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

Minwoo Ahn and Jeongmin Han, *Sungkyunkwan University;*
Youngjin Kwon, *Korea Advanced Institute of Science and Technology* (*KAIST*)*;*
Jinkyu Jeong, *Yonsei University*

https://www.usenix.org/conference/osdi24/presentation/ahn

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

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

Minwoo Ahn[1], Jeongmin Han[1], Youngjin Kwon[2], Jinkyu Jeong[3]
[1]*Sungkyunkwan University,* [2]*KAIST,* [3]*Yonsei University*
{minwoo.ahn, jeongmin.han}@csi.skku.edu, yjkwon@kaist.ac.kr, jinkyu@yonsei.ac.kr

Abstract

The rapid advancement of computer system components has necessitated a comprehensive profiling approach for both on-CPU and off-CPU events simultaneously. However, the conventional approach lacks profiling both on- and off-CPU events, so they fall short of accurately assessing the overhead of each bottleneck in modern applications.

In this paper, we propose a sampling-based profiling technique called *blocked samples* that is designed to capture all types of off-CPU events, such as I/O waiting, blocking synchronization, and waiting in CPU runqueue. Using the blocked samples technique, this paper proposes two profilers, *bperf* and *BCOZ*. Leveraging blocked samples, bperf profiles applications by providing symbol-level profile information when a thread is either on the CPU or off the CPU, awaiting scheduling or I/O requests. Using the information, BCOZ performs causality analysis of collected on- and off-CPU events to precisely identify performance bottlenecks and the potential impact of optimizations. The profiling capability of BCOZ is verified using real applications. From our profiling results followed by actual optimization, BCOZ identifies bottlenecks with off-CPU events precisely, and their optimization results are aligned with the predicted performance improvement by BCOZ's causality analysis.

1 Introduction

Application profiling encompasses the analysis of two types of events: on-CPU events and off-CPU events. Profiling on-CPU events aims to analyze instructions executed on a CPU [1,4,7,15,17,19–21,33,45,53,54]. In contrast, profiling off-CPU events aims to analyze waiting events within an application such as waiting for blocking I/O completion, locks, scheduling, etc [27,35,38,39,55,58].

In the past, applications were clearly characterized as either CPU-boud or I/O-bound due to the use of slow I/O devices such as HDD or SATA SSD. Therefore, existing profiling tools have applied bottleneck analysis techniques separately for on-CPU events or off-CPU events. However, with the recent advancements in fast storage and many-core CPUs, modern applications often exhibit complex behaviors. Especially, their performance bottleneck is combined by on-CPU events and off-CPU events, necessitating the need for comprehensive profiling of both on-CPU and off-CPU events *simultaneously* and capturing their interactions. For example, with the emergence of NVMe SSDs and ultra-low latency SSDs, the critical path of I/O-intensive applications often shifts from I/O to CPU events [23, 30–32]. Consequently, studies have focused on optimizing on-CPU events rather than I/O events to enhance the performance of I/O-intensive applications [23,30–32].

However, existing profilers focus on analyzing only on-CPU [17, 20, 33] or off-CPU events [3, 38], so they cannot analyze the complicated behaviors of modern applications. COZ [15], a state-of-the-art causal profiler, estimates performance gain through its virtual speedup approach. COZ intentionally delays competing threads to estimate the performance impact by optimizing certain code lines without actually optimizing them. However, COZ applies virtual speedup profiling exclusively to on-CPU events as it lacks the capability to incorporate execution information from off-CPU events. wPerf [58], a state-of-the-art off-CPU analysis profiler, traces waiting events between threads during the execution and reports the result in the form of a graph (called a wait-for graph). While wPerf can analyze interactions between on-CPU and off-CPU events, it falls short in assessing the actual impact of bottlenecks on application performance. Furthermore, although wPerf identifies off-CPU bottlenecks, it lacks detailed information about the application contexts related to these bottlenecks. These limitations require programmers to attempt optimizations for various bottlenecks to achieve actual performance improvements and demand additional effort to pinpoint the application code to be optimized (Section 2.2).

