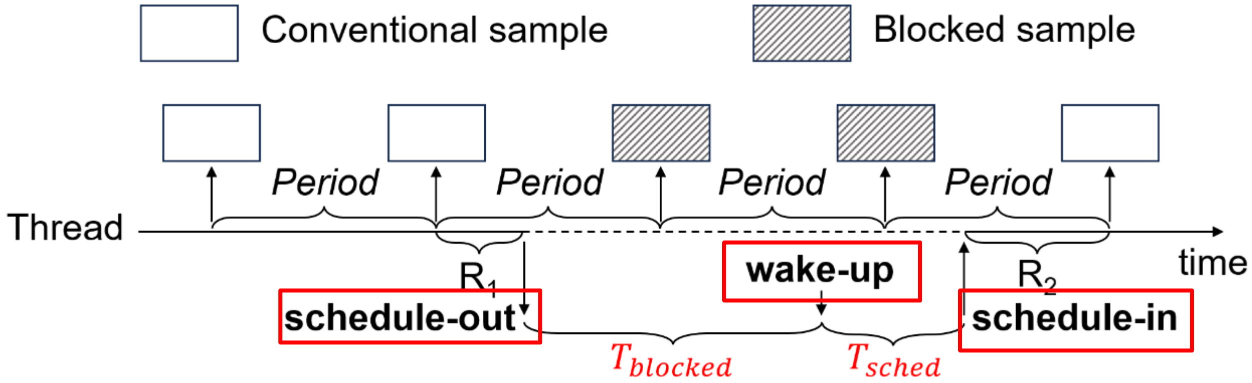
Blocked Samples (*task-clock-plus*)

- 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



task-clock-plus Implementation

- Extending task-clock event in the Linux perf subsystem



  - Hooks in scheduling-related operations
    - Sched-out (*prepare_task_switch→task_clock_event_del*)
      - Records timestamp, and off-CPU subclass
    - Wake-up (*try_to_wake_up*)
      - Records timestamp
    - Sched-in (*finish_task_switch→task_clock_event_add*)
      - 1) Calculate the length of blocking period
      - 2) Calculate the number of off-CPU samples to record
      - 3) (If exists) Record the off-CPU samples

* Samples are recorded only if sampling points are overlap with off-CPU period
→ Differ from tracing

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

- Extension of Linux perf tool to support blocked samples
  - Sample accounting
  - Result reporting
    - [I]: blocking I/O, [L]: synchronization, [S]: CPU scheduling, [B]: others
    - Both the <u>last user-level IP and last kernel-level IP</u> 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
```

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

- Extension of Linux perf tool to support blocked samples
  - Sample accounting
  - Result reporting
    - [I]: blocking I/O, [L]: synchronization, [S]: CPU scheduling, [B]: others
    - Both the <u>last user-level IP and last kernel-level IP</u> 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

Toy Program with Mixed of On-/Off-CPU Events

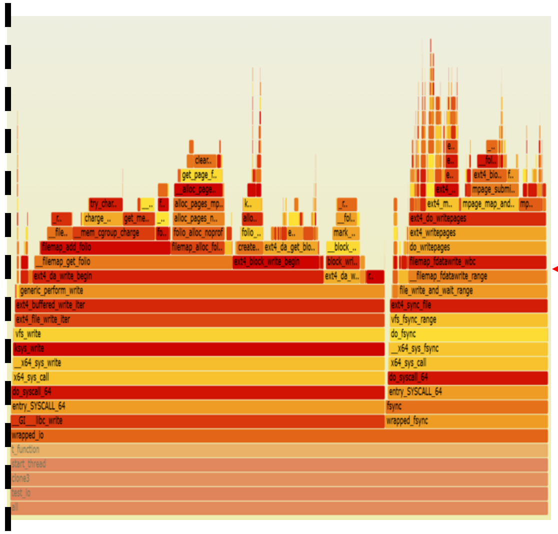
<Toy program>

```
while (i < 300000) {
    write();
    fsync();
}
```

Samples: 32K of event 'task-clock', Event count (approx.): 32620000000

	Overhead	Command	Shared Object	Symbol
+	10.70%	test_io	[kernel.kallsyms]	[k] _raw_spin_unlock_irqrestore
+	6.52%	test_io	[kernel.kallsyms]	[k] _raw_spin_unlock_irq
+	5.37%	test_io	[kernel.kallsyms]	[k] try_charge_memcg
+	3.72%	test_io	[kernel.kallsyms]	[k] __rcu_read_unlock
+	3.70%	test_io	[kernel.kallsyms]	[k] clear_page_erms
+	2.77%	test_io	[kernel.kallsyms]	[k] rep_movs_alternative
+	2.54%	test_io	[kernel.kallsyms]	[k] get_mem_cgroup_from_mm

<w/o blocked samples>



Missing samples

→ 100% kernel I/O stack

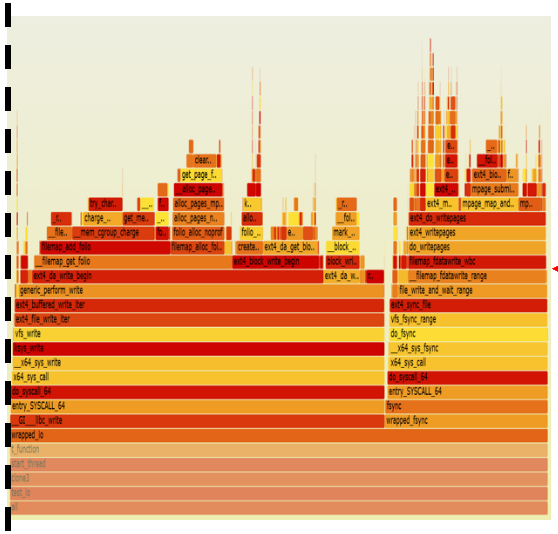
Toy Program with Mixed of On-/Off-CPU Events

<Toy program>

```
while (i < 300000) {
    write();
    fsync();
}
```

```
Samples: 32K of event 'task-clock', Event count (approx.): 32620000000
  Overhead  Command  Shared Object      Symbol
+   10.70%  test_io  [kernel.kallsyms]  [k] _raw_spin_unlock_irqrestore
+    6.52%  test_io  [kernel.kallsyms]  [k] _raw_spin_unlock_irq
+    5.37%  test_io  [kernel.kallsyms]  [k] try_charge_memcg
+    3.72%  test_io  [kernel.kallsyms]  [k] __rcu_read_unlock
+    3.70%  test_io  [kernel.kallsyms]  [k] clear_page_erms
+    2.77%  test_io  [kernel.kallsyms]  [k] rep_movs_alternative
+    2.54%  test_io  [kernel.kallsyms]  [k] get_mem_cgroup_from_mm
```

<w/o blocked samples>



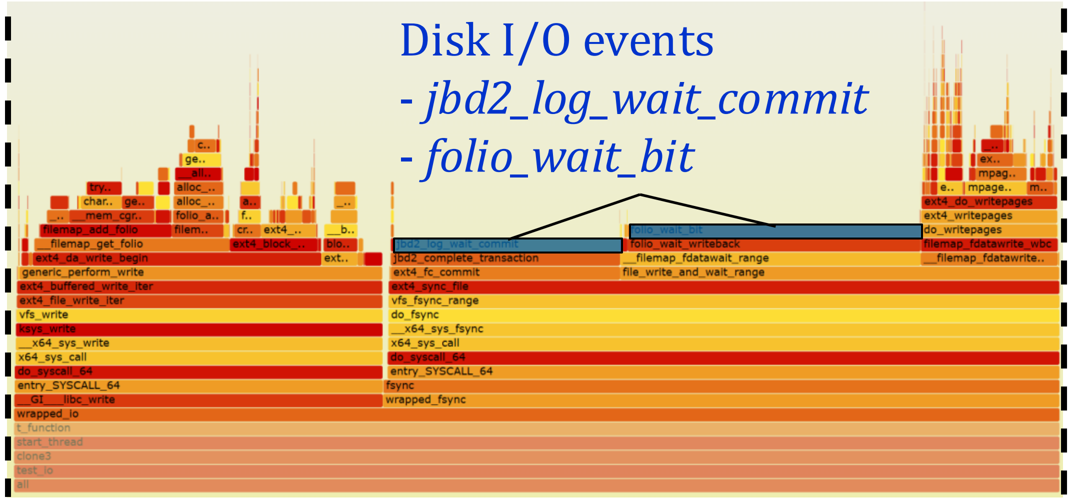
Missing samples

→ 100% kernel I/O stack

CPU: 55%, IO wait: 25.4%, Idle(jbd2 wait): 19.7%

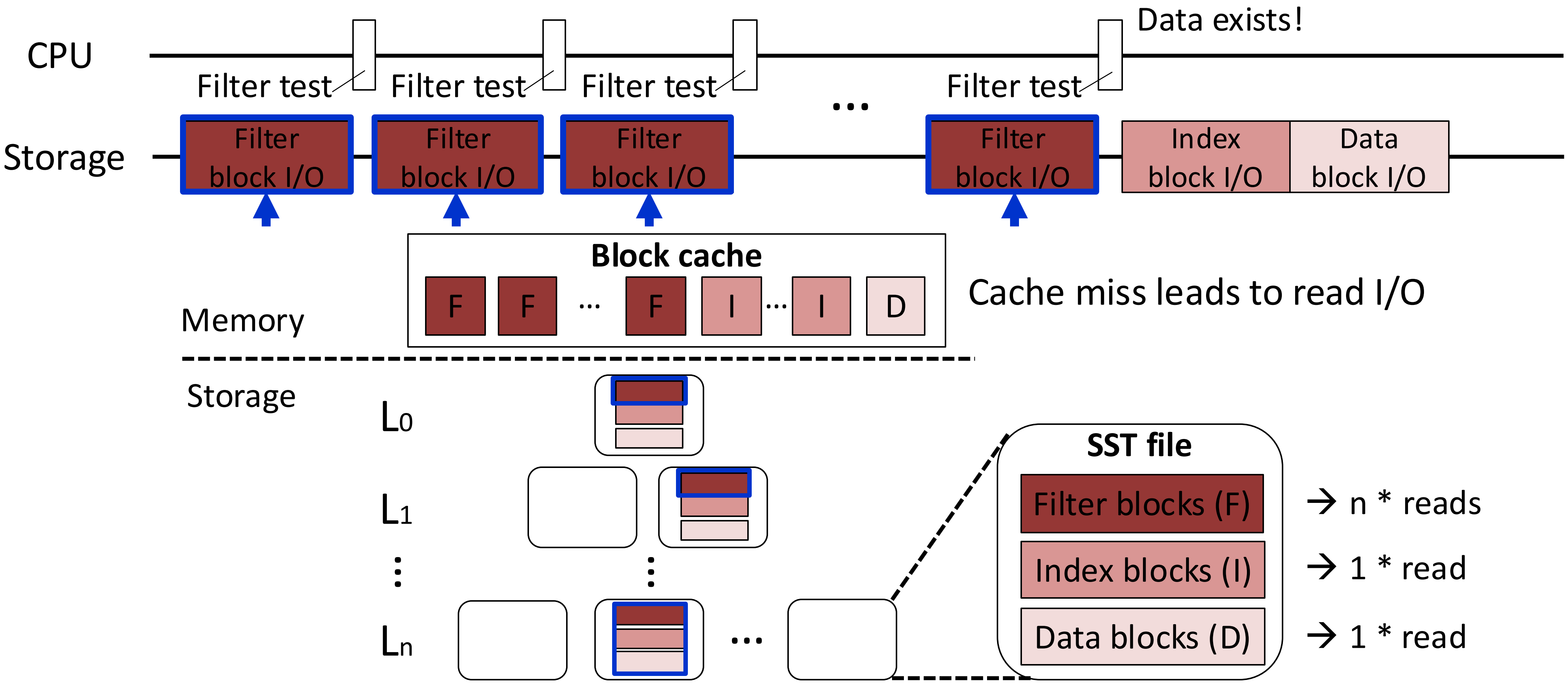
```
Samples: 56K of event 'task-clock-plus', Event count (approx.): 56140000000
  Overhead  Command  Shared Object      Symbol
+   25.40%  test_io  [kernel.kallsyms]  [I] folio_wait_bit
+   19.66%  test_io  [kernel.kallsyms]  [B] jbd2_log_wait_commit
+    5.99%  test_io  [kernel.kallsyms]  [k] _raw_spin_unlock_irqrestore
+    3.44%  test_io  [kernel.kallsyms]  [k] _raw_spin_unlock_irq
+    3.05%  test_io  [kernel.kallsyms]  [k] try_charge_memcg
+    2.21%  test_io  [kernel.kallsyms]  [k] __rcu_read_unlock
+    1.92%  test_io  [kernel.kallsyms]  [k] clear_page_erms
+    1.53%  test_io  [kernel.kallsyms]  [k] rep_movs_alternative
+    1.39%  test_io  [kernel.kallsyms]  [k] get_mem_cgroup_from_mm
```

<bperf>



Case Study – 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 – 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)

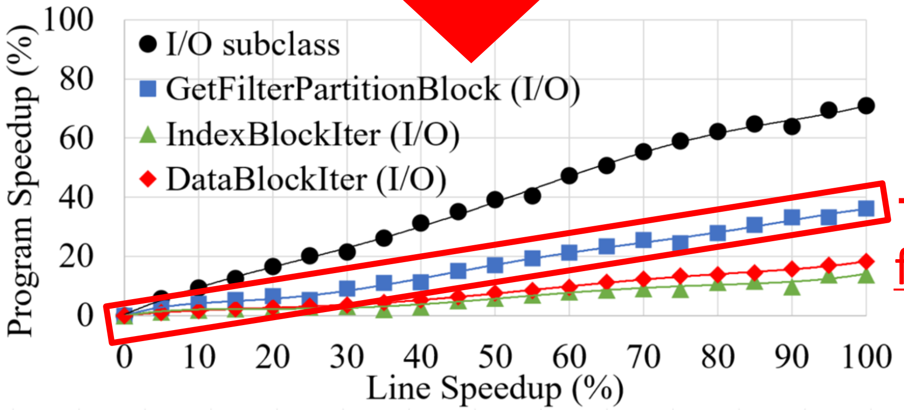
```
Samples: 1M of event 'task-clock', Event count (approx.): 1074412000000
  Overhead  Command         Shared Object       Symbol
-   85.33%  db_bench_vanill  libpthread-2.30.so  [I] __libc_pread64
   - __libc_pread64
      - rocksdb::PosixRandomAccessFile::Read
        rocksdb::RandomAccessFileReader::Read
      - 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

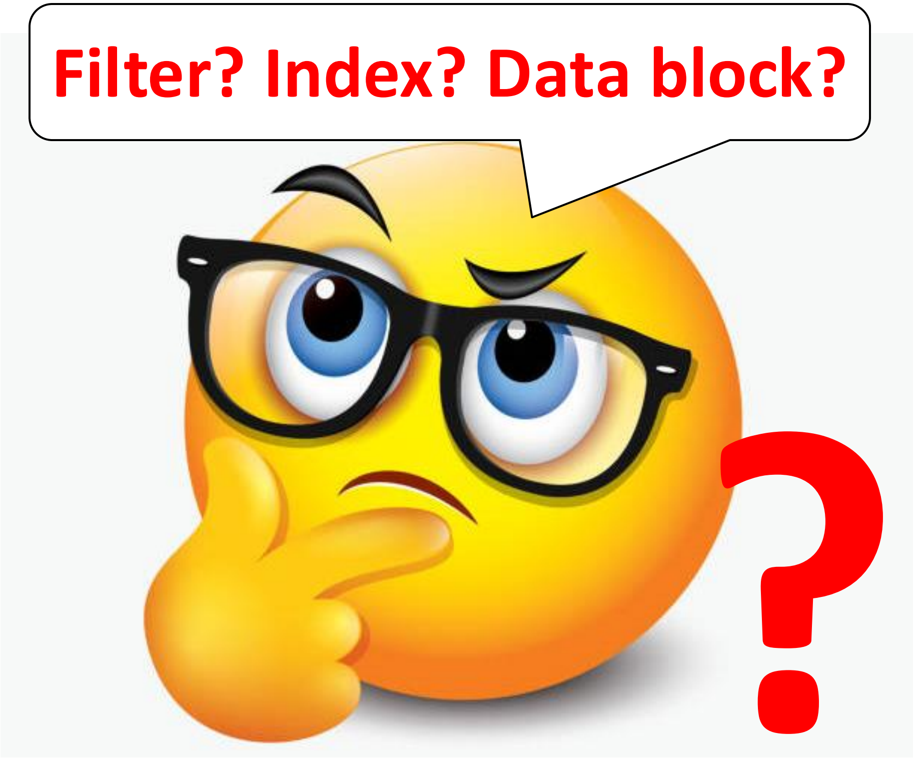
<bperf>

Causality analysis



<BCOZ>

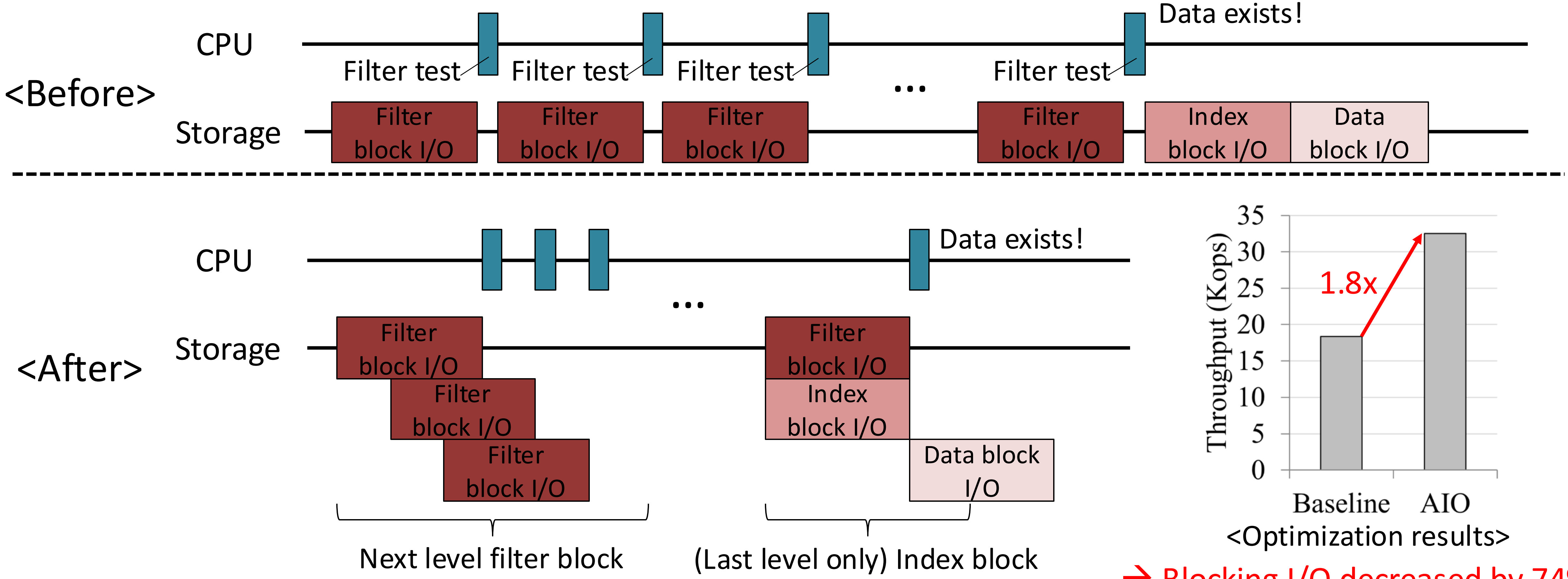
→Optimizing disk I/O of filter block is most important!



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

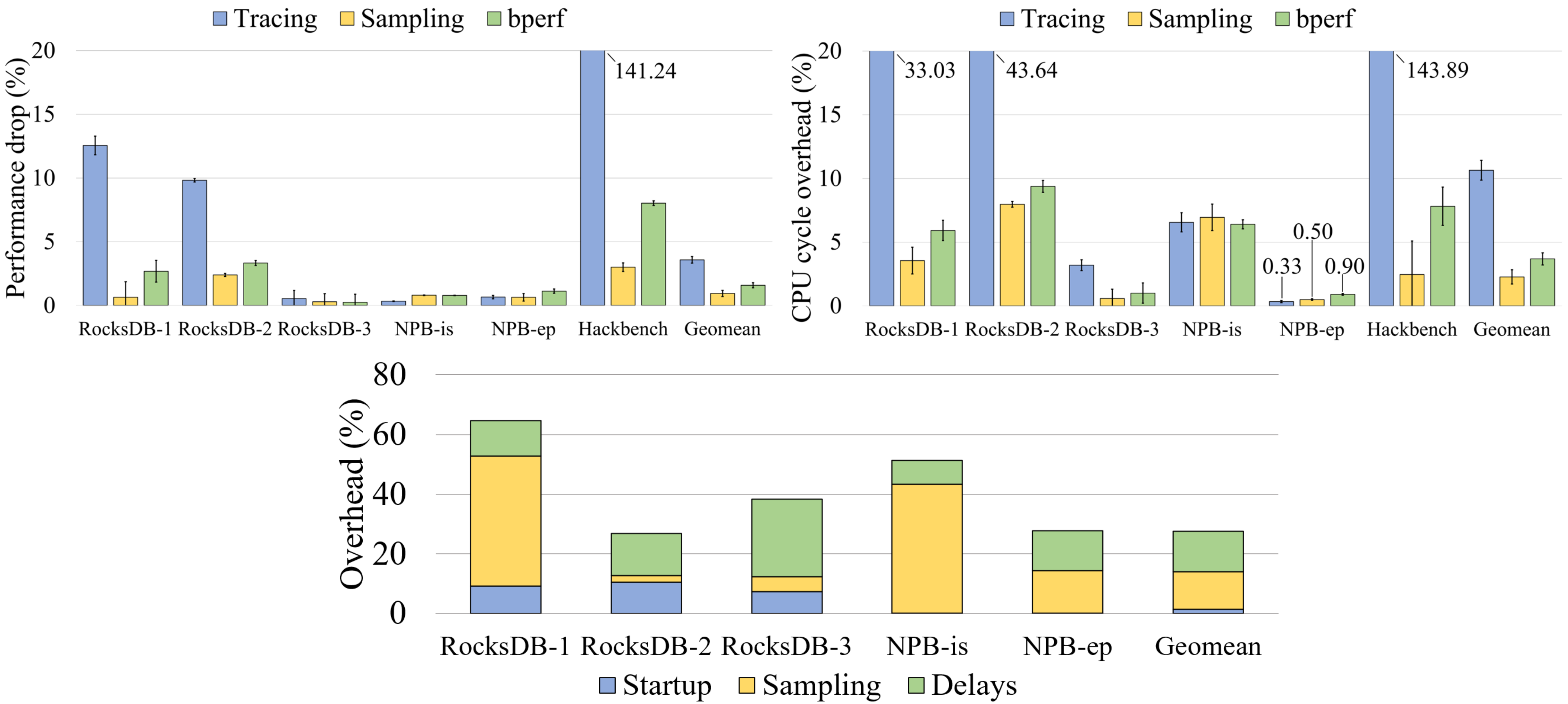
Case Study – RocksDB (Block Read Operation)

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



→ Blocking I/O decreased by 74%

Profiling Overhead



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 is available at:

https://github.com/s3yonsei/blocked_samples

https://github.com/s3yonsei/linux-blocked_samples

Thank you!