This paper introduces a new profiling technique called *blocked samples*, which is designed to capture off-CPU events. Drawing inspiration from the event-based sampling (e.g., Linux perf subsystem [17]), our approach employs sampling-based profiling of off-CPU events. Similar to Linux perf, the blocked samples technique periodically captures snapshots of events with designated information, such as the instruction pointer (IP) and the call stack of each thread. However, unlike Linux perf, which disables profiling while the target thread is blocked, the blocked samples technique overcomes this limitation by allowing sampling of off-CPU events while the target thread is blocked. Leveraging the information provided by blocked samples, we aim to devise profiling techniques that consider both on-CPU and off-CPU events simultaneously, identify bottlenecks and estimate their performance impact if optimized.

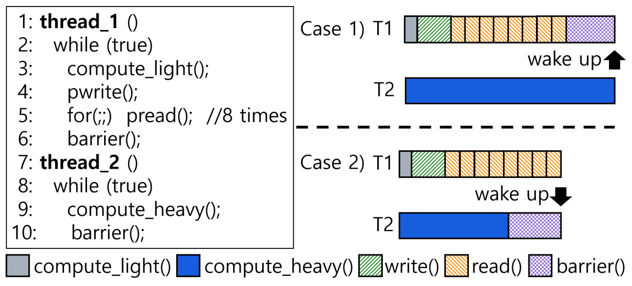
To demonstrate our idea, we devise two profilers, *bperf* and *BCOZ*. bperf is an easy-to-use sampling-based profiler, following the interfaces of the familiar Linux perf tool. bperf incorporates the blocked samples technique. By considering the sampling results that include blocked samples, bperf accurately calculates the overhead of each event. This enables bperf to provide precise overhead information for both on-CPU and off-CPU events. Additionally, bperf reports detailed information about the sampled off-CPU events such as call stacks, kernel stacks, and blocking types (e.g., I/O, synchronization, scheduling, etc.), assisting users in gaining a deeper understanding of overheads.

BCOZ is a causal profiler that leverages the concept of virtual speedup [15] for off-CPU events obtained by blocked samples. BCOZ provides estimated performance improvement not only of a single off-CPU event but also of a function involving multiple on-/off-CPU events. Additionally, BCOZ supports a per-subclass virtual speedup technique, which estimates the potential speedup if off-CPU events of a particular type are optimized, such as using faster I/O devices or eliminating CPU scheduling delays. This feature allows users to consider various optimization alternatives, such as upgrading I/O devices or assigning additional CPU cores.

Our evaluation results demonstrate that BCOZ successfully identifies bottlenecks of real applications including both on- and off-CPU events. Specifically, BCOZ identifies various bottlenecks of the RocksDB key-value store [50] with read-intensive and write-intensive workloads. We observe that with various memory configurations and workload patterns, diverse parts of the program exhibit distinct performance bottlenecks. BCOZ precisely identifies such bottlenecks and provides an estimated speedup of the optimization of each bottleneck. We verify that the reported virtual speedup results are also aligned with the actual speedup when various optimization techniques are applied. These results prove the effectiveness of BCOZ by profiling on- and off-CPU events simultaneously.

Our contributions are summarized as follows:

- We demonstrate the need for integrated profiling of both on-CPU and off-CPU events to overcome the limitations of conventional profilers (Section 2).
- We propose a new sampling technique called blocked samples, designed for capturing off-CPU events. By incorporating blocked samples with on-CPU samples, we enable the identification of application bottlenecks related to both on- and off-CPU events (Section 3.1).
- We introduce two profilers, bperf and BCOZ, that utilize the blocked samples technique. These profilers provide insights into the overhead of off-CPU events and uncover potential performance improvement opportunities that were previously unrecognized (Section 3.2 and 3.3).
- We present practical use cases of our profilers, bperf and BCOZ. Through profiling, we identify off-CPU bottlenecks in applications. Then, we validate the identified bottlenecks through simple optimizations or comparison with previous optimization studies (Section 4).



```
thread_1 ()
  while (true)
    compute_light();
    pwrite();
    for(;;)  pread();  //8 times
    barrier();
thread_2 ()
  while (true)
    compute_heavy();
    barrier();
```

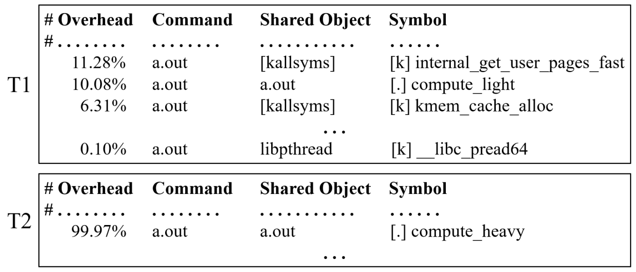
Figure 1: Motivational example of mixed on-/off-CPU events.

2 Background

2.1 Sampling-based Profiling

Sampling-based profiling (e.g., task-clock in Linux perf [17]) is a widely used and efficient method for profiling the application execution with low overhead [17, 20, 33]. It periodically captures (or samples) the execution information of programs on the CPU such as the instruction pointer (IP) and the callchain of each thread (or CPU). With the sampling results, profilers can report statistical overhead [17, 33], perform causal analysis [15], and visualize callchains of captured events [21]. However, identifying the exact bottleneck in sampling-based profiling remains challenging due to the absence of off-CPU events, such as I/O waiting, synchronization, and CPU scheduling.

We illustrate the limitations of profiling without off-CPU events using a simple example program. Figure 1 presents an example of an application involving both on-/off-CPU events. In this example, two threads are executed concurrently and are synchronized through a `barrier` that is implemented using a mutex and condition variable. *Thread 1* executes `compute_light`, which is a small on-CPU computation (iterative integer increment), one 4-KB disk write (`pwrite`), and eight 512-byte disk reads (`pread`). The two types of off-CPU events, hence disk write and reads, are the majority of the execution of Thread 1. *Thread 2* executes `compute_heavy` only, which performs a large on-CPU computation. We adjust the computation load of Thread 2 to make two distinct cases: *Case 1* where Thread 2 is on the critical path and *Case 2* where Thread 1 is on the critical path, as shown in the right



```
    # Overhead  Command   Shared Object  Symbol
    #........   ........  ...........    ......
       11.28%   a.out     [kallsyms]     [k] internal_get_user_pages_fast
T1     10.08%   a.out     a.out          [.] compute_light
        6.31%   a.out     [kallsyms]     [k] kmem_cache_alloc
                                 ...
        0.10%   a.out     libpthread     [k] __libc_pread64
```

```
    # Overhead  Command   Shared Object  Symbol
T2  #........   ........  ...........    ......
       99.97%   a.out     a.out          [.] compute_heavy
                                 ...
```

Figure 2: Sampling-based profiling result of Case 2.

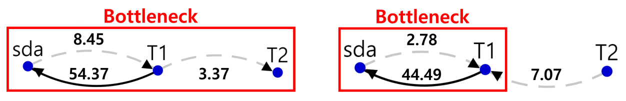
part of the figure. In Case 1, the bottleneck is `compute_heavy`. In Case 2, the bottlenecks are `compute_light` and the I/O events, and their optimization leads to performance improvement. However, in both cases, the optimization of one thread does not improve the performance indefinitely due to the barrier synchronization between the two threads; the critical path changes from one to the other. Hence each optimization has limited global impact on the performance [15, 58].