Credit:
Minwoo Ahn, Jeongmin Han, Youngjin Kwon, Jinkyu Jeong,
"Identifying On-/Off-CPU Bottlenecks Together with Blocked Samples,"
OSDI 2024

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Credit

- **Minwoo Ahn, Jeongmin Han, Youngjin Kwon, Jinkyu Jeong, "Identifying On-/Off-CPU Bottlenecks Together with Blocked Samples," OSDI 2024**

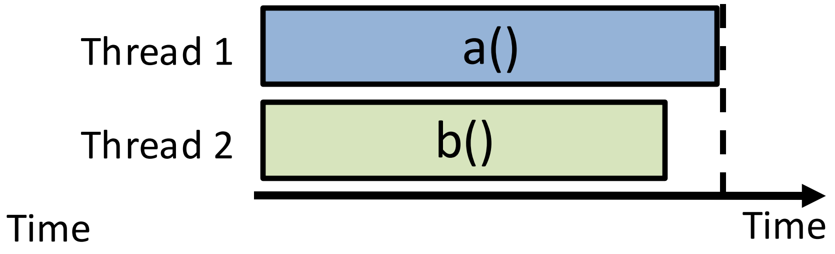
- Most slides are from the OSDI'24 presentation slides

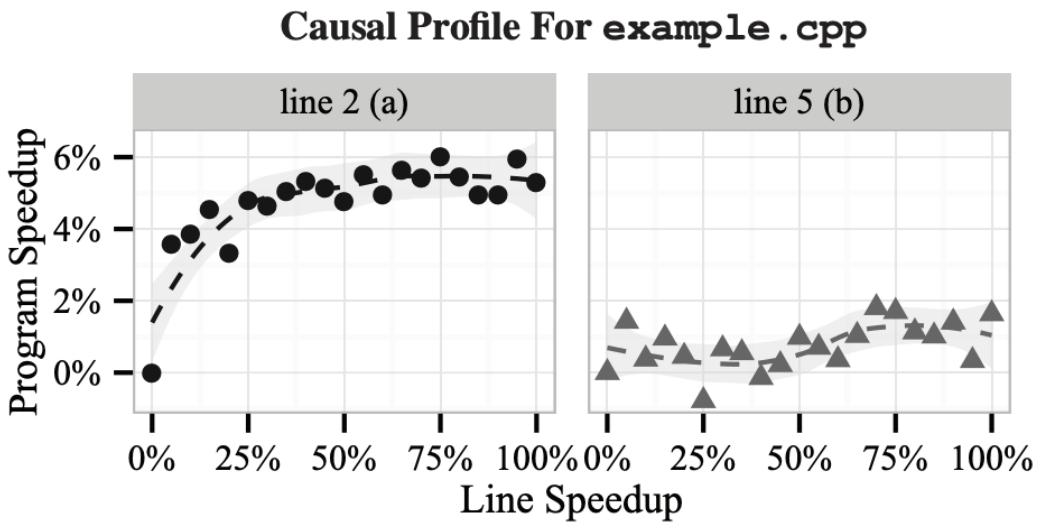
COZ (SOSP '15)

- COZ: Finding Code that Counts with Causal Profiling, SOSP '15
  - Charlie Curtsinger, Emery D. Berger

example.cpp

```
1  void a() { // ~6.7 seconds
2    for(volatile size_t x=0; x<2000000000; x++) {}
3  }
4  void b() { // ~6.4 seconds
5    for(volatile size_t y=0; y<1900000000; y++) {}
6  }
7  int main() {
8    // Spawn both threads and wait for them.
9    thread a_thread(a), b_thread(b);
10   a_thread.join(); b_thread.join();
11 }
```

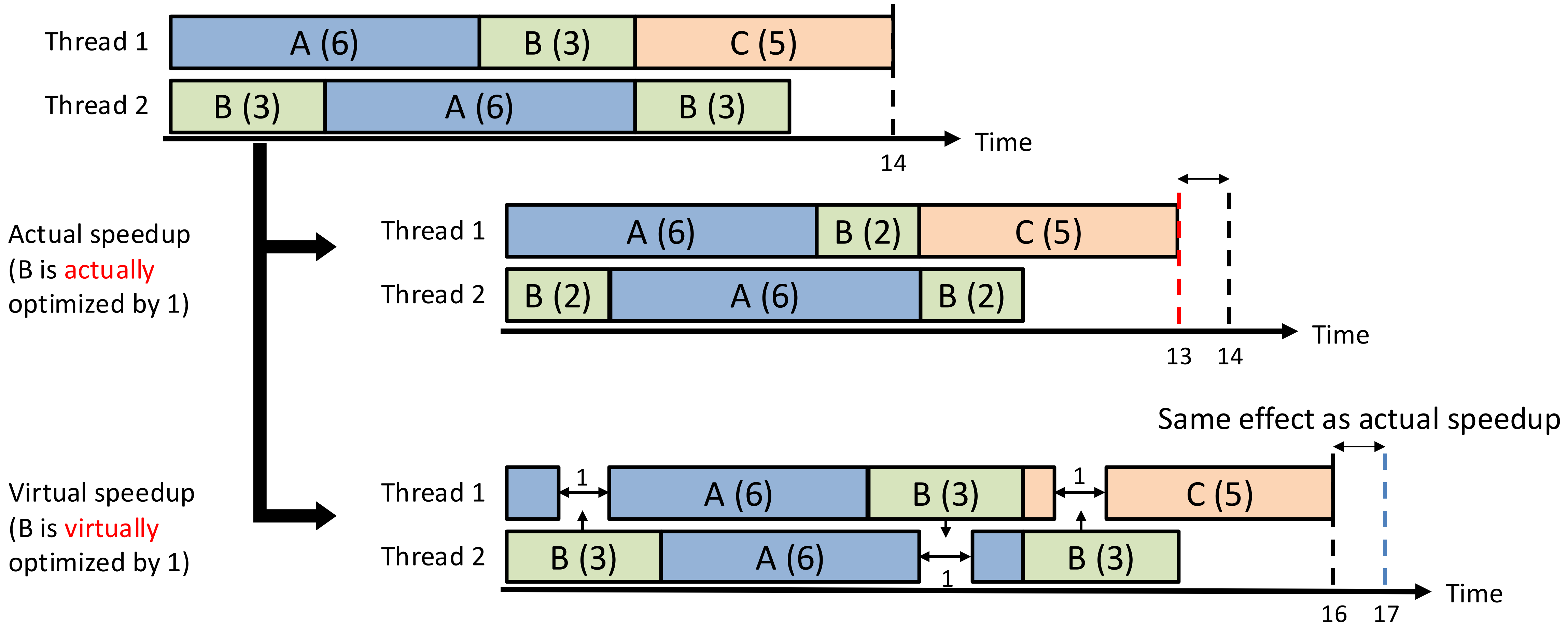




(b) Causal profile for example.cpp

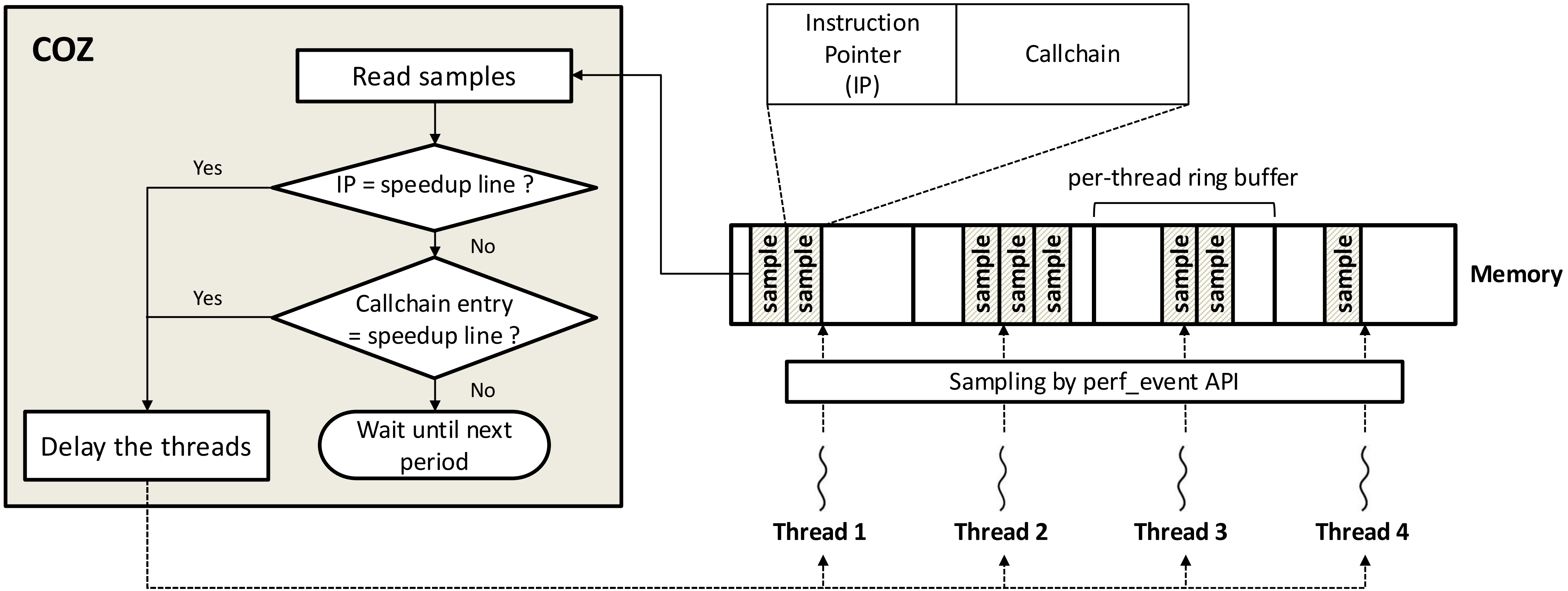
COZ (SOSP '15)

- Virtual speedup
  - Predict speedup of functions without actually speeding up code lines



COZ (SOSP '15)

- COZ is causal profiler using the virtual speedup technique
  - perf sampling + batch processing + thread sleeping and synchronization

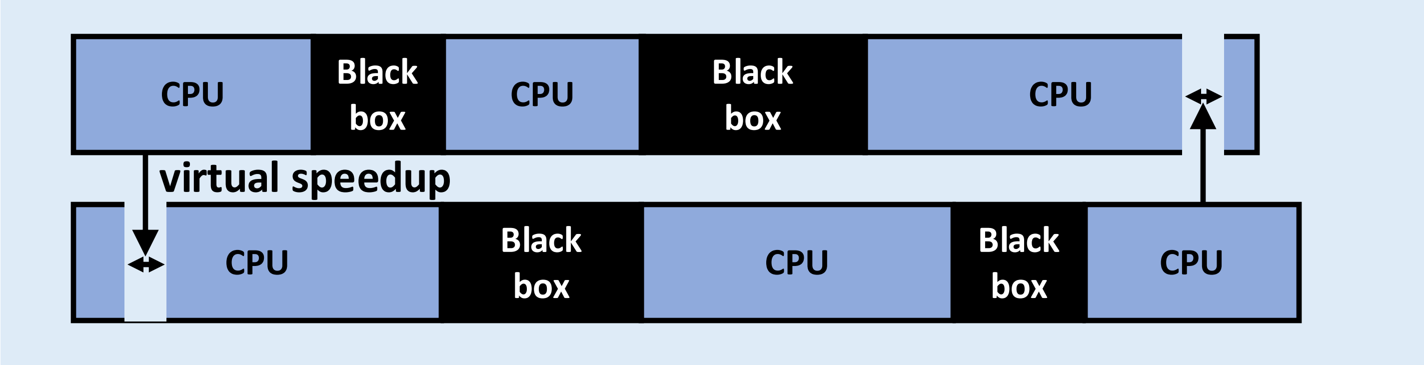


Research Question

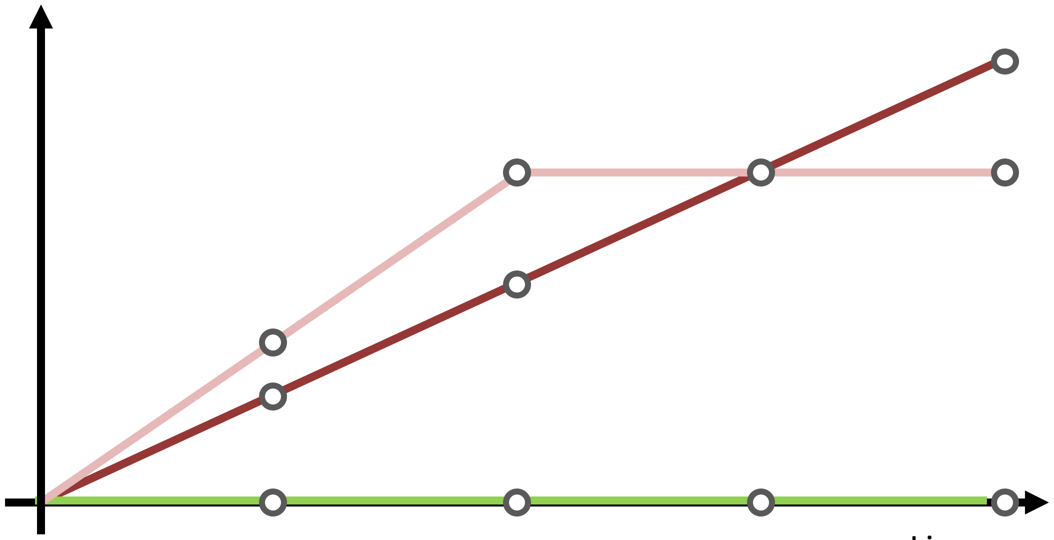
- What if virtual speedup can be applied to I/O events
  – COZ has profiled on-CPU events only
- How to make COZ apply the virtual speedup idea to I/O events (or off-CPU events)
  – E.g., disk I/Os

BCOZ: Causal Profiler for both On-/Off-CPU Events

- Virtual speedup the off-CPU events by ***blocked samples***
  - Shows potential speedup when off-CPU events are optimized
    - Locks, I/O, scheduling delay, etc.



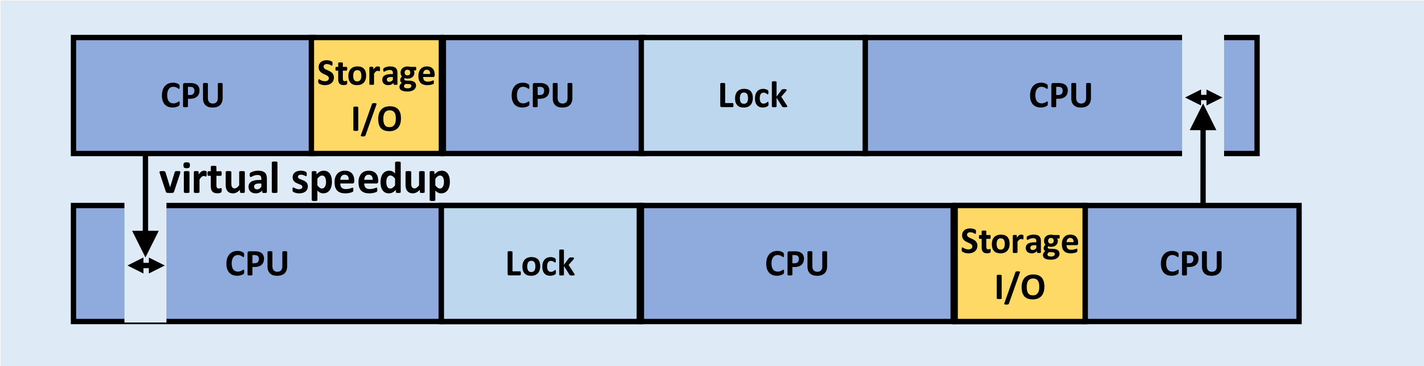
<Virtual speedup of on-CPU events (COZ)>



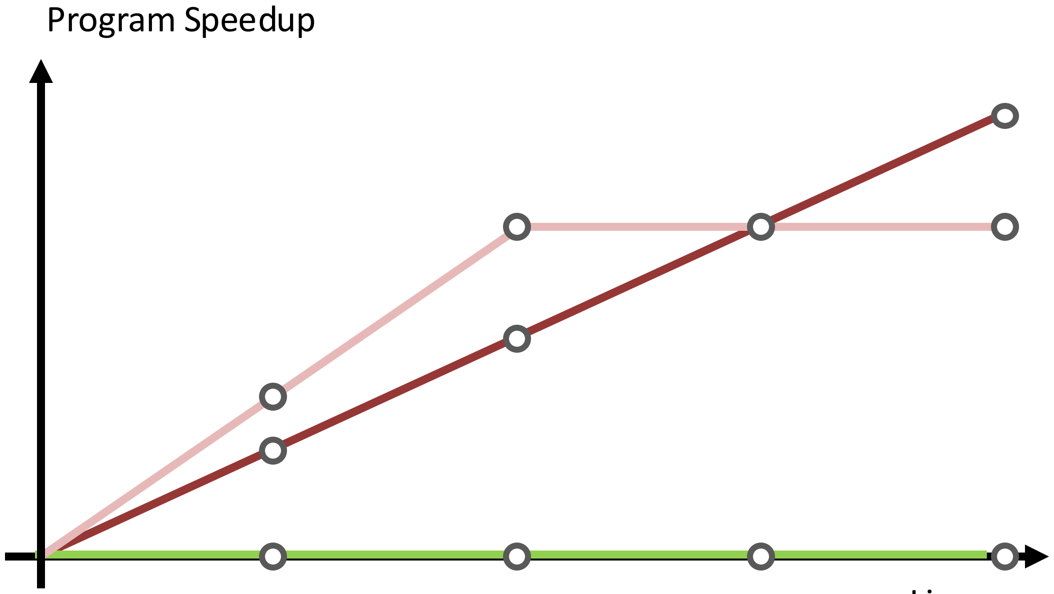
<Potential speedup result>

BCOZ: Causal Profiler for both On-/Off-CPU Events

- Virtual speedup the off-CPU events by ***blocked samples***
  - – Shows potential speedup when off-CPU events are optimized
    - Locks, I/O, scheduling delay, etc.



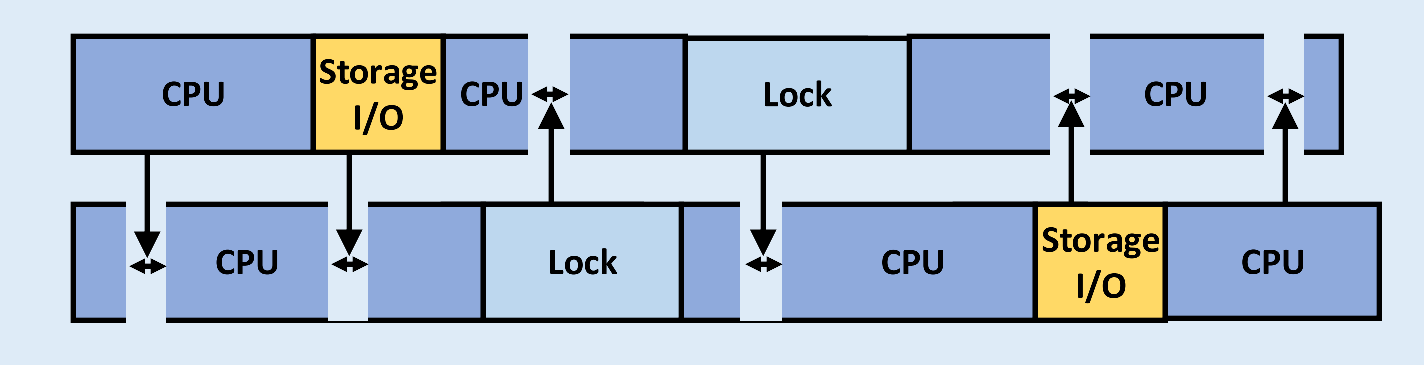
<Virtual speedup of both on-/off-CPU events (BCOZ)>



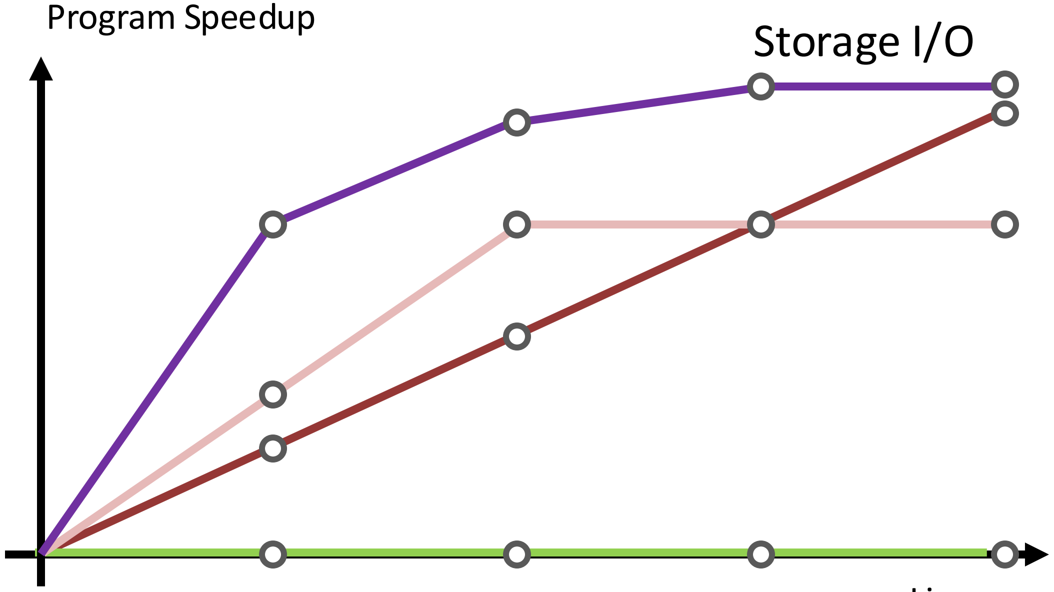
<Potential speedup result>

BCOZ: Causal Profiler for both On-/Off-CPU Events

- Virtual speedup the off-CPU events by ***blocked samples***
  – Shows potential speedup when off-CPU events are optimized
    - Locks, I/O, scheduling delay, etc.



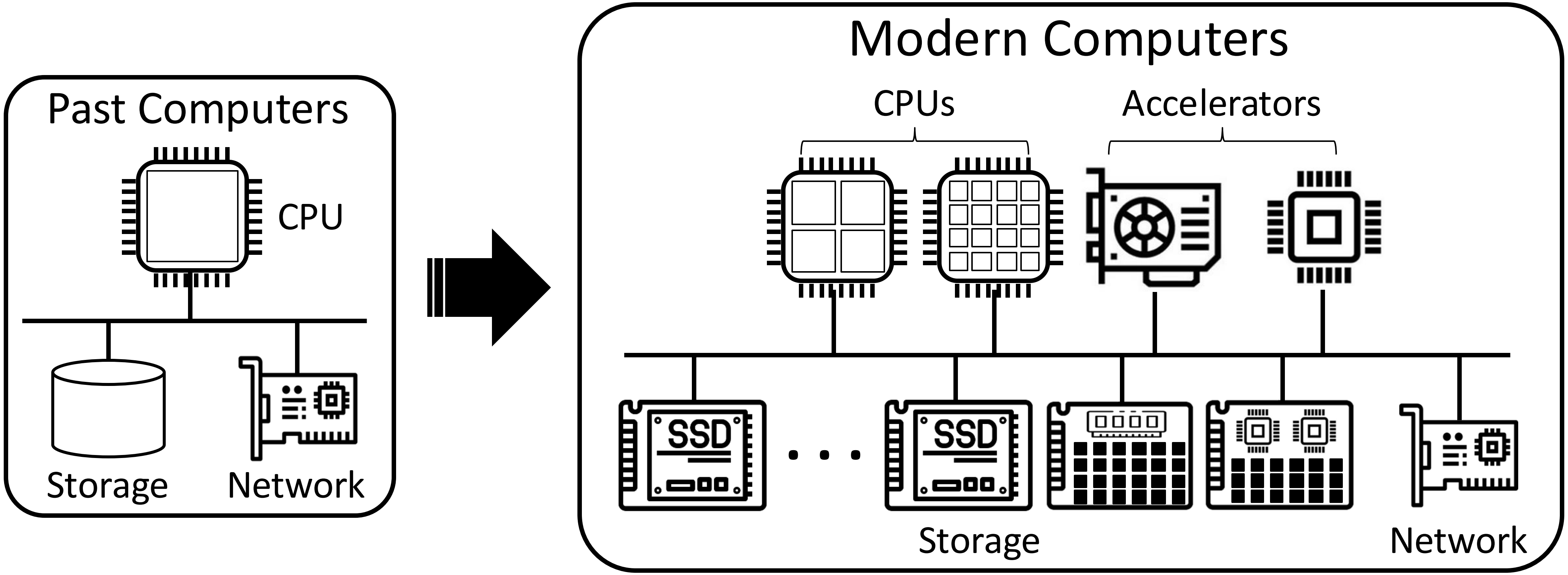
<Virtual speedup of both on-/off-CPU events (BCOZ)>



<Potential speedup result>

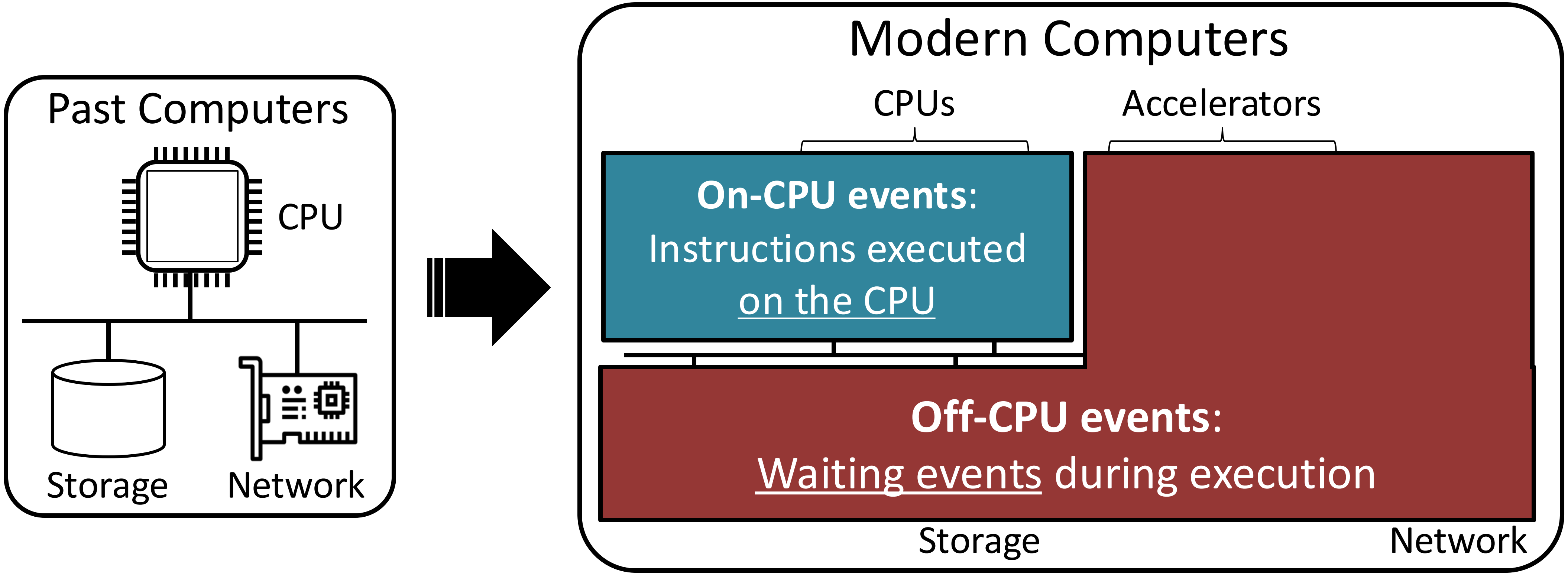
Trend of Computing Environments

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



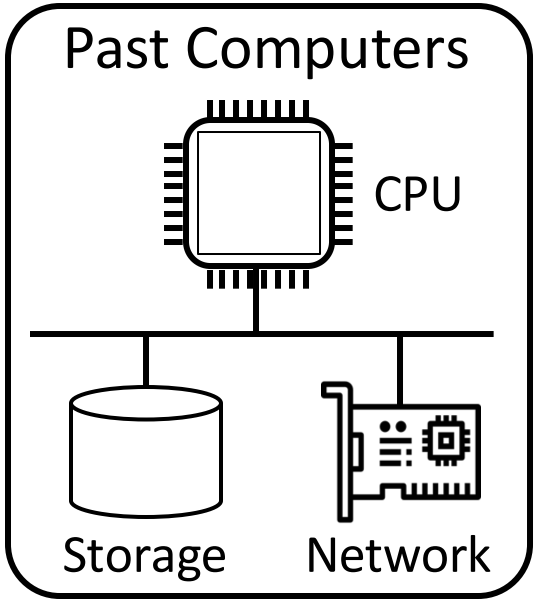
Trend of Computing Environments

- 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

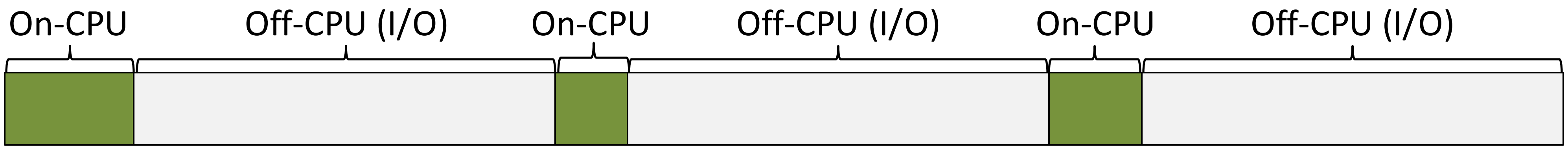
- Computing environments are becoming more complex and advanced
  - Events executed outside the CPU (i.e., <u>off-CPU</u>) have become more diverse



Where are Bottlenecks?

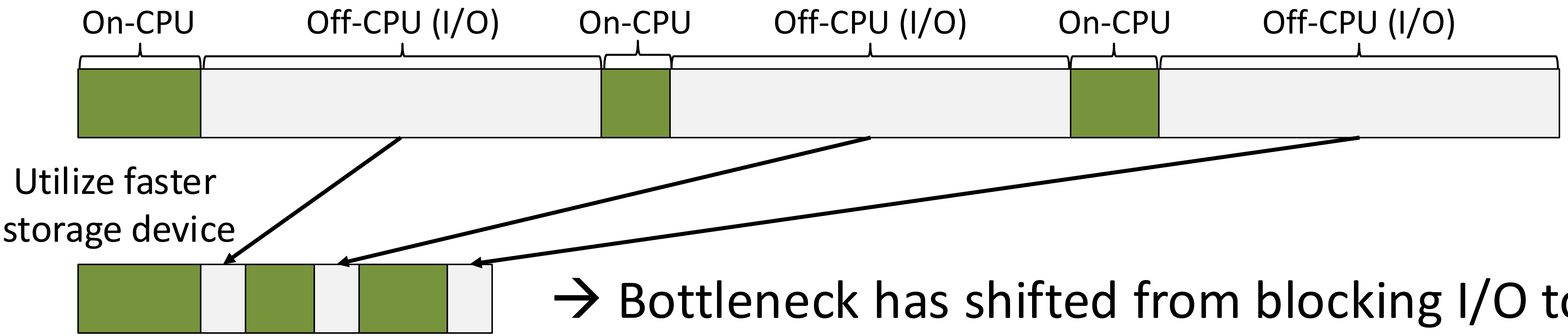
Bottlenecks of Modern Applications

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



Bottlenecks of Modern Applications

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

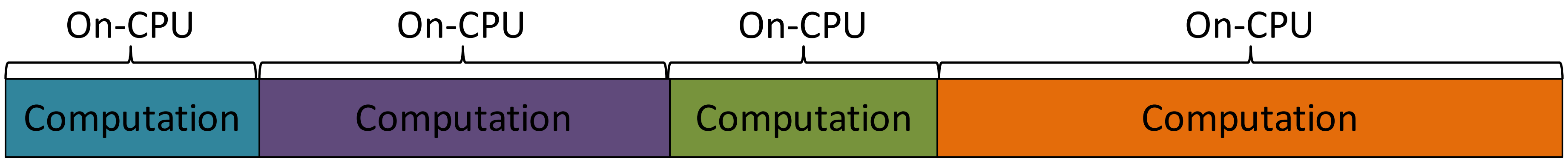


→ Bottleneck has shifted from blocking I/O to CPU

- "*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]

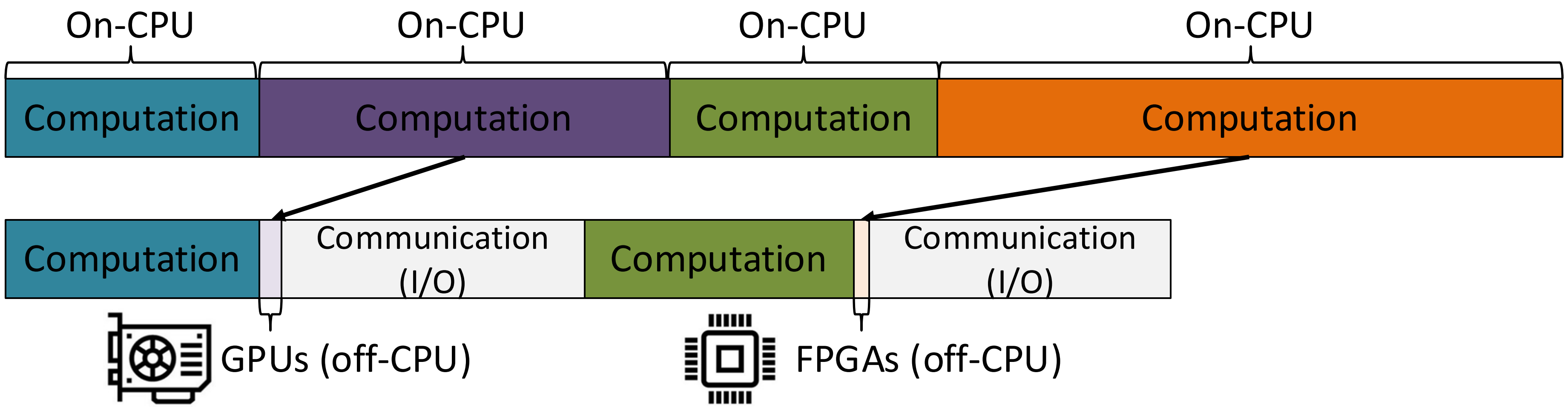
Bottlenecks of Modern Applications

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



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

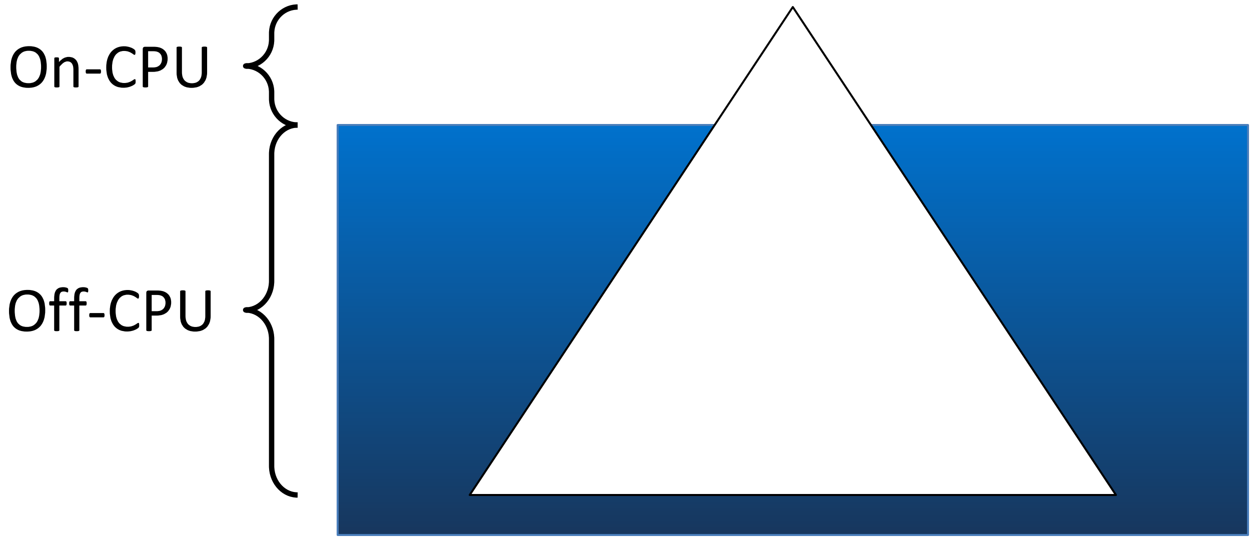


→ Bottleneck has shifted from CPU computation to I/O and communication

- "*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]

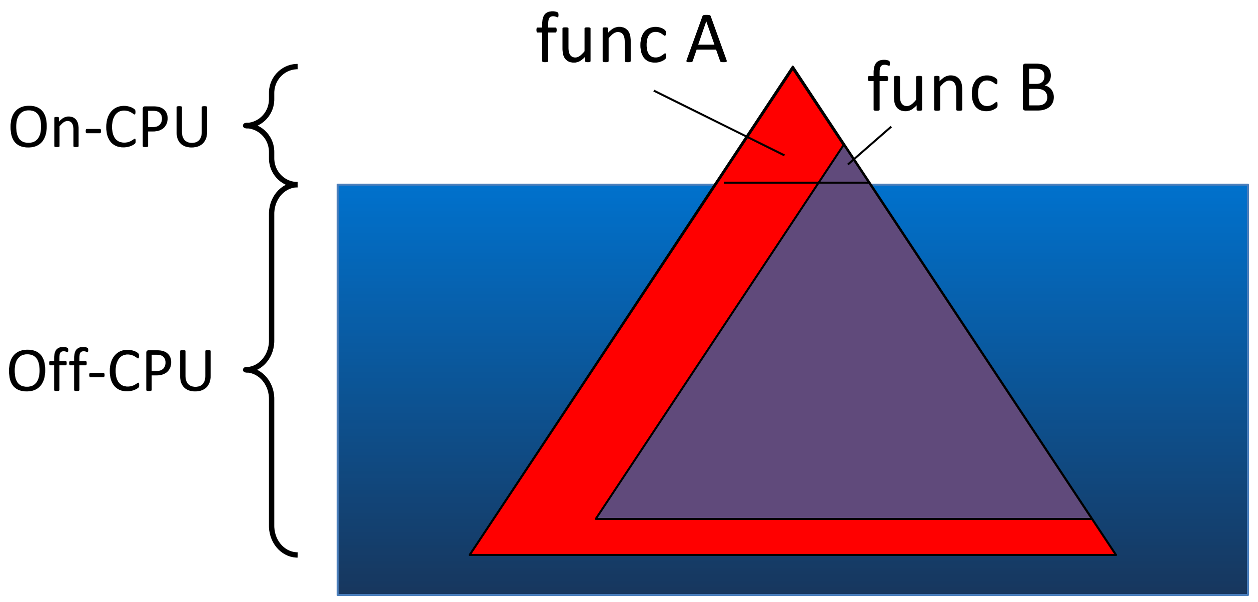
Profiling Challenge

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



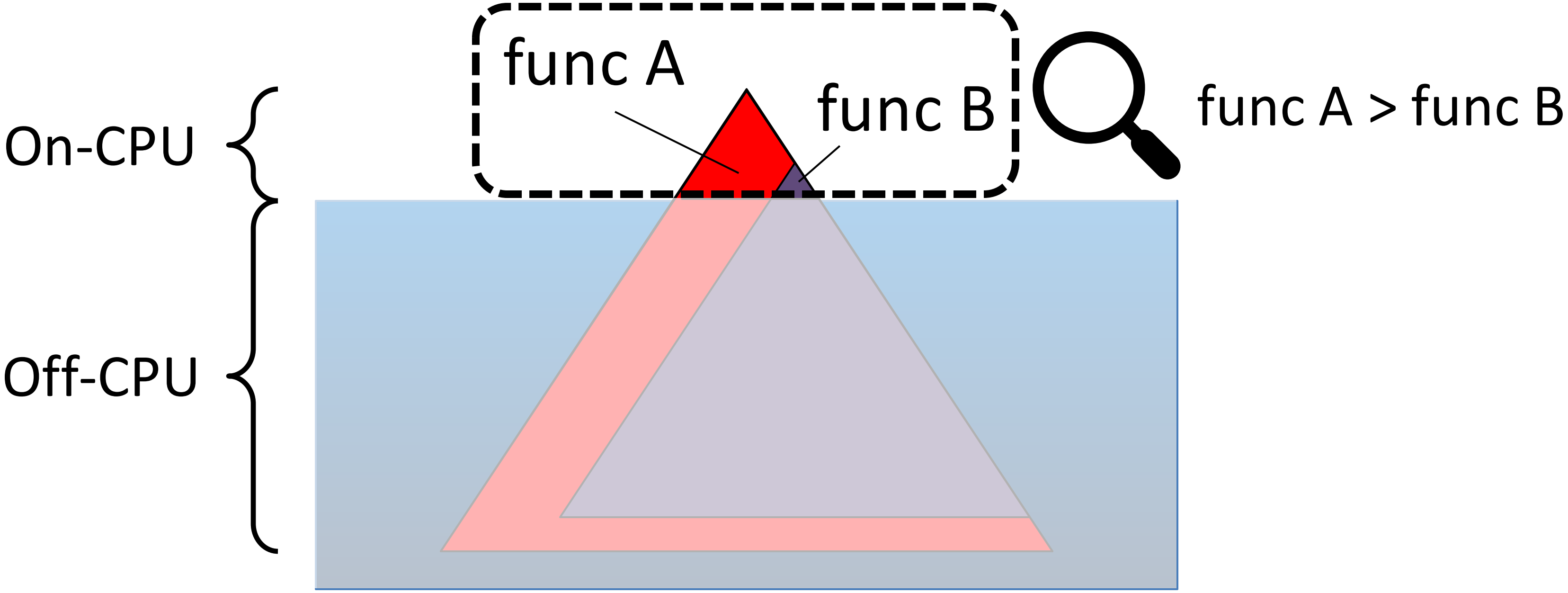
Profiling Challenge

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



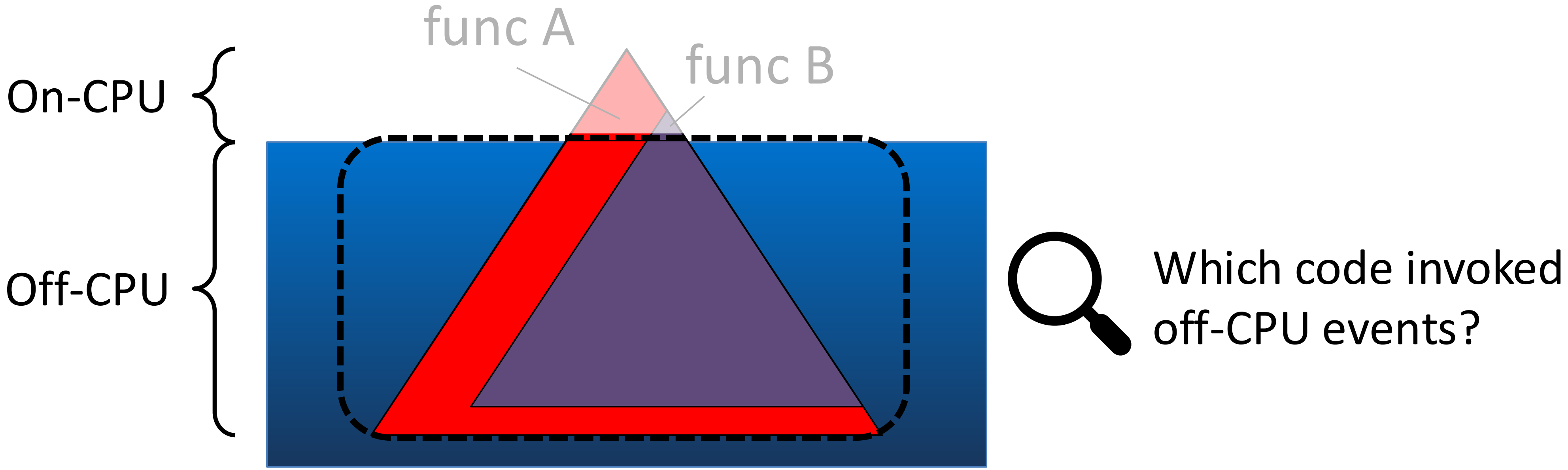
Profiling Challenge

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



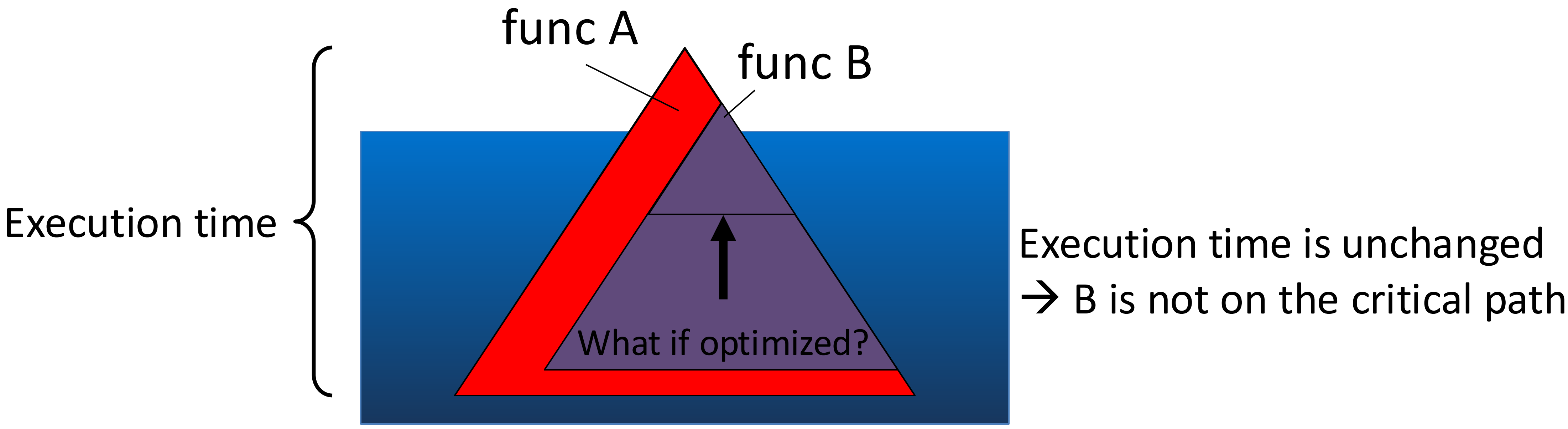
Profiling Challenge

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



Profiling Challenge

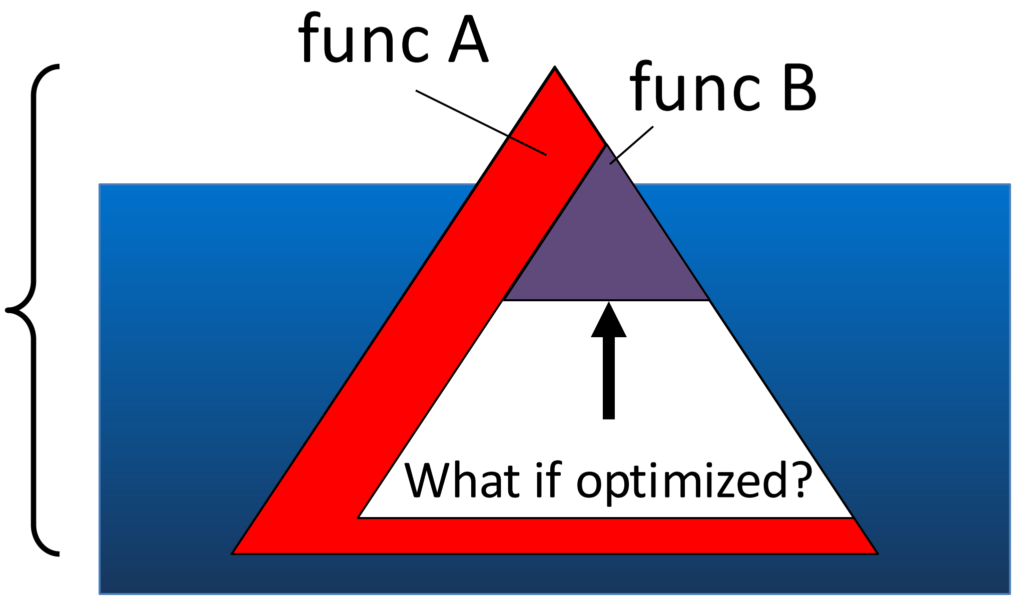
- Both on-CPU and off-CPU events need to be considered simultaneously
  - (Challenge #1) Analysis is conducted using only partial information
  - (Challenge #2) Hard to assess the impact of optimizing off-CPU events