Limitation of On-CPU Analysis with Sampling. Consider Case 2 where the actual bottleneck is the off-CPU events (i.e., `pread`/`pwrite`), the on-CPU analysis only cannot identify the correct bottleneck. Figure 2 shows the profiling results of Case 2 with the Linux perf tool [17], a popular on-CPU sampling-based profiler. In the figure, each line represents the overhead portion, command name, shared object name, and event symbol; `[.]` and `[k]` indicate user-level and kernel-level symbols, respectively. Due to the tool's inability to capture off-CPU events caused by the blocking I/O events, it only reports the overhead of on-CPU events such as `compute_heavy`, `compute_light` and Linux kernel internal events. The C library function related to `pread` (i.e., `__libc_pread64`) is captured, but it includes only the on-CPU parts, indicating only 0.10% of the overhead. Consequently, the user may examine the results shown in the figure, identify the `compute_heavy` event as the significant overhead among on-CPU events, and focus on optimizing it even though the computation of Thread 2 is not on the critical path. This limitation necessitates off-CPU analysis and causality analysis to precisely identify the performance bottleneck.

2.2 Off-CPU Analysis

Basic Utilities. Basic utilities, such as `top` or `iostat`, can provide information of the overall I/O usage. However, these utilities are primitive and ineffective for detailed analysis of I/O events and correlating those to application contexts, such as IP and callchain.

Off-CPU Tools. Various off-CPU profiling tools [27, 35, 38, 39, 55, 58] exist to support profiling of off-CPU events. However, these profilers are limited in terms of (1) focusing on a specific type of off-CPU events (e.g., syncperf [35]) or (2) providing unsorted information (e.g., off-CPU time distribution [14], call-chains of off-CPU events [40]), which requires



(a) Case 1 in Figure 1. (b) Case 2 in Figure 1.

Figure 3: *wait-for* graph results of Figure 1. For Case 1, the knot includes both I/O and synchronization (`barrier`), while the knot in Case 2 includes only I/O.

a programmer's additional efforts to identify important bottlenecks and their impact on the program performance. For example, off-CPU flamegraph [39] provides and visualizes the overhead of off-CPU events and their callchains. However, the tool does not provide the performance impact of each off-CPU event; an off-CPU event with the largest overhead does not necessarily mean its optimization results in performance improvement to the same extent [15, 58].

wPerf. wPerf [58] is a state-of-the-art profiler that traces all types of waiting events including off-CPU events and reports their relationship in the form of a wait-for graph. Moreover, wPerf identifies bottlenecks by analyzing the global impact of waiting events on other threads. Hence, wPerf reports the performance bottleneck by identifying waiting events for which all the worker threads are waiting to progress; these waiting events are called a knot. Each knot contains a bottleneck.

However, we observe two important limitations when identifying bottlenecks using wPerf. Let us explain the limitations using the example program in Figure 1.

Profiling Result. Figure 3 shows the profiling result of Case 1 and Case 2 in Figure 1 using wPerf. In this figure, the vertices labeled *T1* and *T2* correspond to Thread 1 and Thread 2, respectively, and *sda* represents the disk, hence an I/O device. `sda` $\leftarrow$ T1 indicates that Thread 1 is waiting for completion of disk I/O requests (i.e., `pread` and `pwrite`), and `sda` $\dashrightarrow$ T1 represents the I/O device is waiting for I/O requests from Thread 1 (i.e., `compute_light`) [58]. Moreover, T1 $\dashrightarrow$ T2 (Case 1), and T1 $\dashleftarrow$ T2 (Case 2) indicate the waiting period caused by the synchronization (`barrier`). The numbers on the edges are the global impact of each edge reported by the wPerf which can be interpreted as waiting time.