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



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

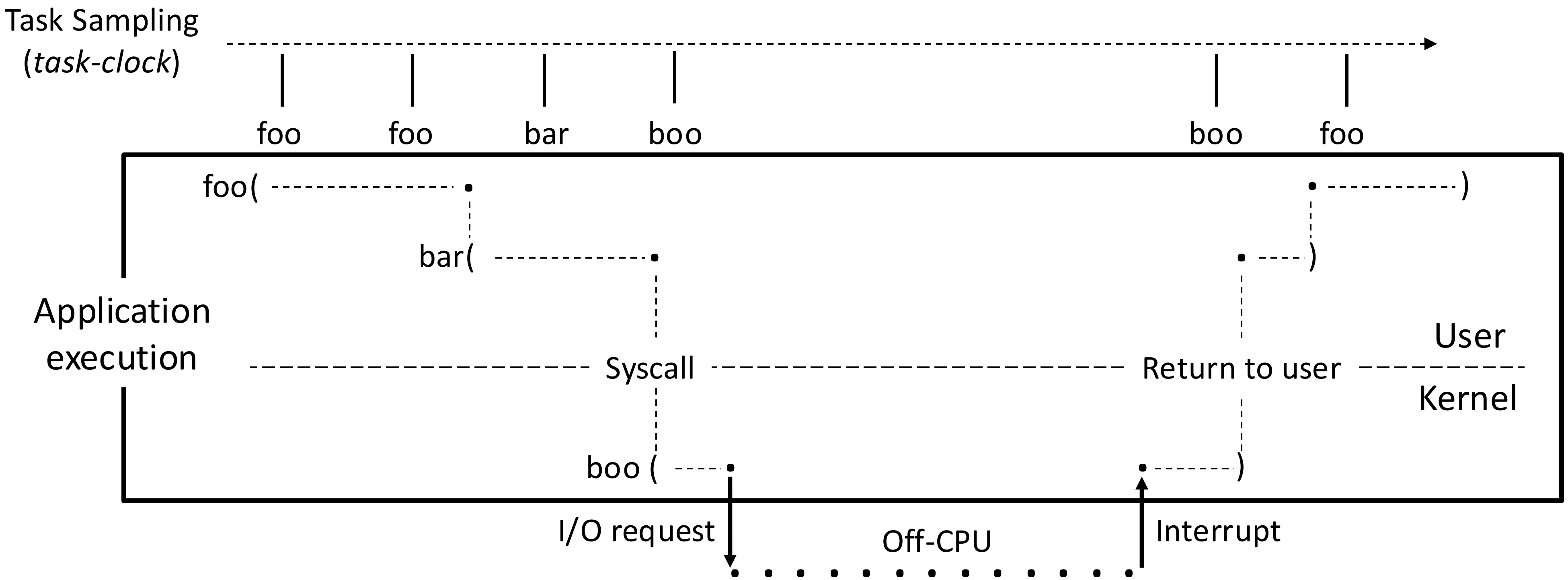
On-CPU Analysis

- 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

```
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
```

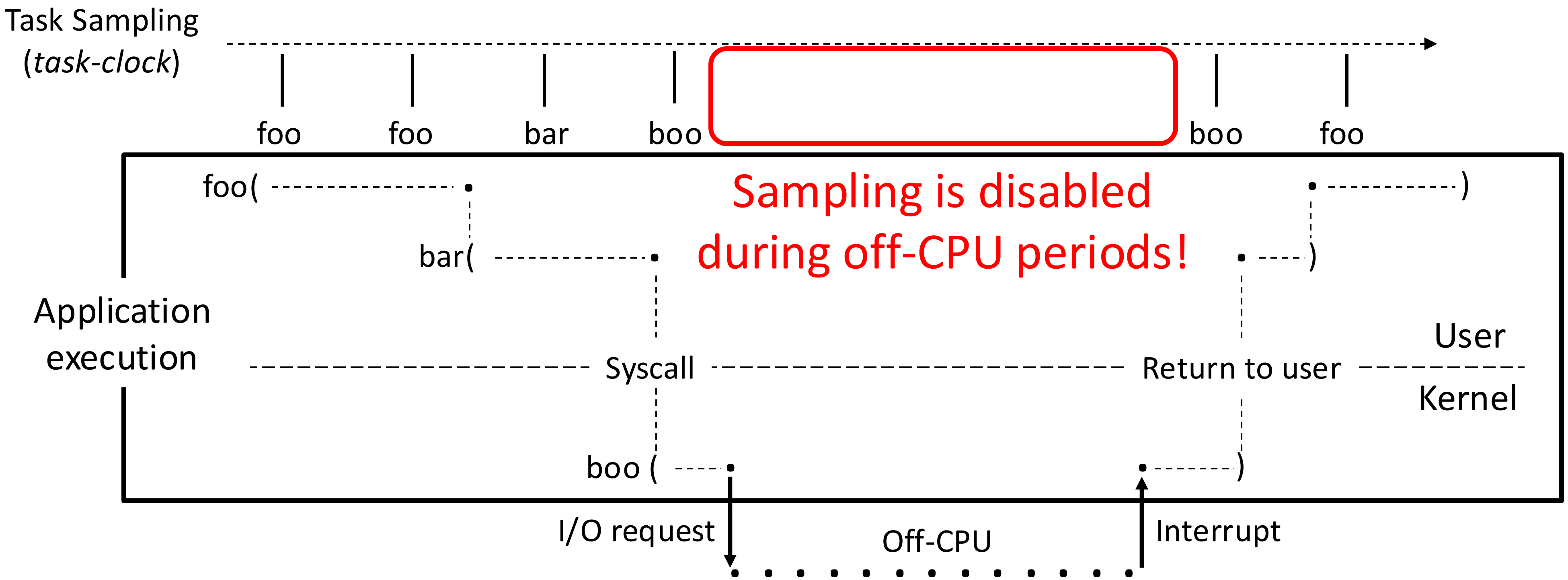
On-CPU Analysis

- 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



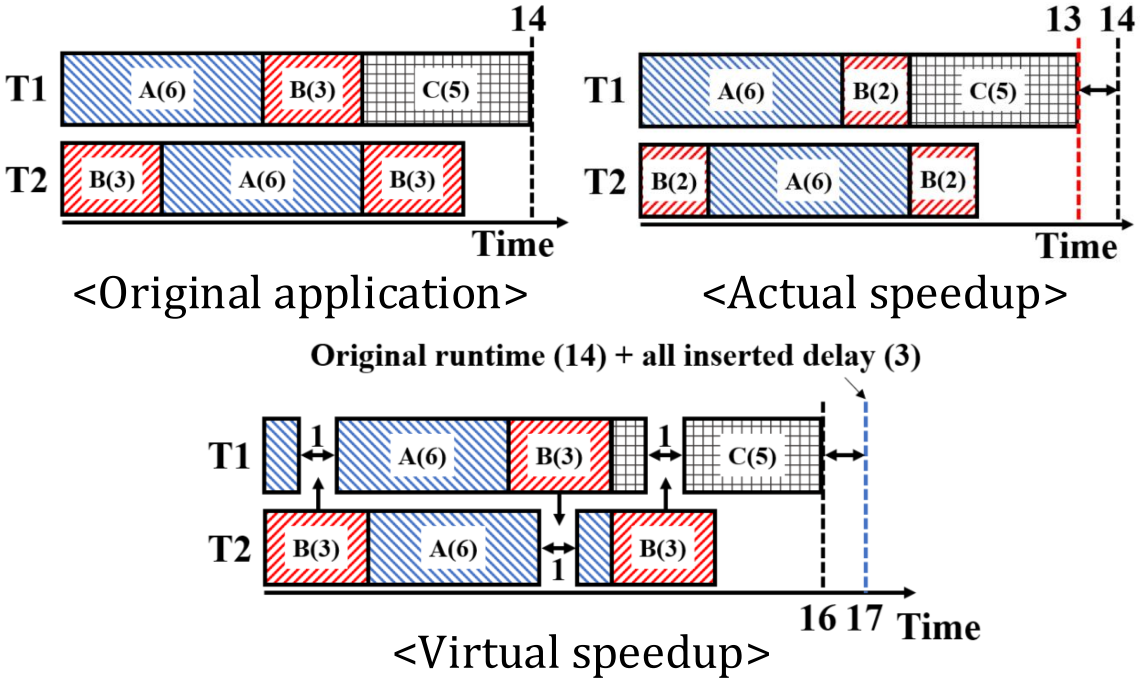
On-CPU Analysis

- 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

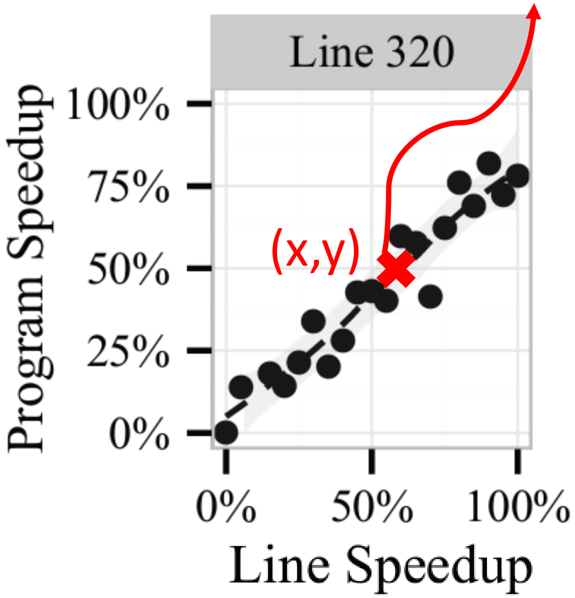


On-CPU Analysis

- COZ [SOSP ‘15]
  - Predict the impact of optimizing the specific code line <u>without actual optimization</u>
  - Virtual speedup



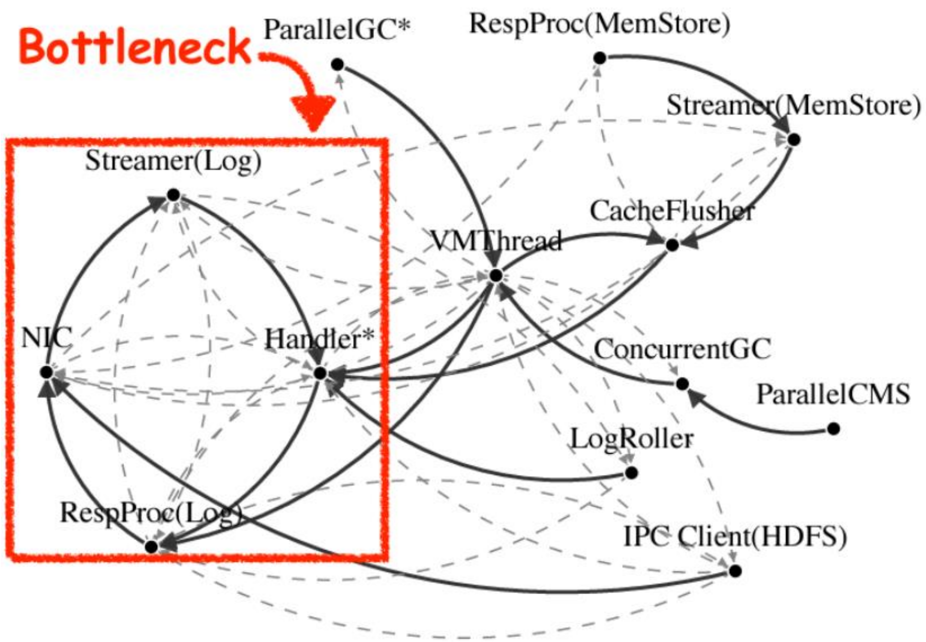
If line 320 becomes x% faster,
the program will become y% faster



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

Off-CPU Analysis

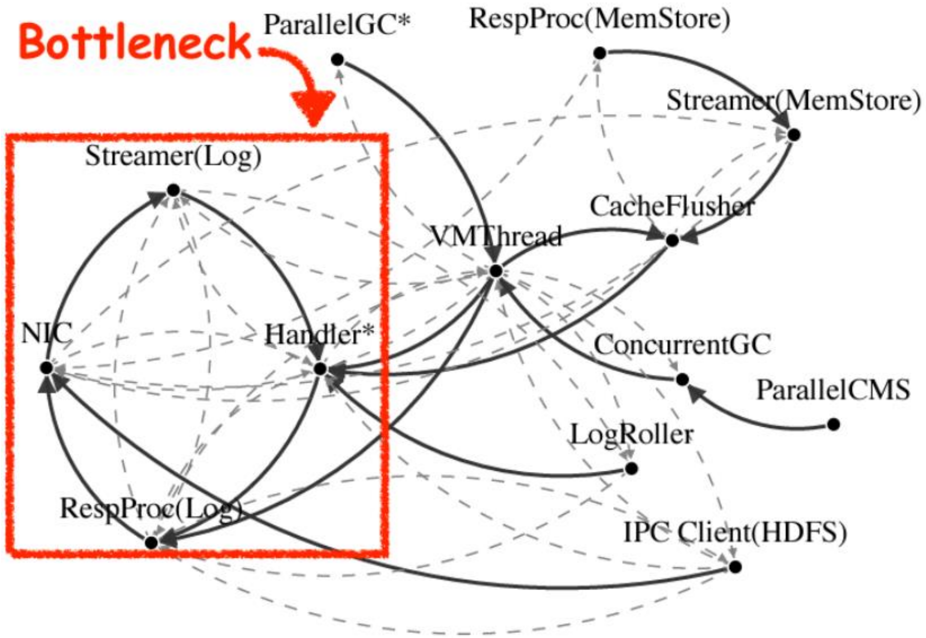
- wPerf [OSDI '18]
  - Traces all kinds of waiting events including I/O and their dependencies
  - Wait-for graph: Dependency graph of executed threads
    - Identifying closed loops (i.e., knots) through graph analysis



<Example wait-for graph>

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 <u>context information</u> of the bottleneck
   → Additional effort is needed to determine where to optimize

2) Does not provide the <u>actual impact</u> 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

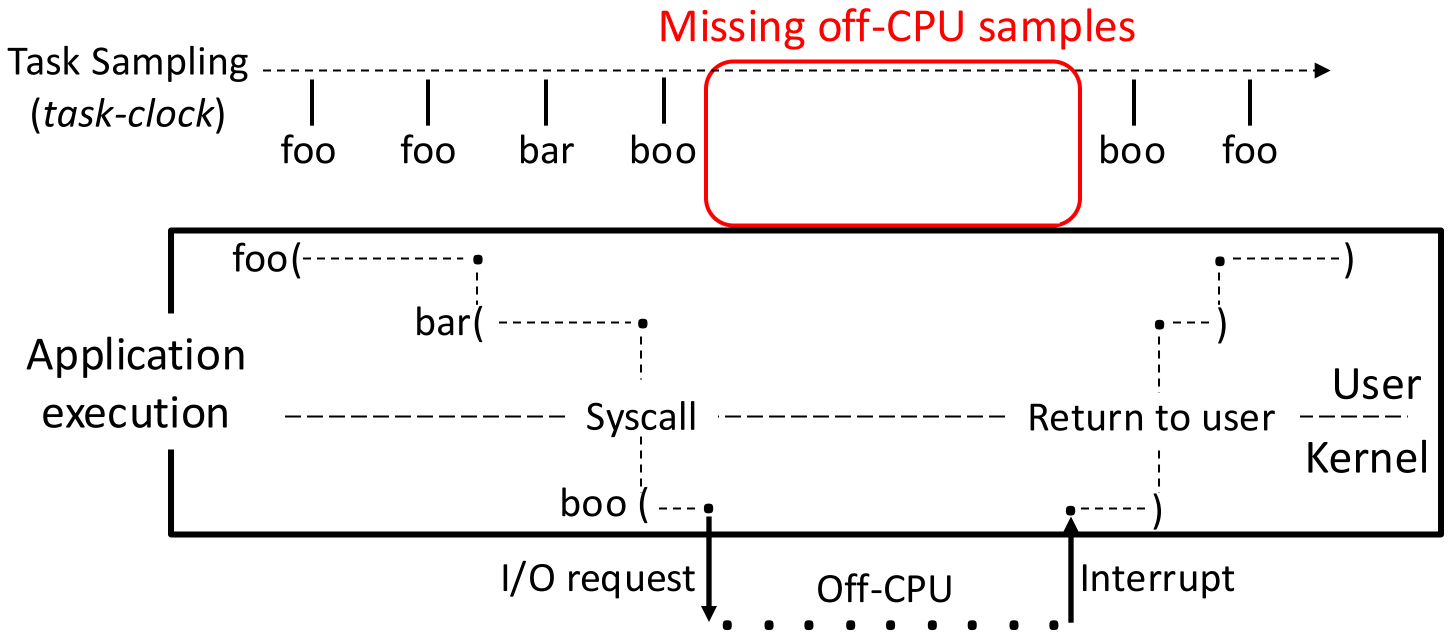
→ (Limitation #2) Causality analysis is not supported for off-CPU events

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

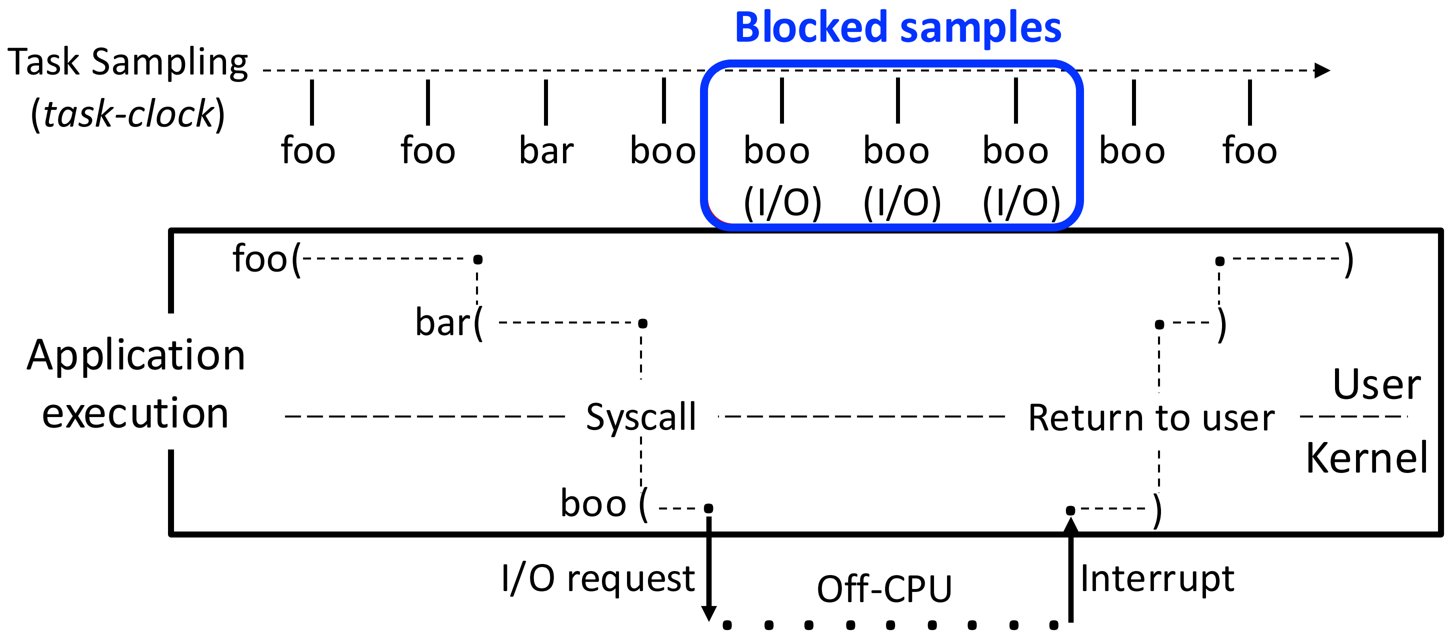
Blocked samples
(Linux perf subsystem)



Our Approach: Blocked Samples

- Goal: sampling on- and off-CPU events simultaneously
  - ***Blocked samples***: sampling technique for off-CPU events

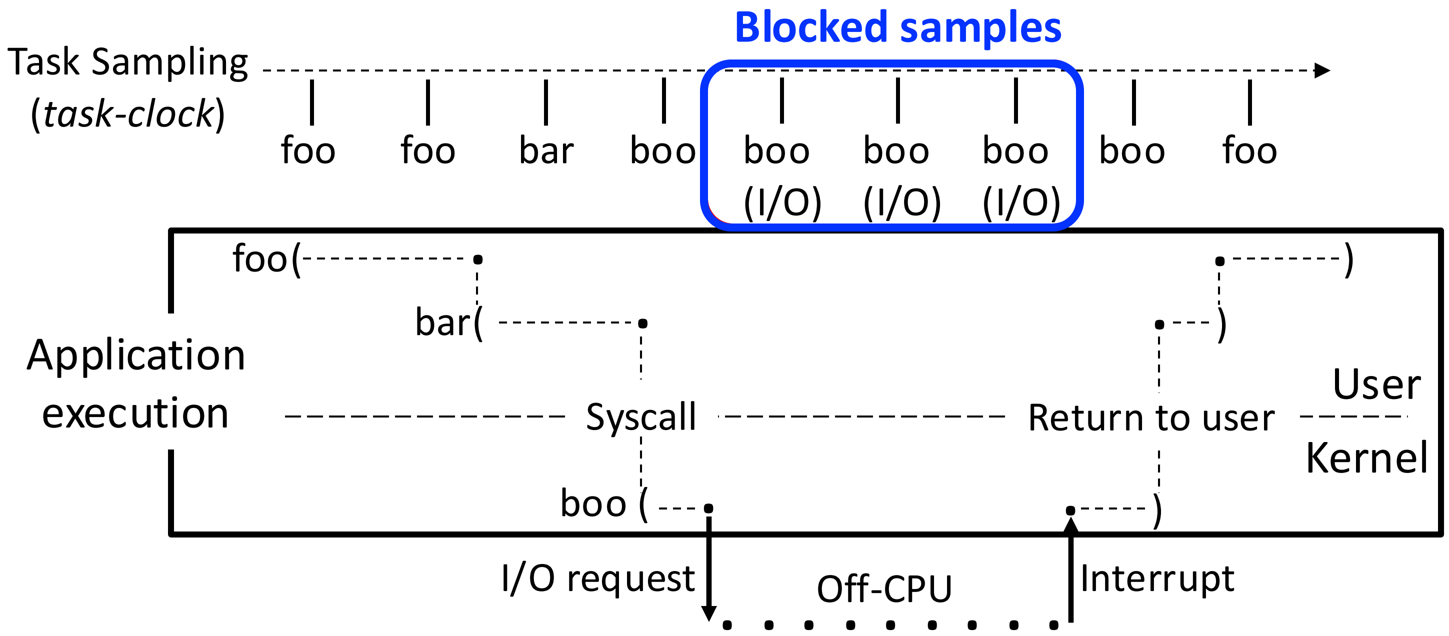
Blocked samples (Linux perf subsystem)



Our Approach: Blocked Samples

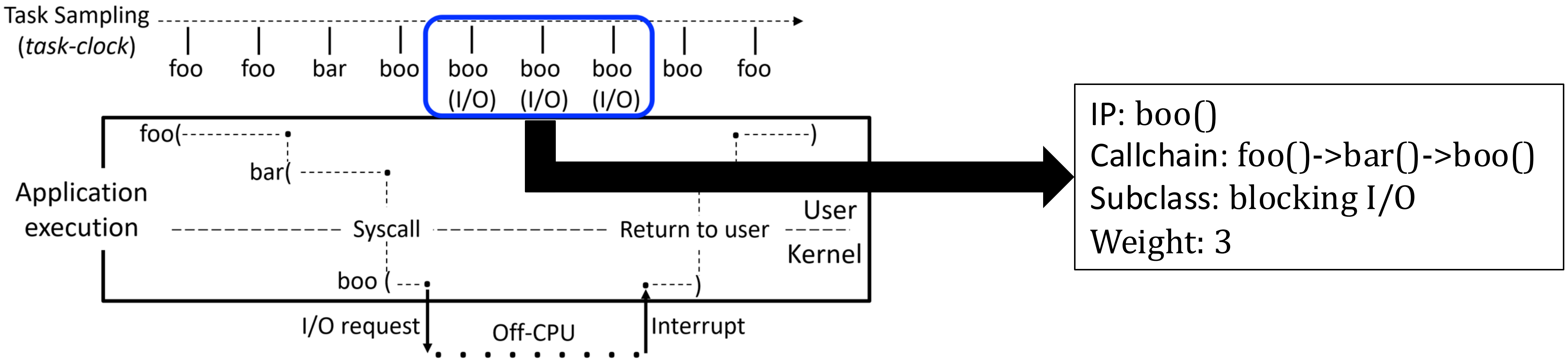
- Goal: sampling on- and off-CPU events simultaneously
  - ***Blocked samples***: sampling technique for off-CPU events
  - Proposed profilers using blocked samples
    - ***bperf***: sampling-based statistical profiler on both on-/off-CPU events
    - ***BCOZ***: causal profiler that supports virtual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

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

- Extension of Linux perf tool to support blocked samples
  - Sample accounting
  - Result reporting
    - [I]: blocking I/O, [L]: synchronization, [S]: CPU scheduling, [B]: others
    - Both the <u>last user-level IP and last kernel-level IP</u> are reported for blocked samples
      - Enables an in-depth understanding of off-CPU events

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

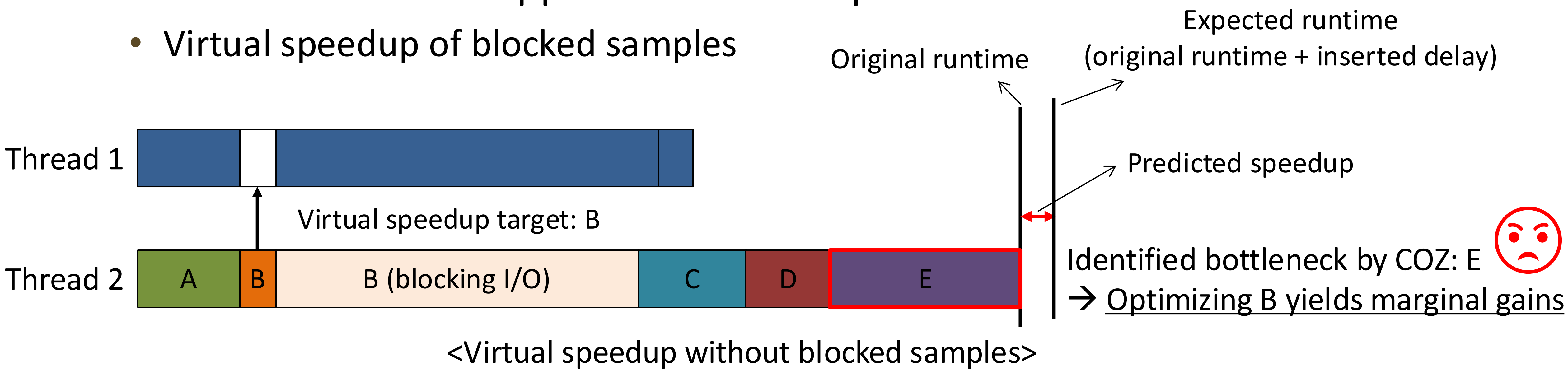
```
# Overhead  Command  Shared Object      Symbol
# ........  .......  ................   ..................................
#
    55.35%  test_io  [kernel.vmlinux]   [I] wait_on_page_bit
                                         ---[.] fsync
    27.12%  test_io  [kernel.vmlinux]   [B] jbd2_log_wait_commit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

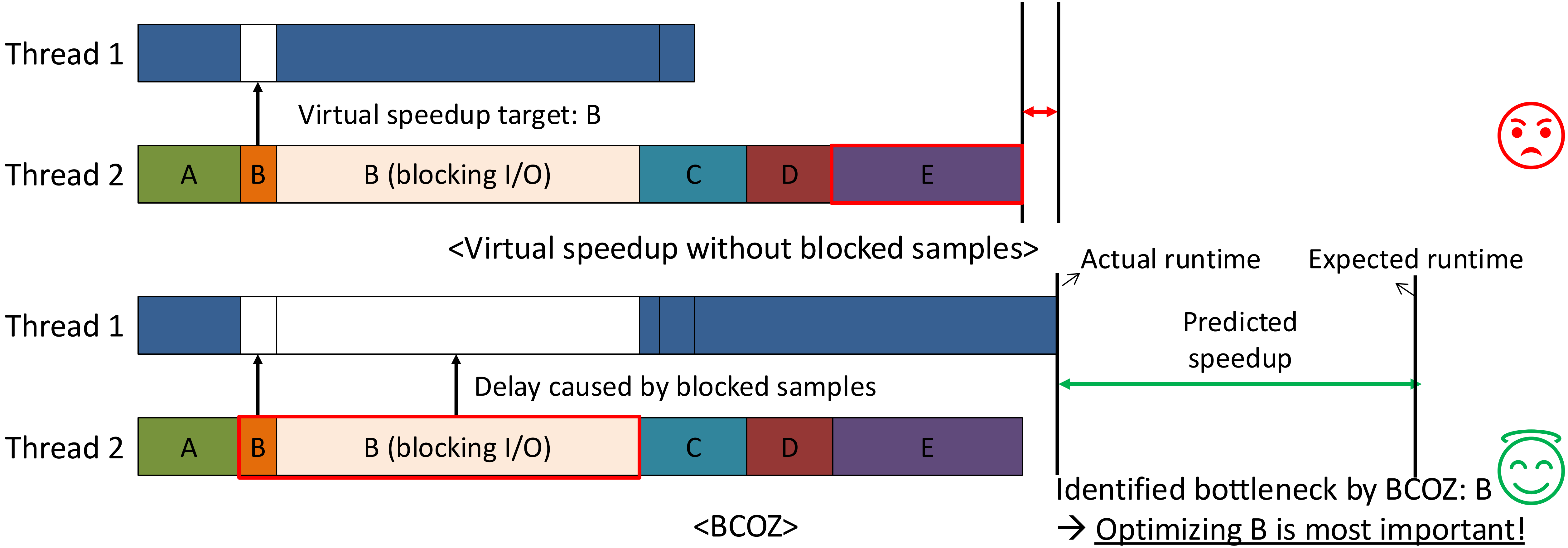
BCOZ: Causal Profiler on Both On-/Off-CPU Events

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



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

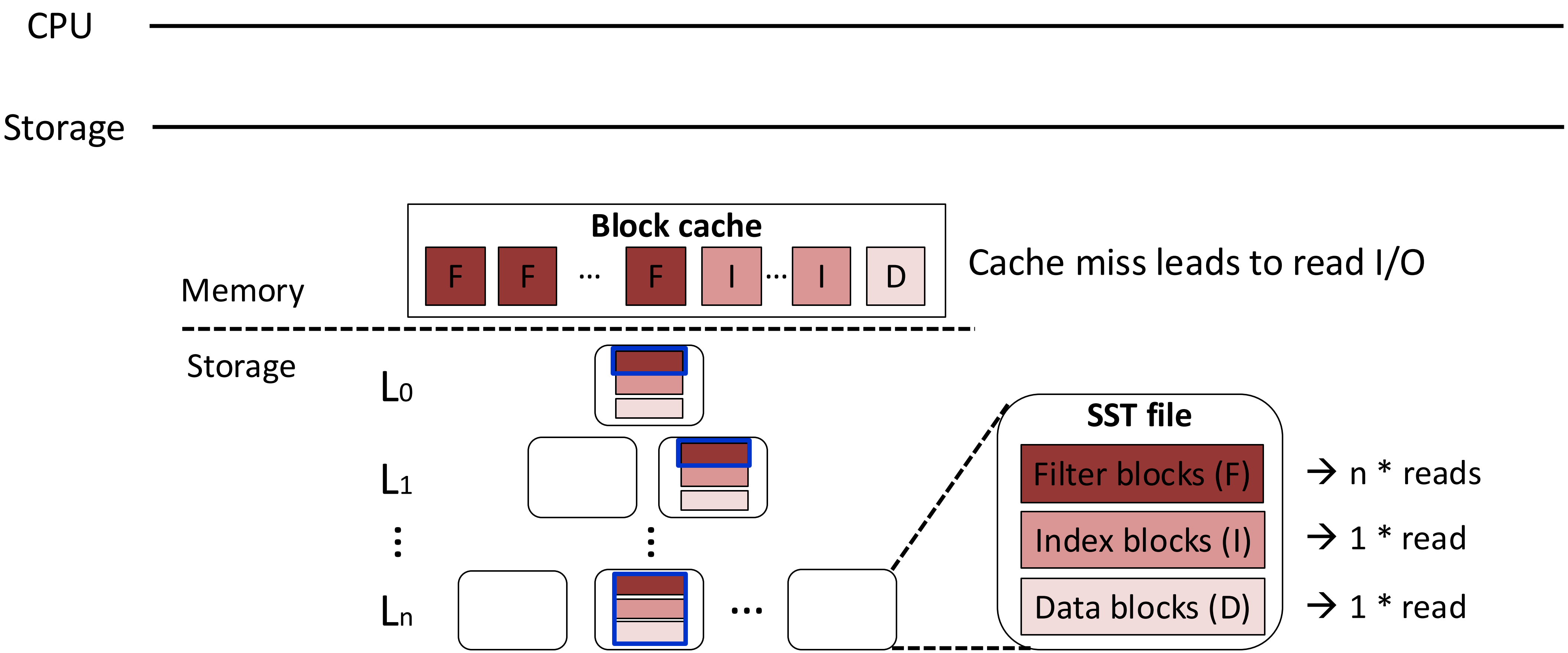
Benchmark	Workload	Identified bottlenecks	Optimization	Speedup?	Known solution?	
RocksDB	prefix_dist	Block cache contention	- Sharding	O (3.4x)	Yes	Case study 2
	allrandom	Block read I/O	- Asynchronous I/O	O (1.8x)	No	Case study 1
	fillrandom	Compaction, write stall	- No block compression - Increase 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*

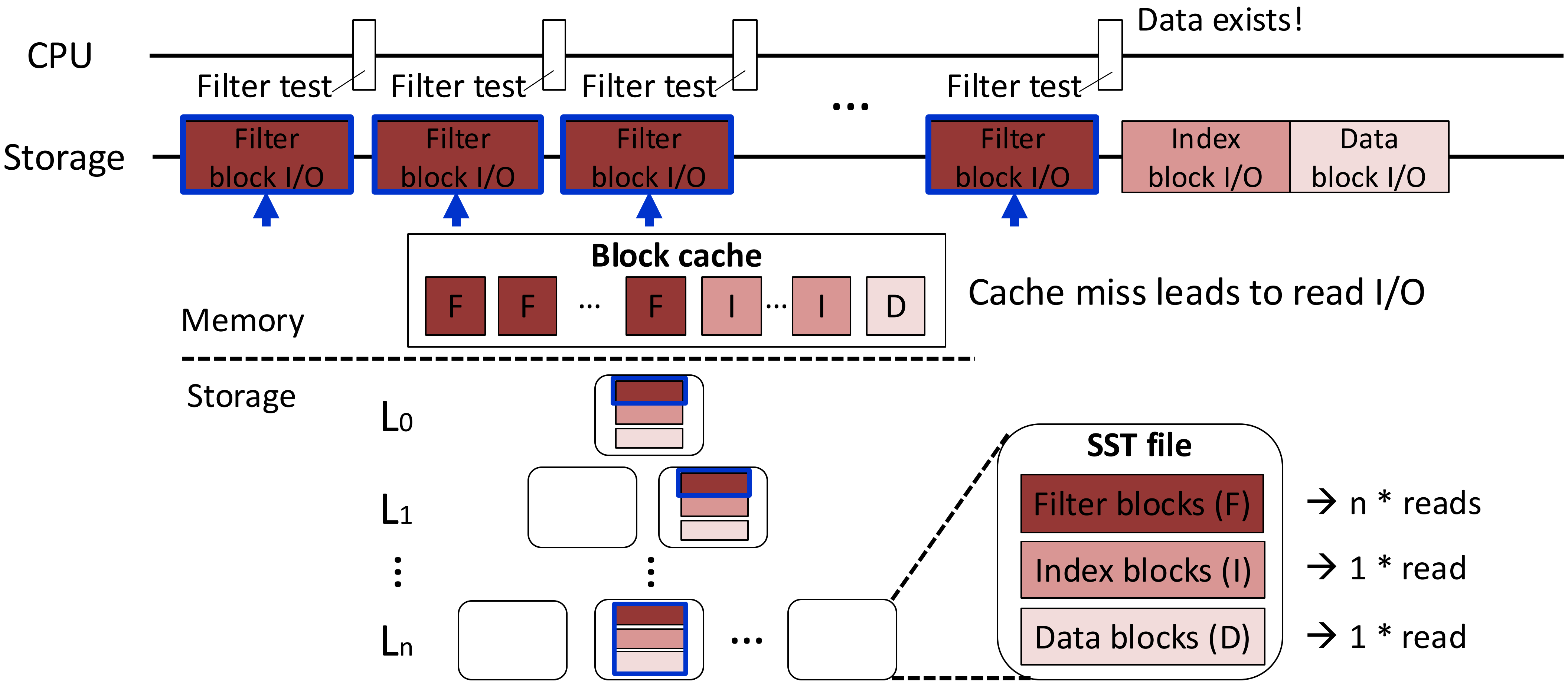
Case Study 1– RocksDB (Block Read Operation)

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



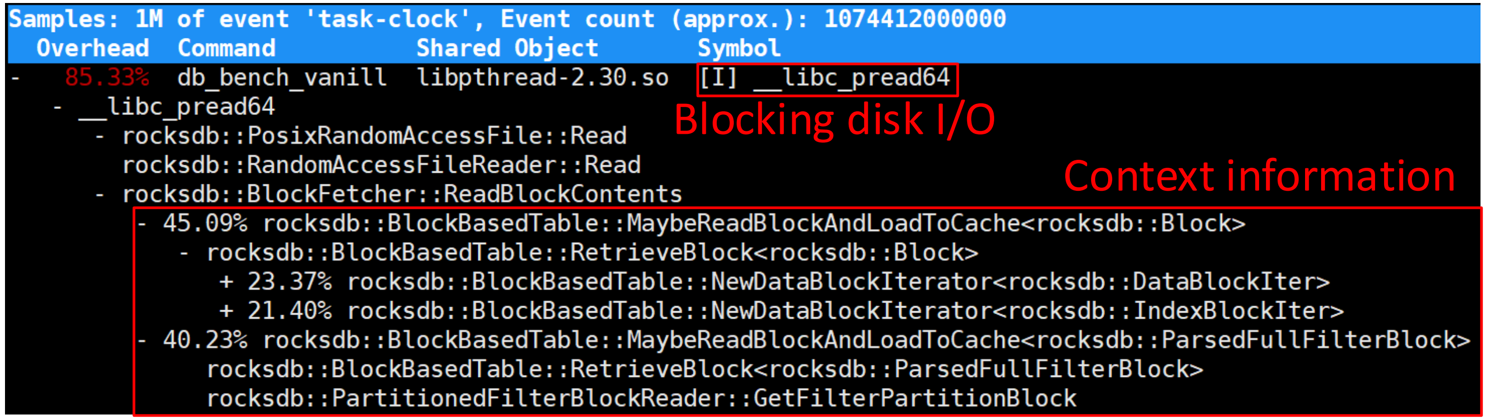
Case Study 1– RocksDB (Block Read Operation)

- Scenario: read-only workload (*allrandom*), small block cache (0.1% of dataset size)
- 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
```

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)

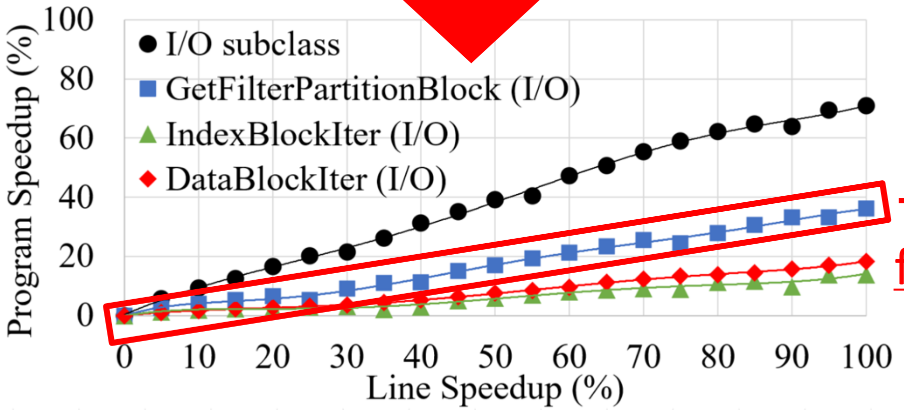
```
Samples: 1M of event 'task-clock', Event count (approx.): 1074412000000
  Overhead  Command         Shared Object        Symbol
-   85.33%  db_bench_vanill  libpthread-2.30.so   [I] __libc_pread64
   - __libc_pread64
      - rocksdb::PosixRandomAccessFile::Read
        rocksdb::RandomAccessFileReader::Read
      - 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

<*bperf*>

Causality analysis



<BCOZ>

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

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)

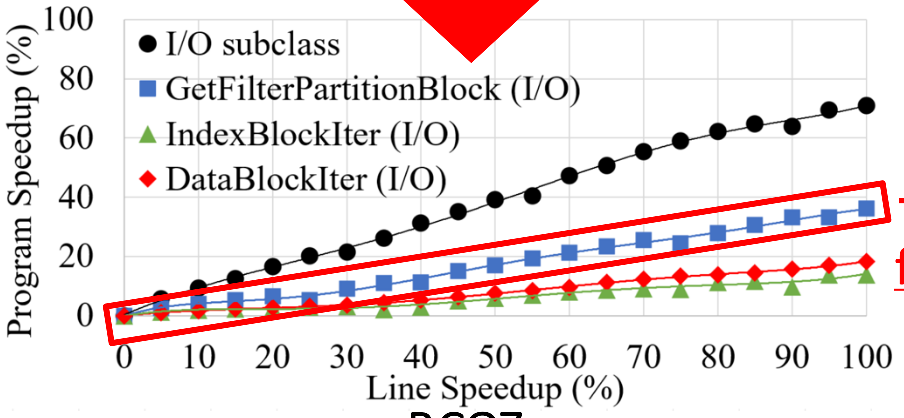
```
Samples: 1M of event 'task-clock', Event count (approx.): 1074412000000
  Overhead  Command         Shared Object       Symbol
-   85.33%  db_bench_vanill  libpthread-2.30.so  [I] __libc_pread64
   - __libc_pread64
      - rocksdb::PosixRandomAccessFile::Read
        rocksdb::RandomAccessFileReader::Read
      - 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

<bperf>

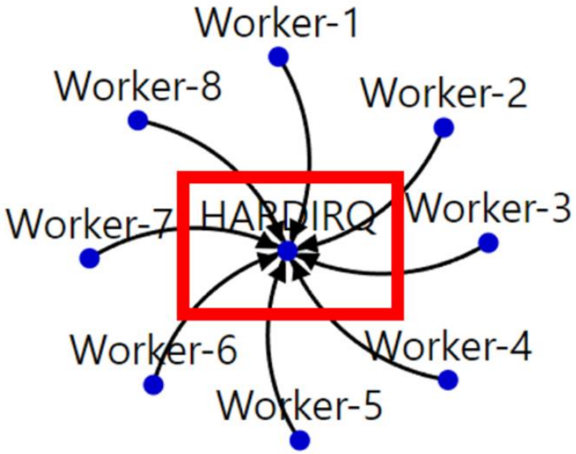
Causality analysis



<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)

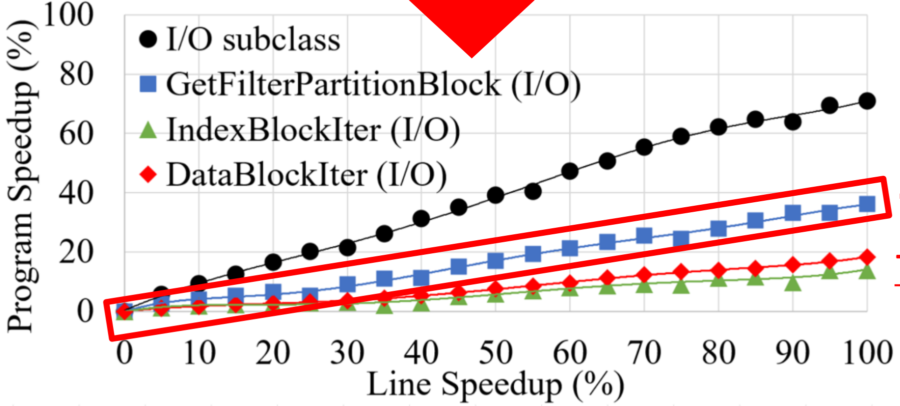
```
Samples: 1M of event 'task-clock', Event count (approx.): 1074412000000
  Overhead  Command         Shared Object       Symbol
-   85.33%  db_bench_vanill  libpthread-2.30.so  [I] __libc_pread64
   - __libc_pread64
      - rocksdb::PosixRandomAccessFile::Read
        rocksdb::RandomAccessFileReader::Read
      - 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

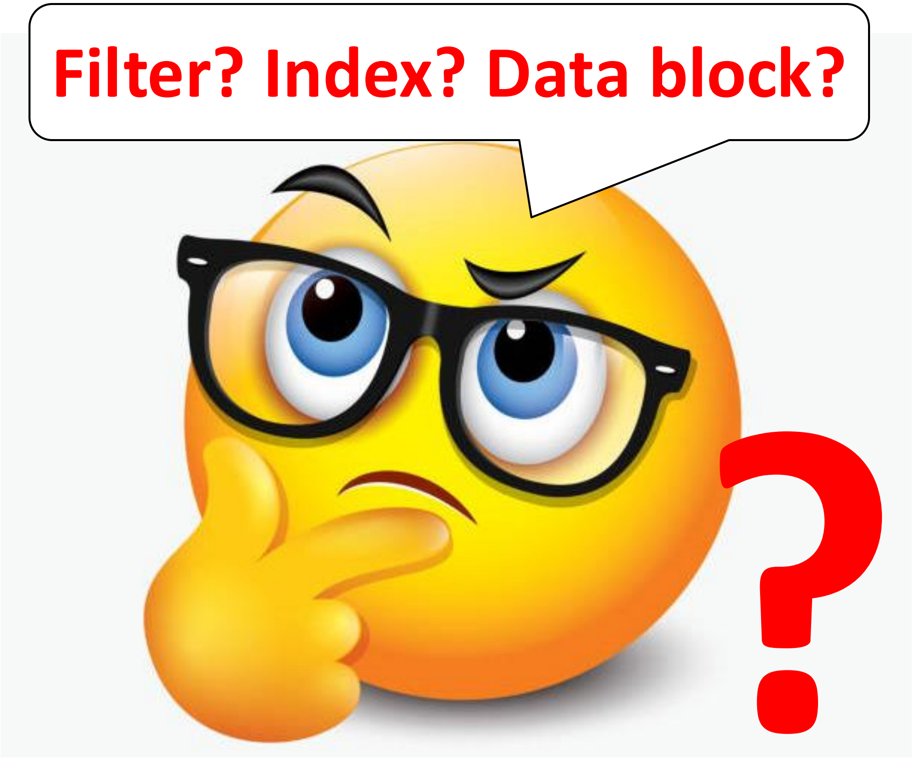
Causality analysis

<*bperf*>



<BCOZ>

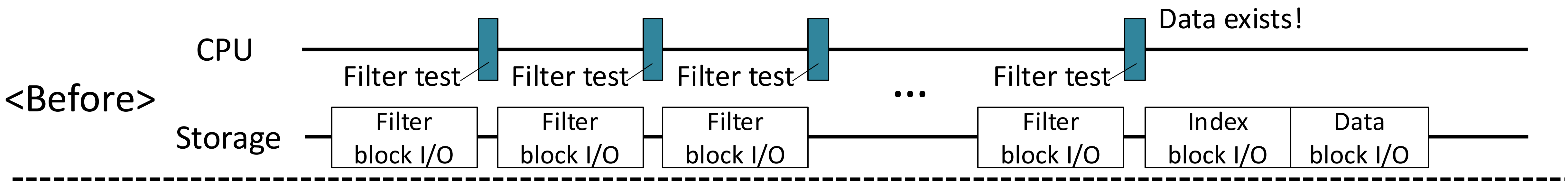
→Optimizing disk I/O of filter block is most important!



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

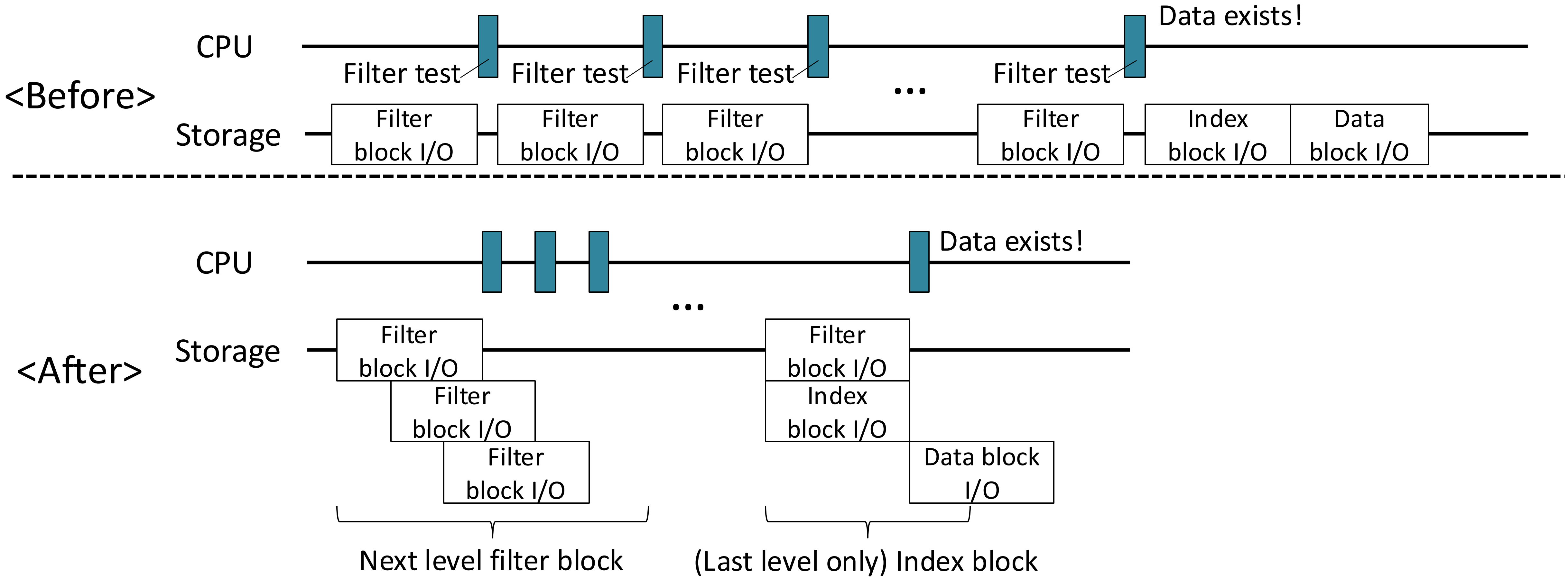
Case Study 1– RocksDB (Block Read Operation)

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



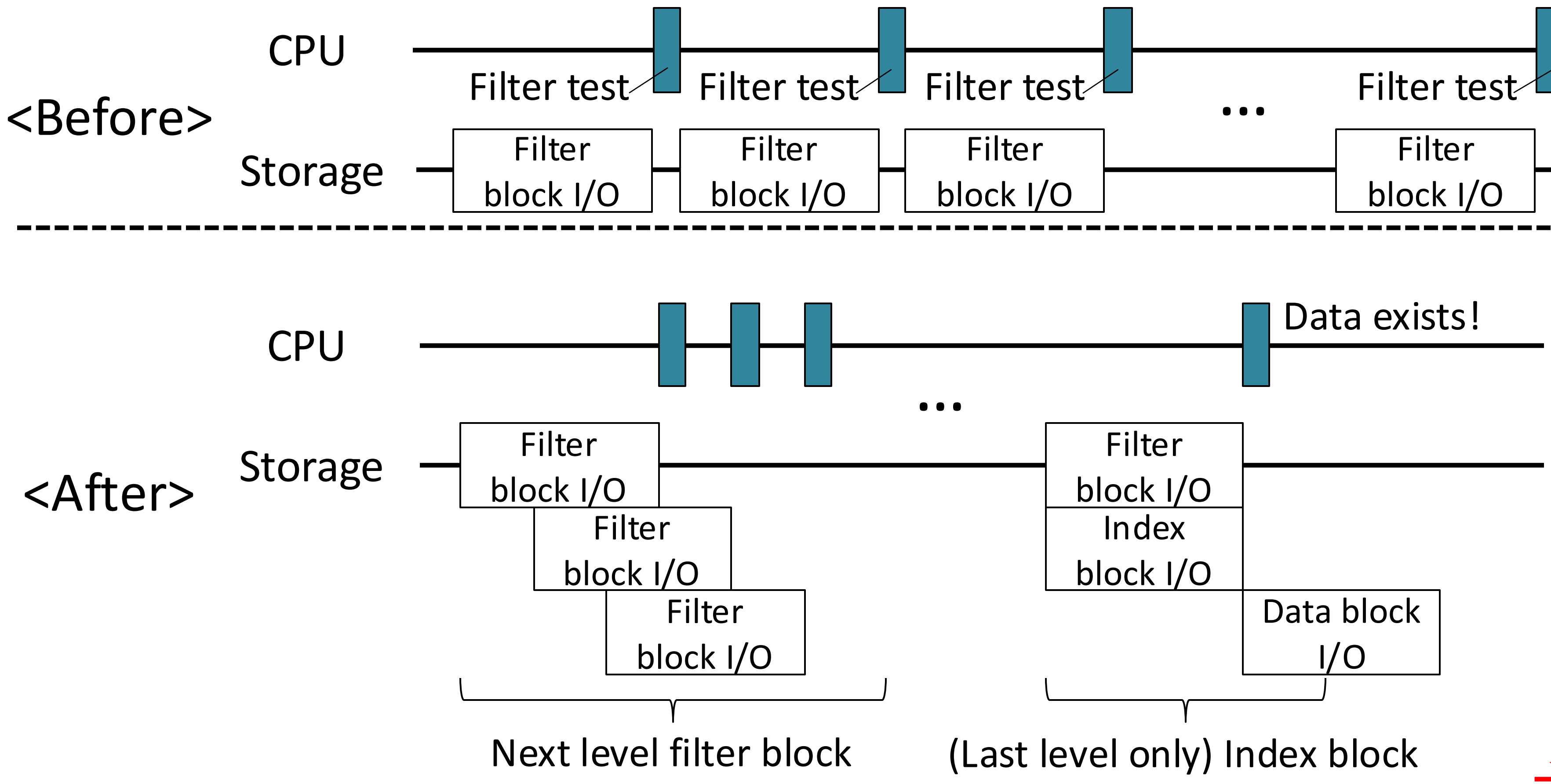
Case Study 1– RocksDB (Block Read Operation)

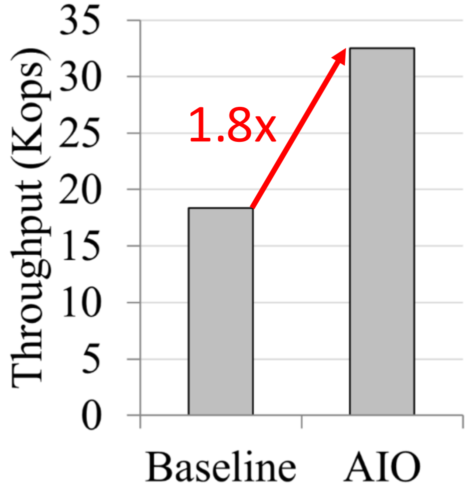
- Scenario: read-only workload (*allrandom*), small block cache (0.1% of dataset size)
- Optimization: asynchronous I/O for filter and index blocks



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



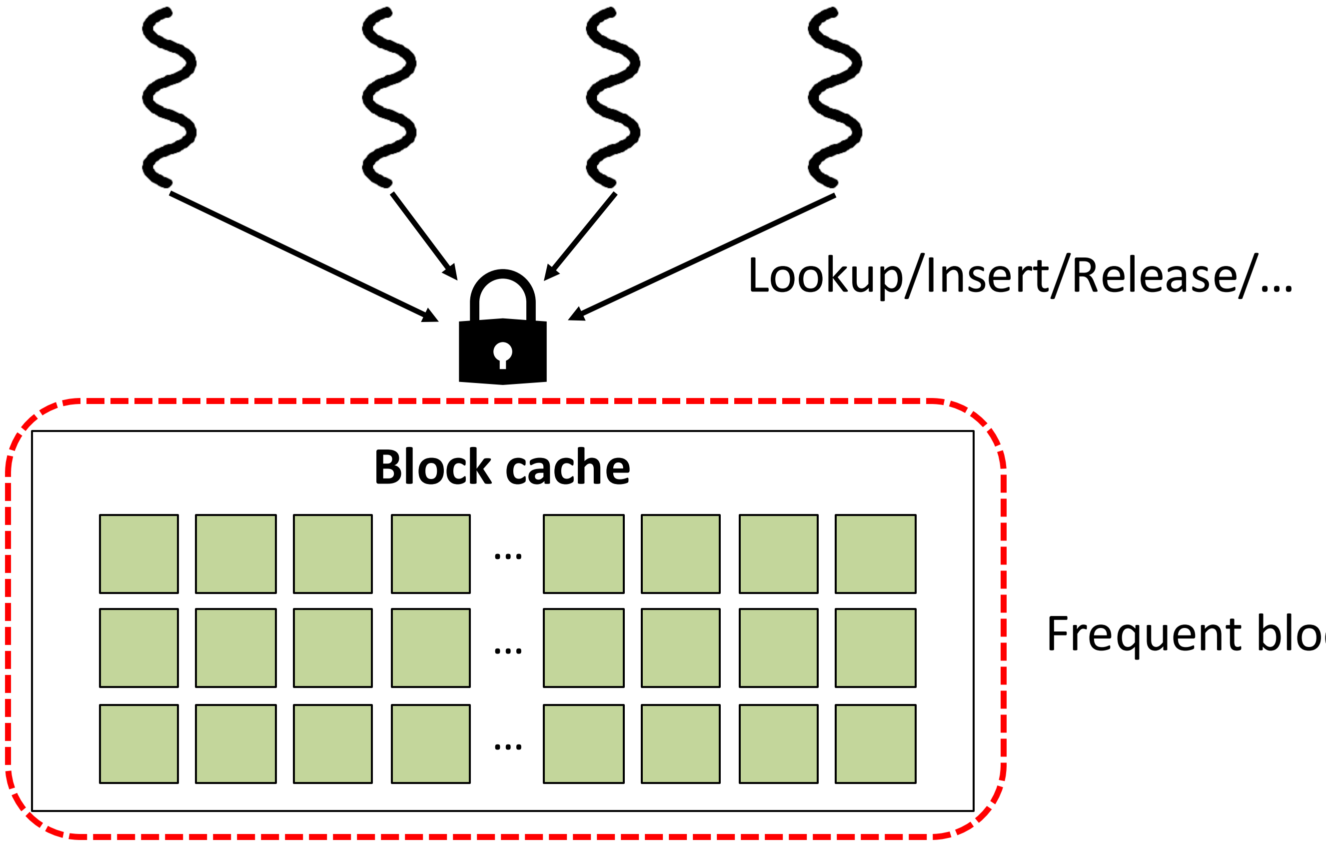


<Optimization results>

→ Blocking I/O decreased by 74%

Case Study 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

Case Study 2– RocksDB (Block Cache Contention)

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

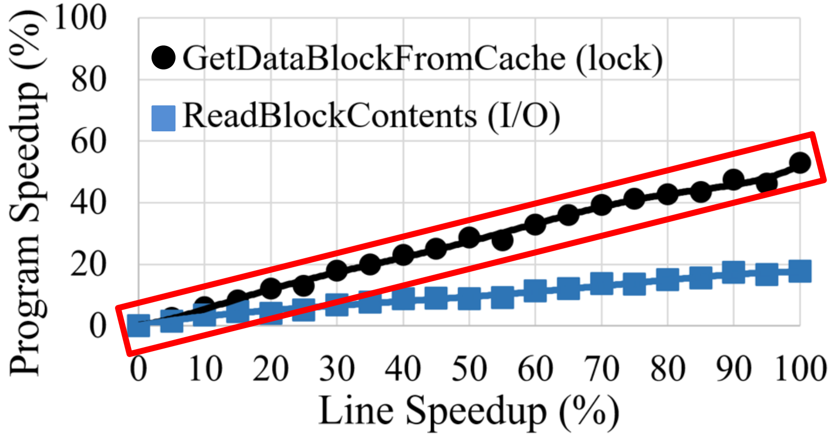
```
Samples: 1M of event 'task-clock', Event count (approx.): 1097249000000
  Overhead  Command          Shared Object        Symbol
+   25.27%  db_bench_vanill  [kernel.vmlinux]     [k] native_queued_spin_lock_slowpath
-   24.16%  db_bench_vanill  libpthread-2.30.so   [L] __lll_lock_wait
   - 24.09% __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

Context information

Blocking I/O

Causality analysis



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

Case Study 2– RocksDB (Block Cache Contention)

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

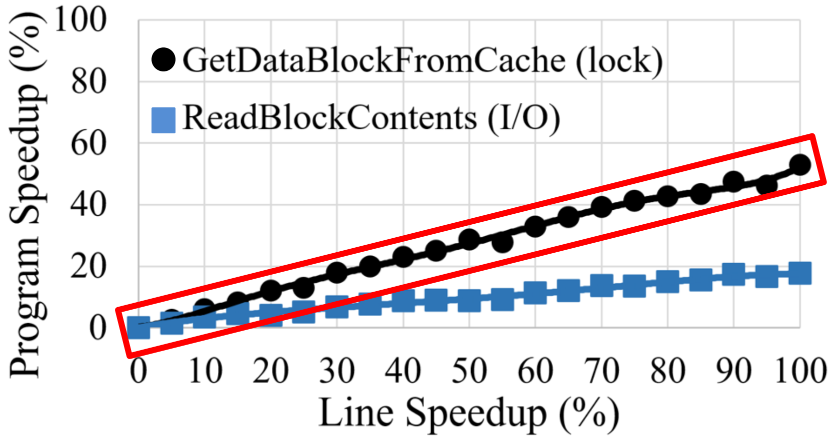
```
Samples: 1M of event 'task-clock', Event count (approx.): 1097249000000
  Overhead  Command         Shared Object       Symbol
+   25.27%  db_bench_vanill  [kernel.vmlinux]    [k] native_queued_spin_lock_slowpath
-   24.16%  db_bench_vanill  libpthread-2.30.so  [L] __lll_lock_wait
   - 24.09% __lll_lock_wait
      - __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

Context information

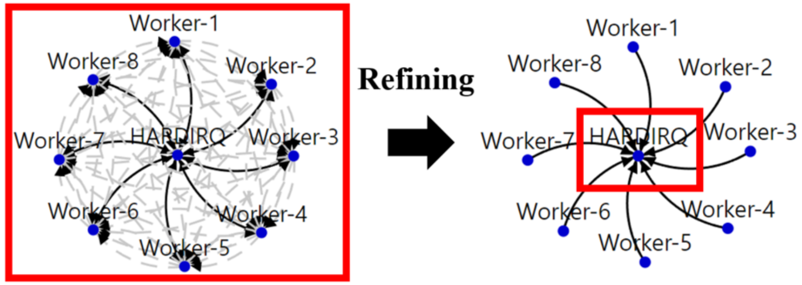
Causality analysis

Blocking I/O



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

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

Case Study 2– RocksDB (Block Cache Contention)

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

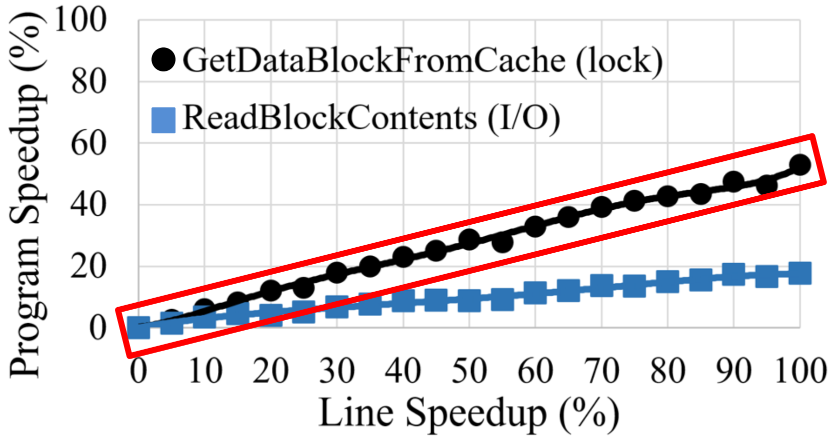
```
Samples: 1M of event 'task-clock', Event count (approx.): 1097249000000
  Overhead  Command          Shared Object        Symbol
+   25.27%  db_bench_vanill  [kernel.vmlinux]     [k] native_queued_spin_lock_slowpath
-   24.16%  db_bench_vanill  libpthread-2.30.so   [L] __lll_lock_wait
   - 24.09% __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

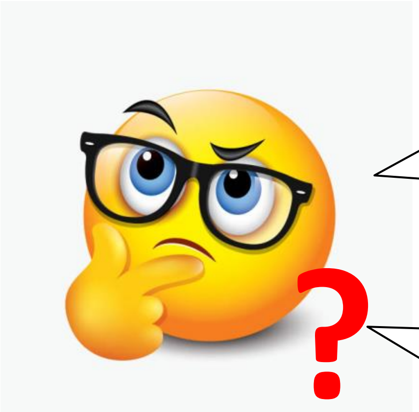
Context information

Blocking I/O

Causality analysis



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



Lock or I/O?

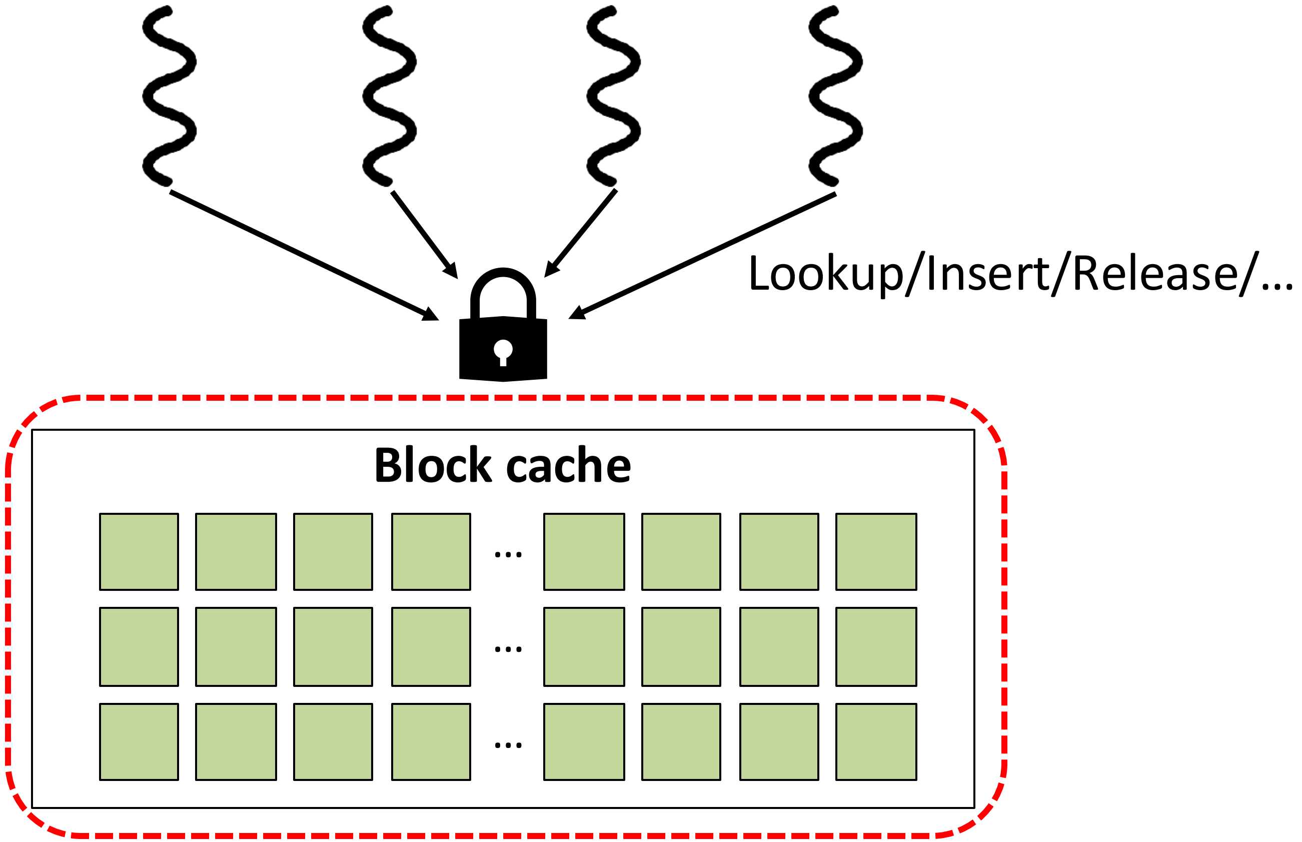
Lookup? Insert? Release?

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

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

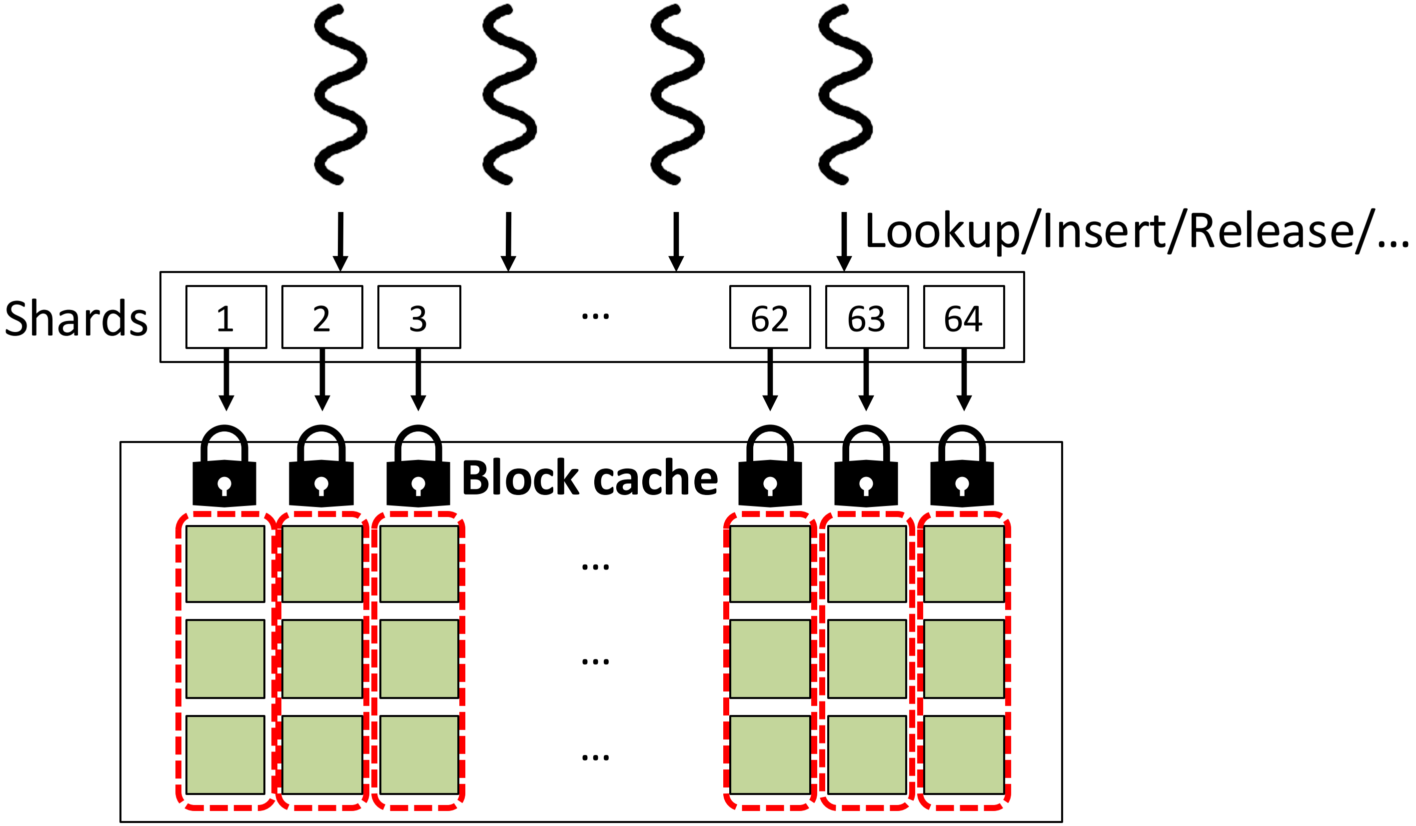
Case Study 2– RocksDB (Block Cache Contention)

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



Case Study 2– RocksDB (Block Cache Contention)

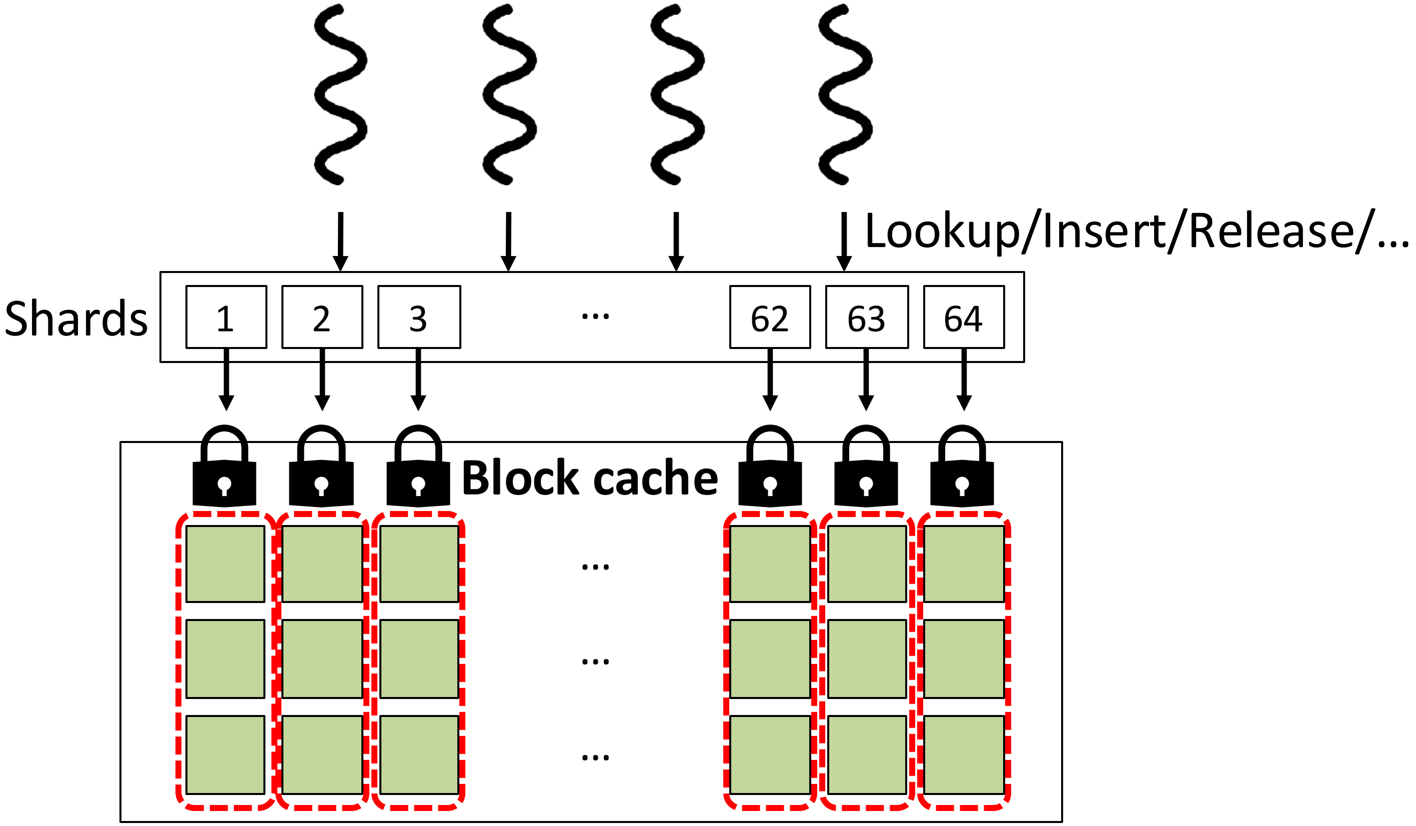
- Scenario: read-only workload (*prefix_dist*), large block cache (10% of dataset size)
- Optimization: apply sharding



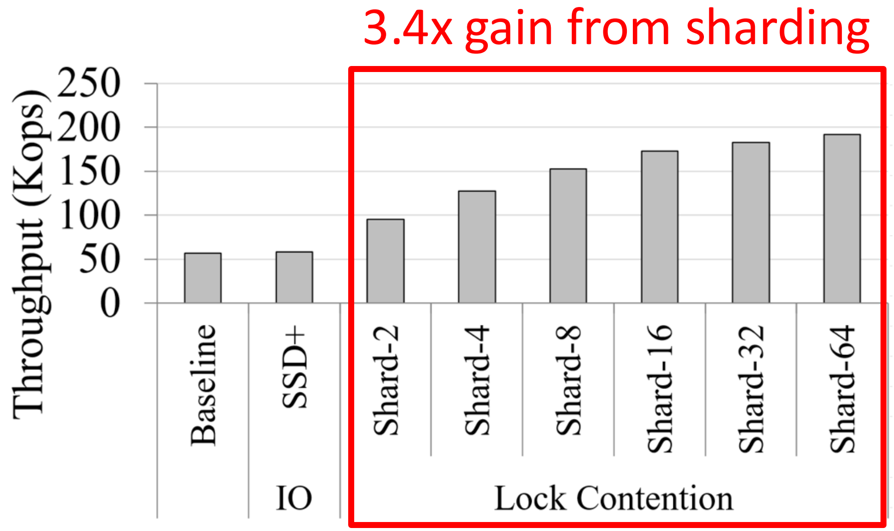
<Optimization (sharding)>

Case Study 2– RocksDB (Block Cache Contention)

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



<Optimization (sharding)>

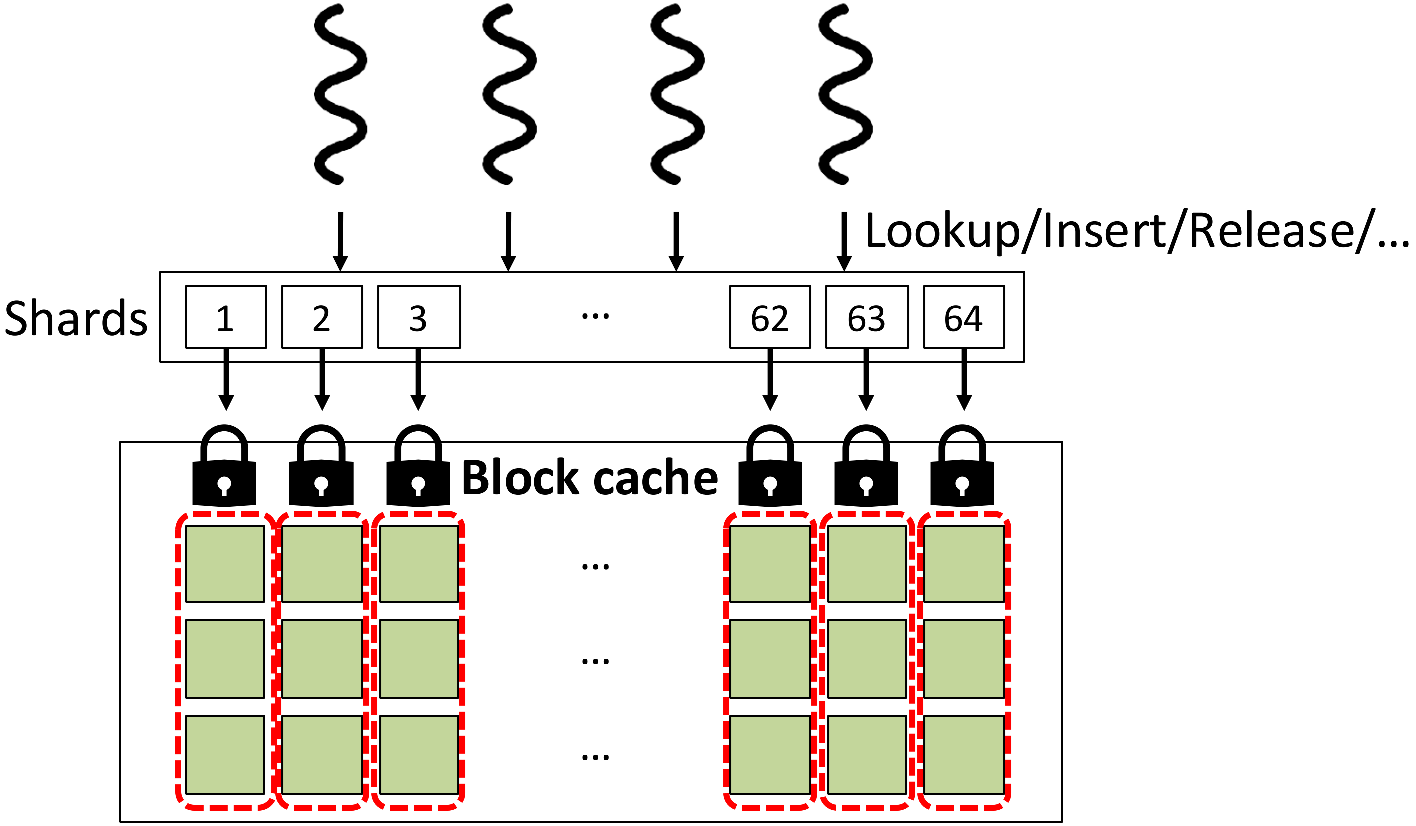


→ Lock-contention decreased by 97%

<Optimization results>

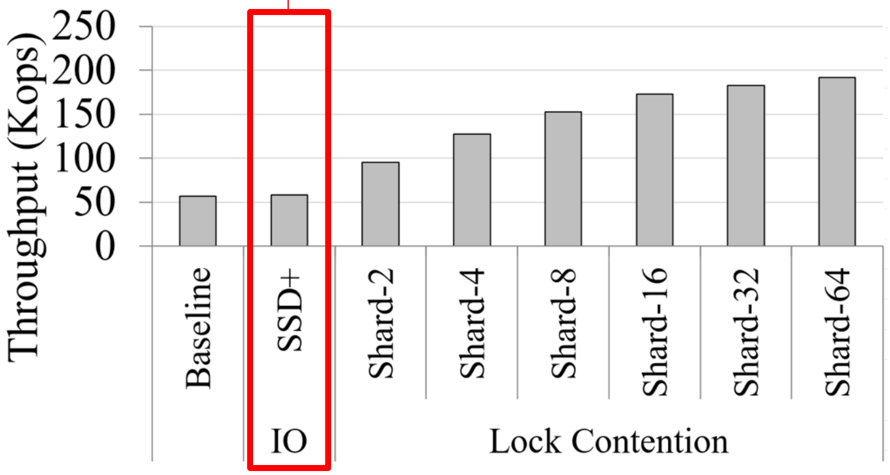
Case Study 2– RocksDB (Block Cache Contention)

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



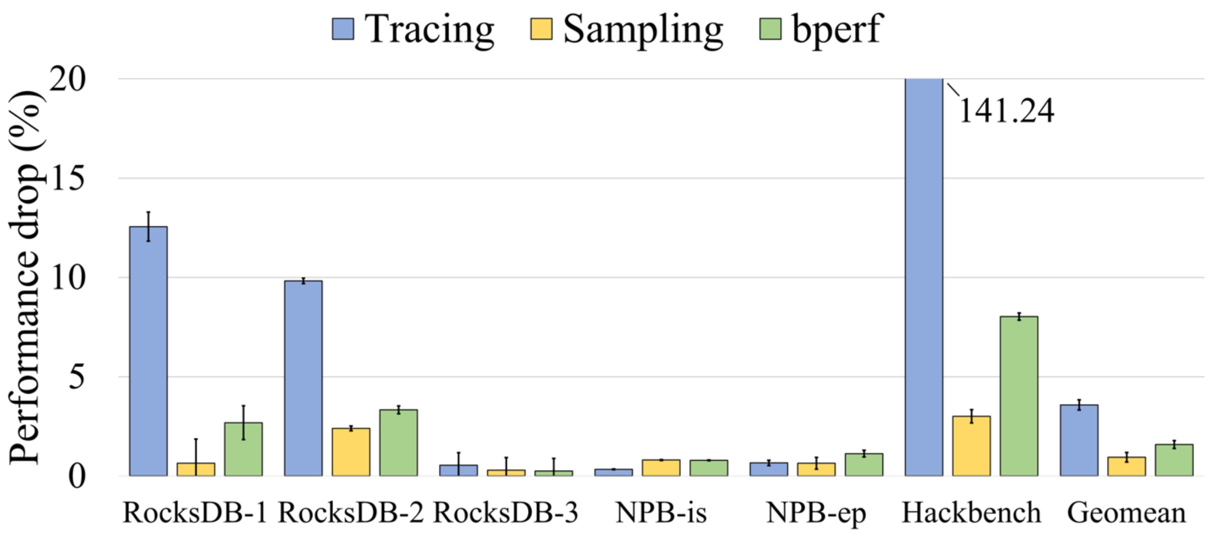
<Optimization (sharding)>

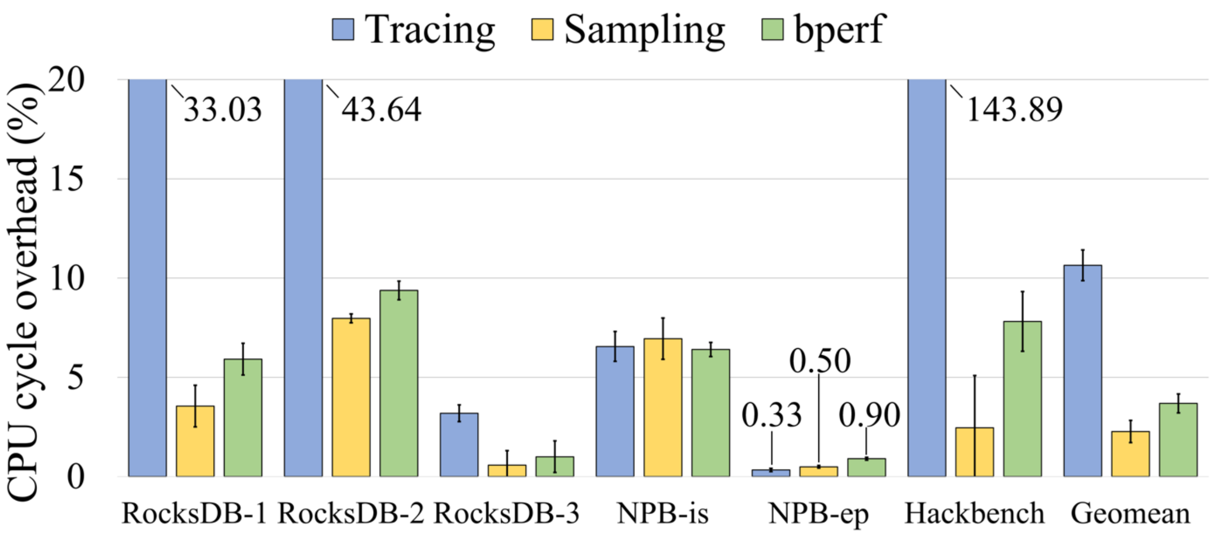
Marginal gain from blocking disk I/O

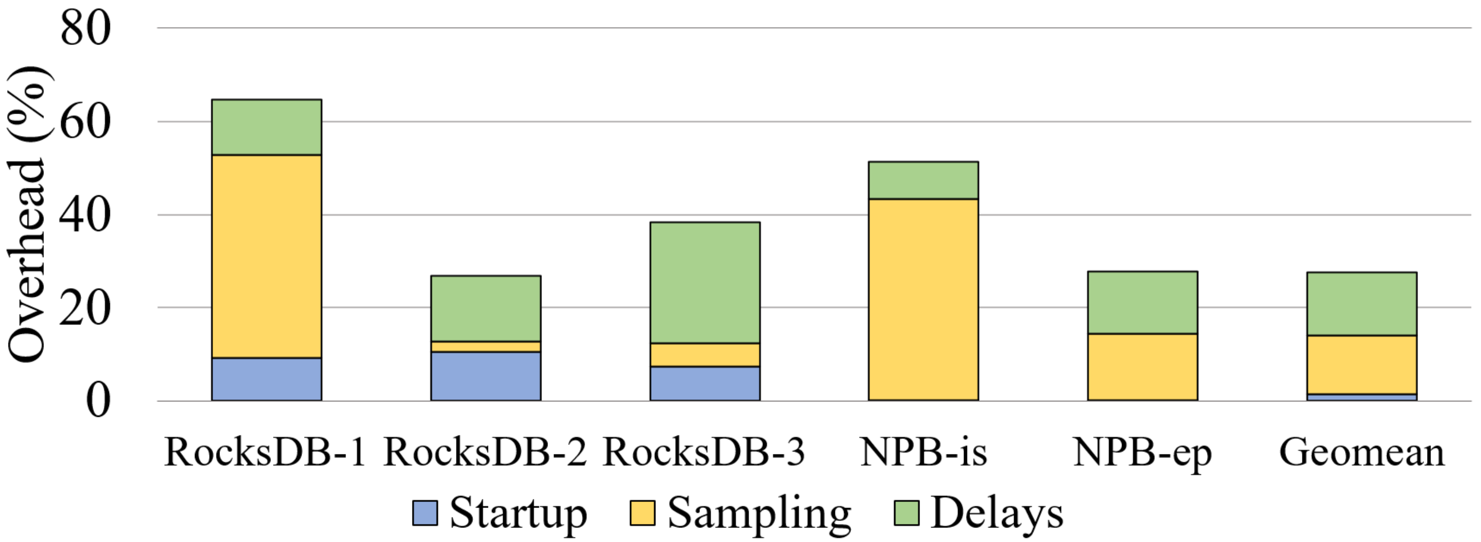


<Optimization results>

Profiling Overhead







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., remote ops.)

Blocked samples is available at:
https://github.com/s3yonsei/blocked_samples

Thank you!

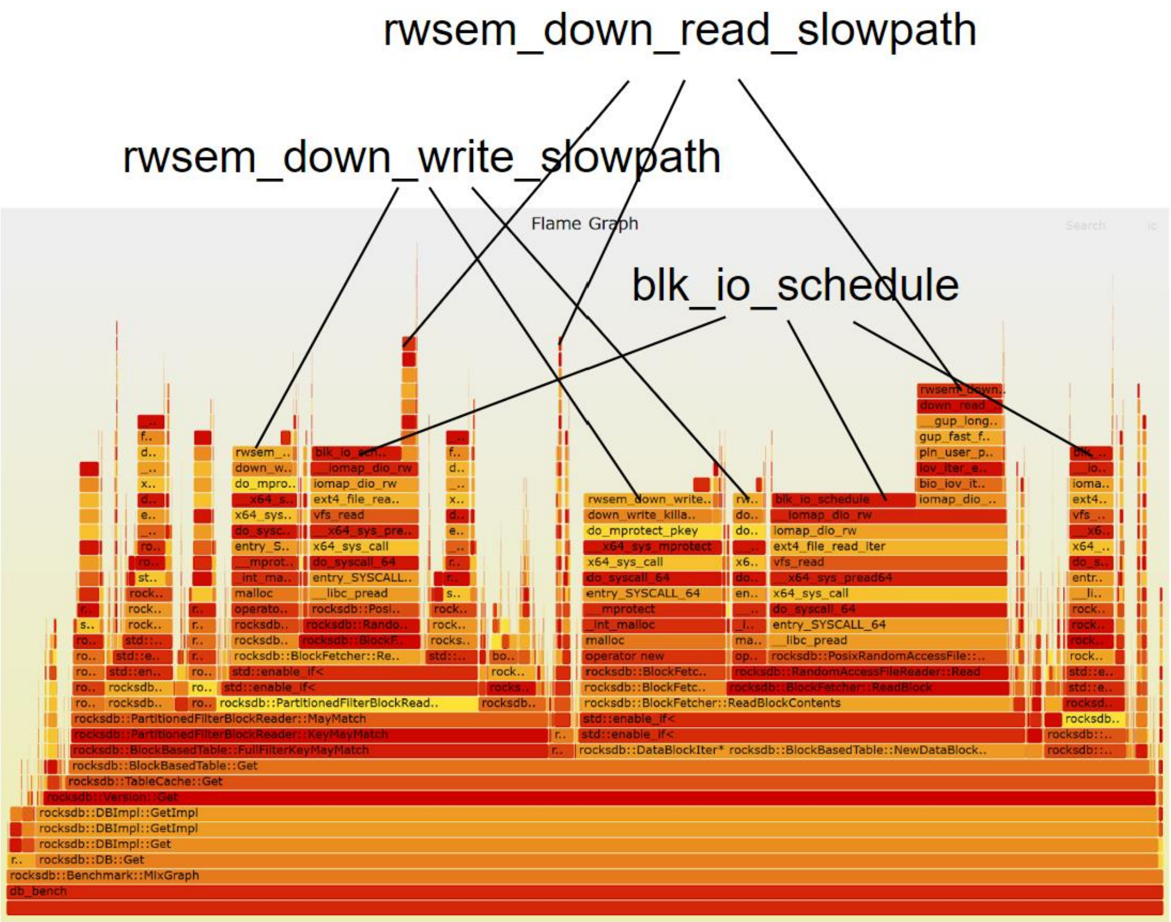
ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

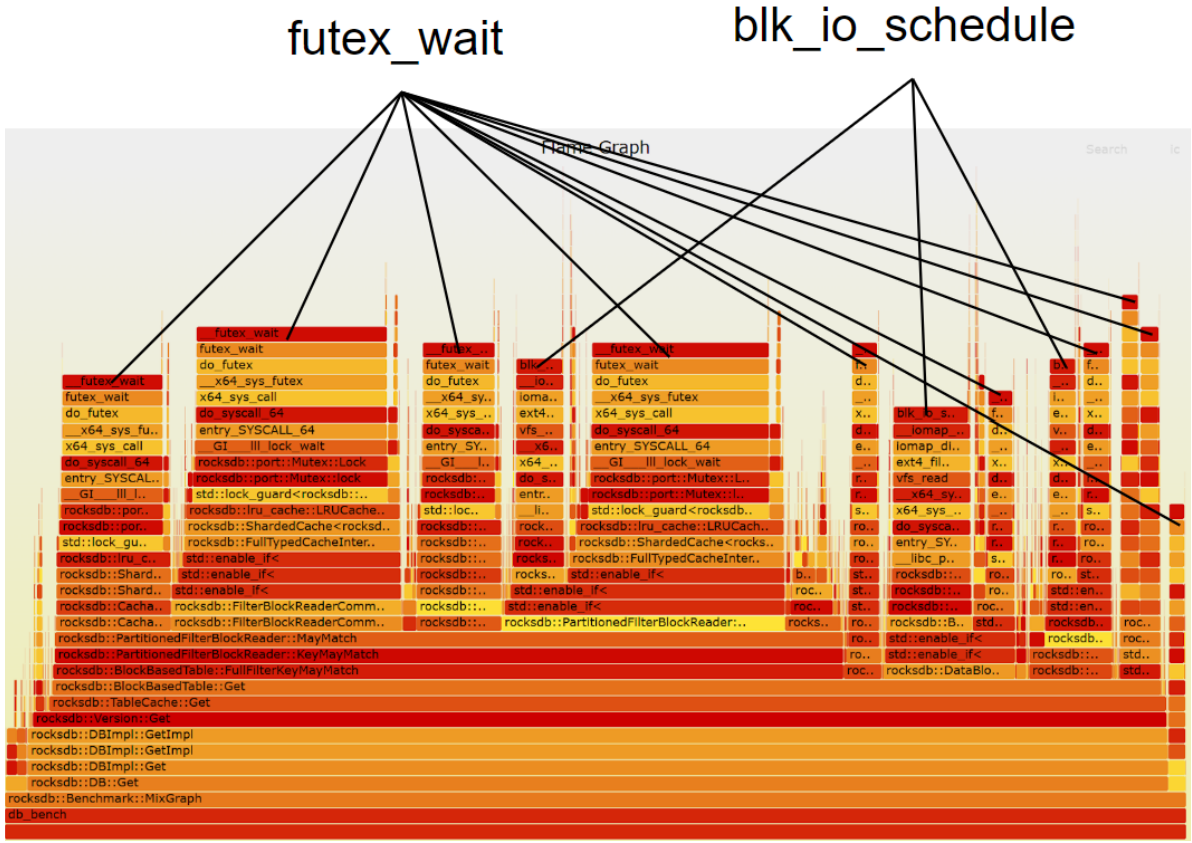
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Appendix: FlameGraph with Blocked Samples

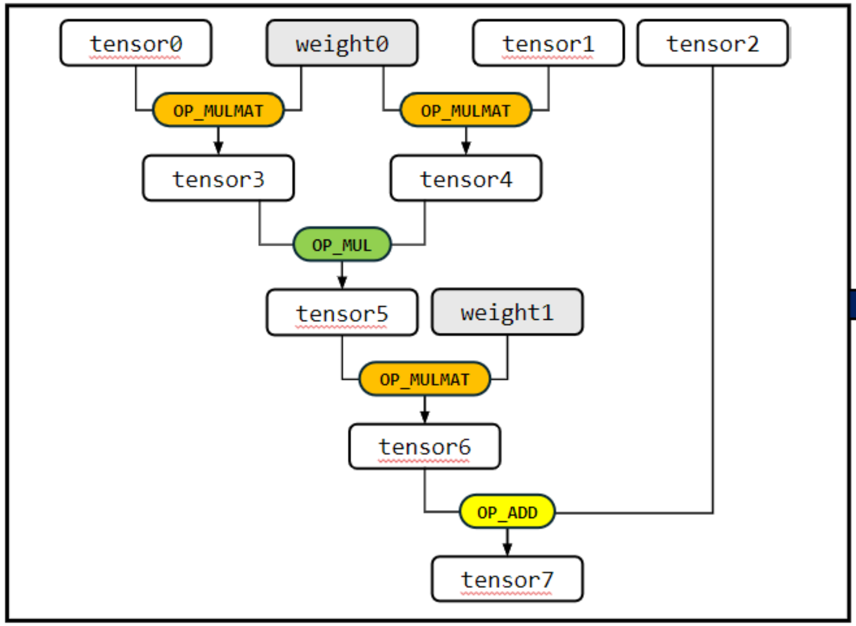
- Callchain visualization of both on-/off-CPU events

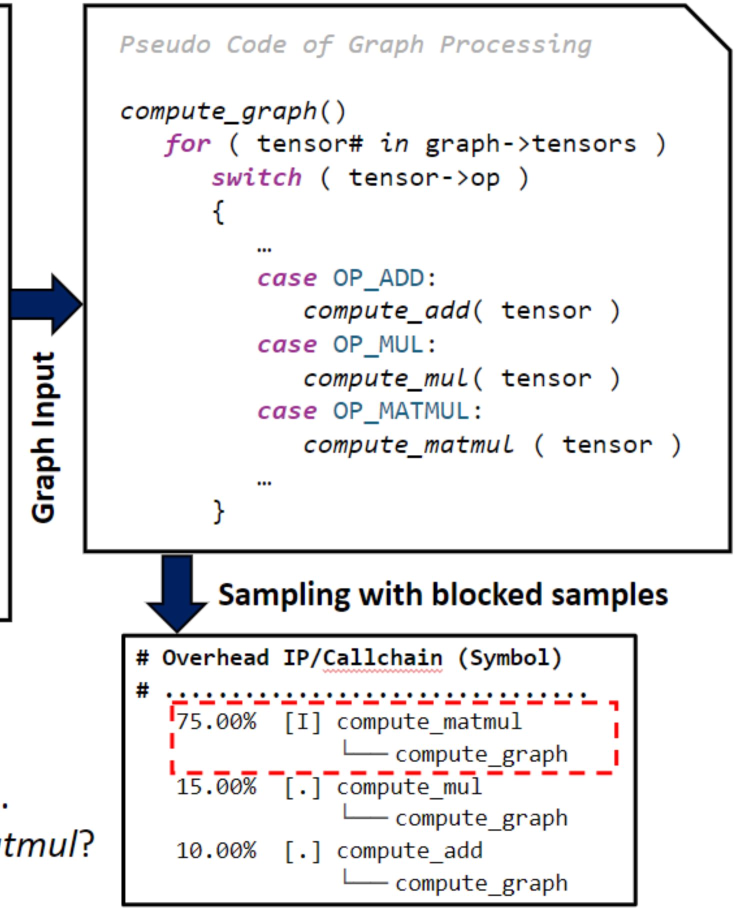




Future Research Questions

- Q1) Does code context is enough to understand bottleneck?
  - e.g., graph-processing applications

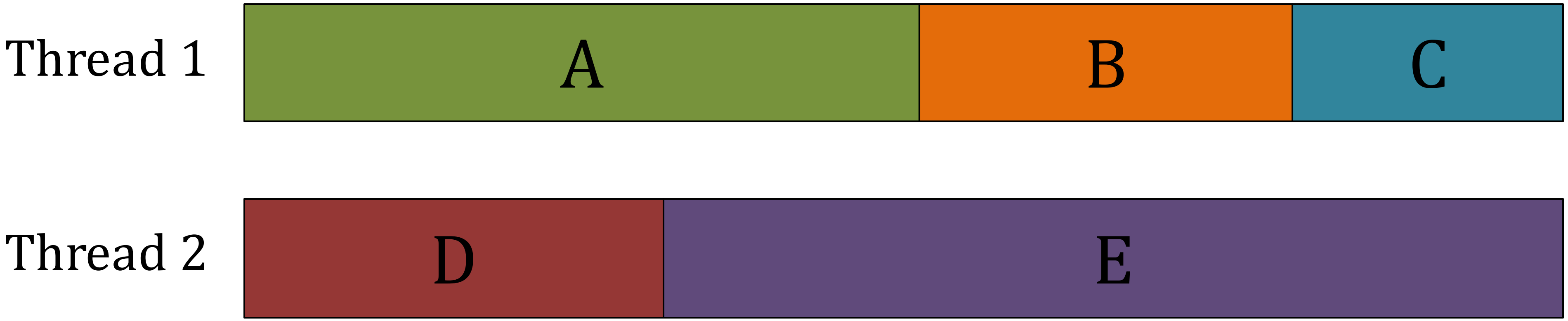




Graph information is missing…
→ **Which tensor** invokes *matmul*?

Future Research Questions

- Q2) What if there is nothing to optimize?



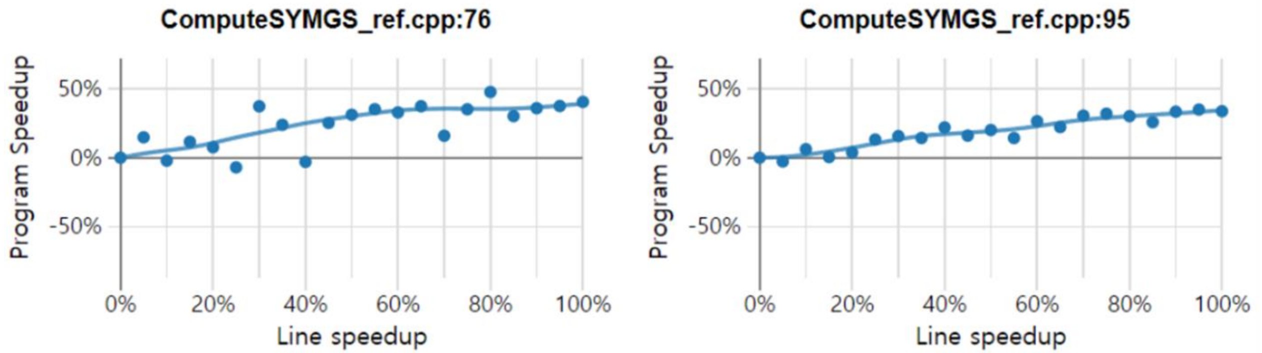
→ Optimizing any single event does not improve performance
→ Does that mean there is no room for further optimization?
→ Optimizing both {A, E} can improve the performance

Appendix

Case Study – HPCG (Serialized SYMGS Kernel)

- Scenario: 64 application threads on 64 logical cores
- Identified bottlenecks: computation
  - ComputeSYMGS_ref (symmetric gauss seidel kernel)
- Needed optimization: parallelize the SYMGS kernel execution

Identified bottlenecks in SYMGS code



<BCOZ>

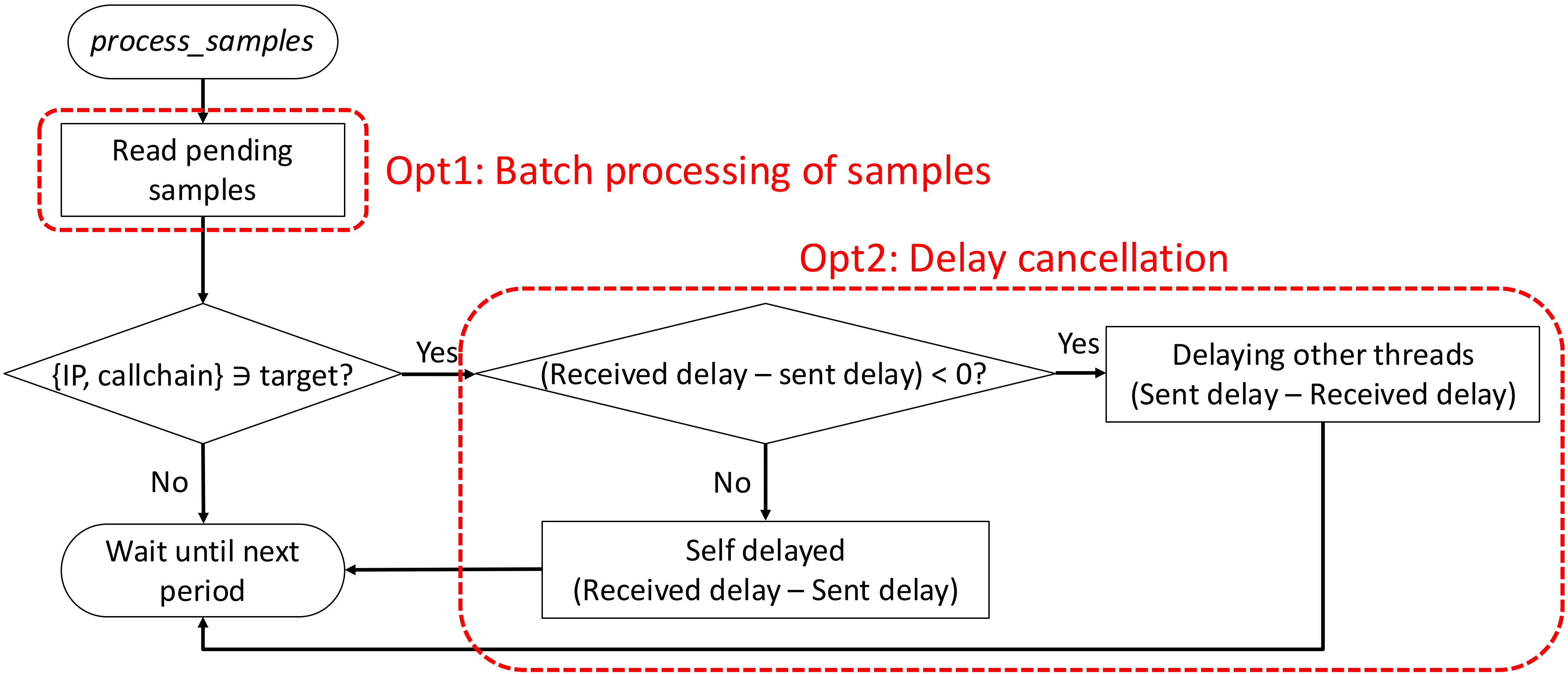
```
74      for (int j=0; j< currentNumberOfNonzeros; j++) {
75        local_int_t curCol = currentColIndices[j];
76        sum -= currentValues[j] * xv[curCol];
77      }
```

```
93      for (int j = 0; j< currentNumberOfNonzeros; j++) {
94        local_int_t curCol = currentColIndices[j];
95        sum -= currentValues[j]*xv[curCol];
96      }
```

<ComputeSYMGS_ref.cpp>

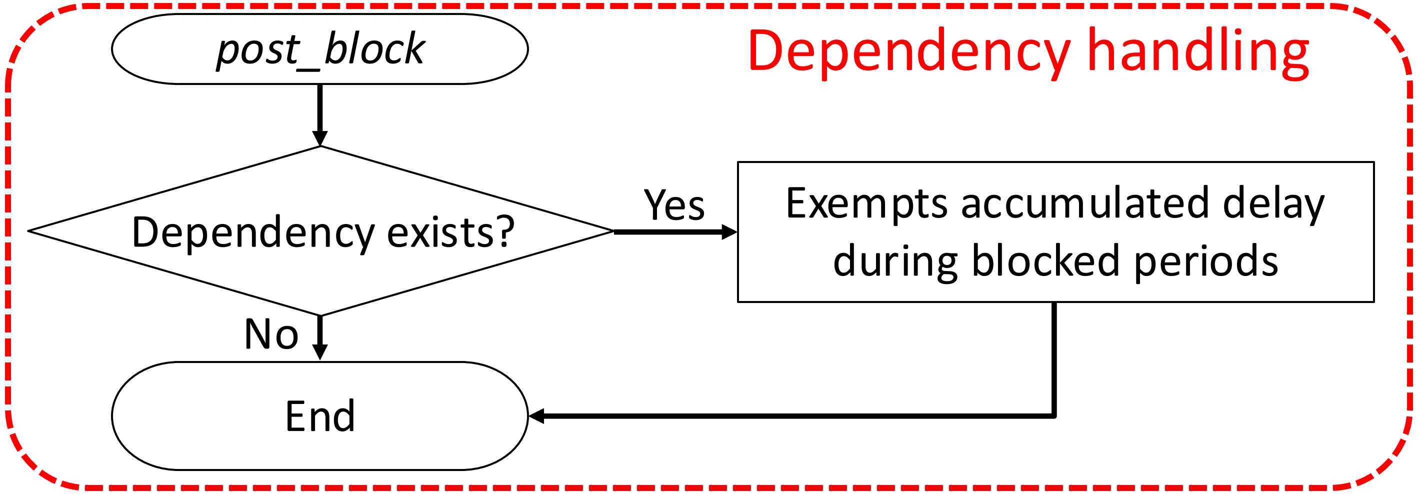
Implementation of COZ

- *process_samples*: periodic virtual speedup operation

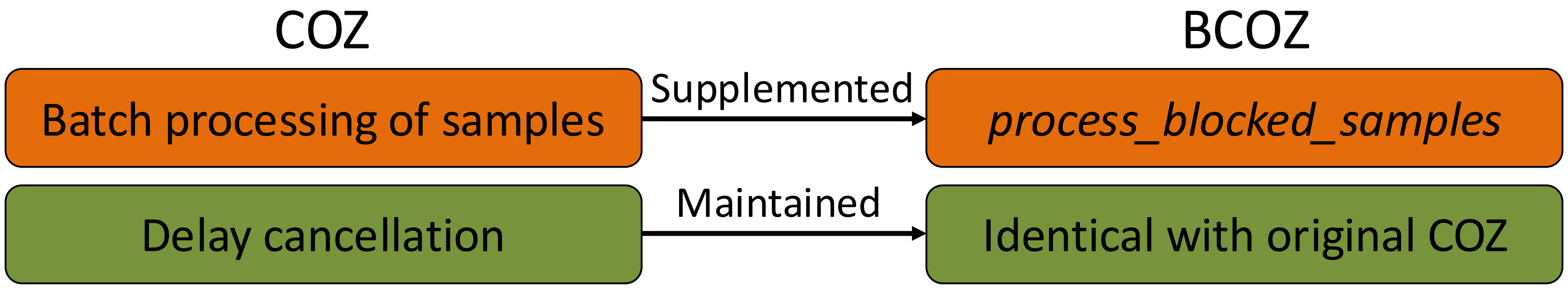


(cont'd) Implementation of COZ

- *post_block*: Delay exemption operation triggered at thread wakeup

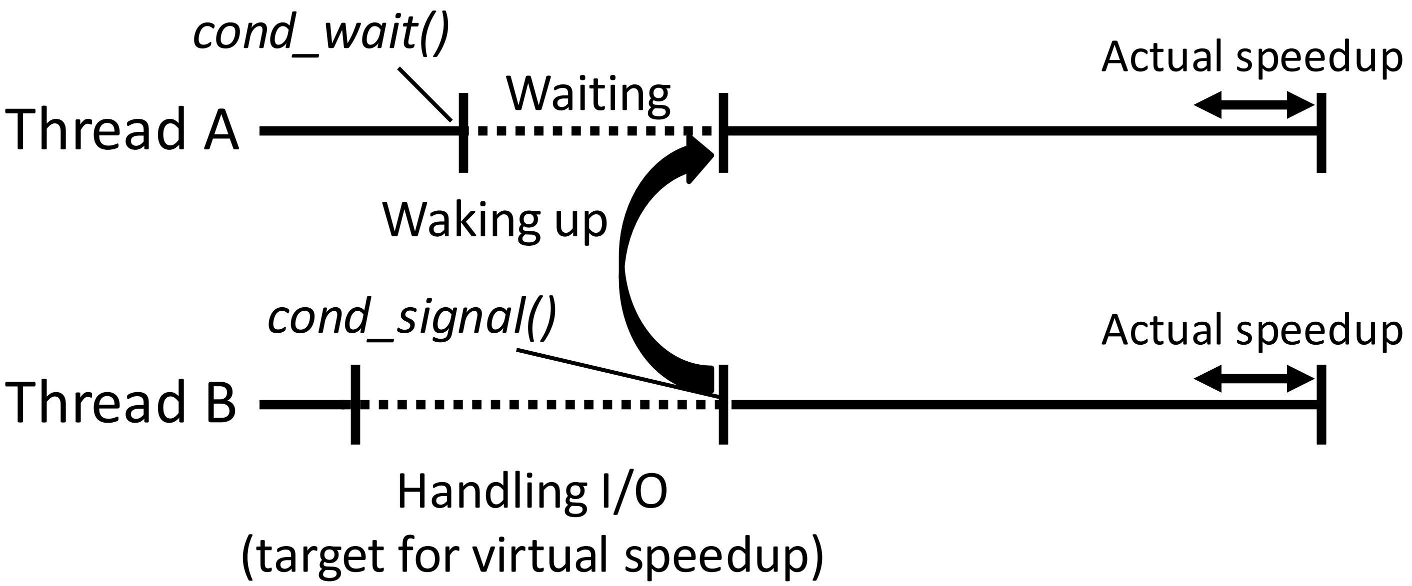


→ However, batch processing of *blocked samples* can compromise the dependency handling



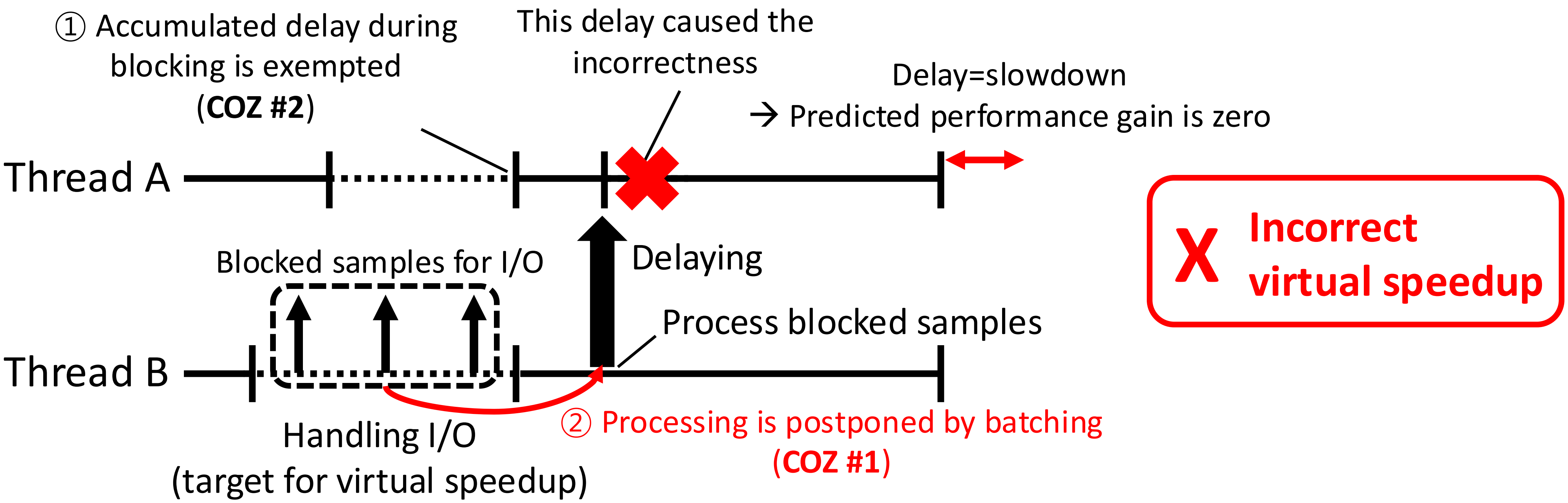
Virtual Speedup of Blocked Samples

- BCOZ handles dependencies between off-CPU events
  - Events with dependencies <u>cannot be sped up independently</u>



Virtual Speedup of Blocked Samples

- BCOZ handles dependencies between off-CPU events
  - Events with dependencies cannot be sped up independently
  - Batch processing of samples can cause inaccurate virtual speedup to occur after wakeup



Virtual Speedup of Blocked Samples

- BCOZ handles dependencies between off-CPU events
  - Events with dependencies cannot be sped up independently
  - Batch processing of samples can cause inaccurate virtual speedup to occur after wakeup
  - BCOZ processes blocked samples immediately when a thread wakes up another thread