Firstly, wPerf does not precisely identify bottlenecks and their actual impact on the program performance. In Figure 3a, the red box denotes the knot of Case 1. Hence, Thread 1/2 and `sda` are the bottlenecks. In addition, `sda`, hence the disk, has the largest global impact; the edge has the weight of 54.37 which is an order of magnitude larger than the weight of the other edges. Hence, the profiling result mislead to optimizing I/O events associated with `sda`. However, as shown in Case 1 in Figure 1, optimizing `pread` or `pwrite` does not improve the performance since the real bottleneck is Thread 2 (i.e., `compute_heavy`). Consequently, wPerf's bottleneck identification can be imprecise and can yield a waste of ineffective optimization efforts.

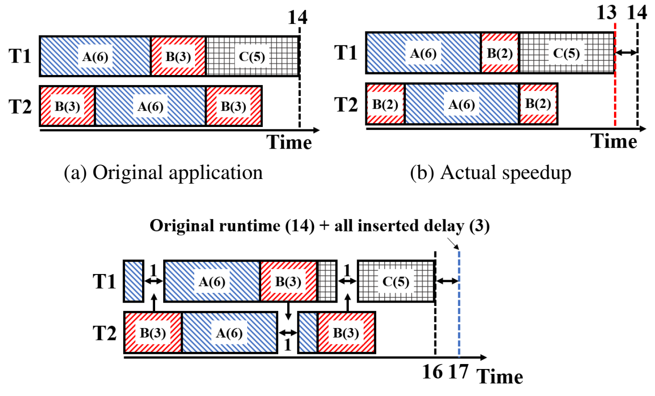
Secondly, wPerf can identify bottlenecks approximately but detailed information or context can be missing, which necessitates programmers' additional efforts to figure out bottleneck points in program codes. In Figure 3b, the knot includes T1 and `sda`. Both are in Thread 1, which are the correct bottlenecks of Case 2 in Figure 1. However, Thread 1 runs three different events, `compute_light`, `pread` and `pwrite`, each of which is on the critical path. As shown in Figure 1, the eight disk reads have the most significant impact on the program performance in Case 2. However, the profiling result of wPerf demonstrates only `sda` has the biggest impact and blurs the precise bottleneck identification among the two off-CPU events. For instance, the programmer may try to optimize `pwrite` but may fail to improve the write performance while the code `compute_light` has a latent performance improvement since the code is implemented inefficiently. Consequently, wPerf's analysis result is less informative and requires additional efforts before targeting events to optimize.

2.3 Causal Profiling

Optimization of bottlenecks identified through conventional profiling does not always guarantee performance improvement [15, 58]. This is especially true for multi-threaded applications since events with significant overhead may not be on the critical path (e.g., `compute_heavy` in Case 2 of Figure 1). Causal profiling [15] is a profiling methodology that estimates the actual impact of optimization on the program performance.

Virtual Speedup. Virtual speedup is a core technique for performing the causality analysis. It offers an estimation of potential performance improvement by virtually speeding up a particular event (e.g., a program code line) [1, 15, 26, 45, 46]. The virtual speedup technique does not require the actual optimization of a particular code line but can assess the causality of the performance optimization. This can be achieved by delaying concurrently running events (or threads). Hence, if a particular event is sped up by a certain amount, it can be simulated by delaying other concurrent events by that amount but not the target event itself.

Figure 4 illustrates an example of the virtual speedup technique by showing the timeline of a two-threaded application. Figure 4a shows the original timeline and Figure 4b shows the actual speedup case when the execution time of function B is optimized from 3 to 2, and as a result the total execution time is reduced by 1. Figure 4c shows the virtual speedup case. The virtual speedup technique speeds up a particular function (B in this case) virtually by delaying co-running threads on every invocation of the target function. This has an effect of which the target function is sped up since other than the function is delayed by the same amount. Hence, in this example, whenever B is called, the other thread is injected a delay of the amount to speed up. The virtual speedup technique measures the speedup by identifying the difference between the actual runtime, 16 in this case, and the expected execution



(a) Original application (b) Actual speedup

(c) Virtual speedup

Figure 4: Illustration of virtual speedup when speeding up B. X(n) means function X runs for n time units.

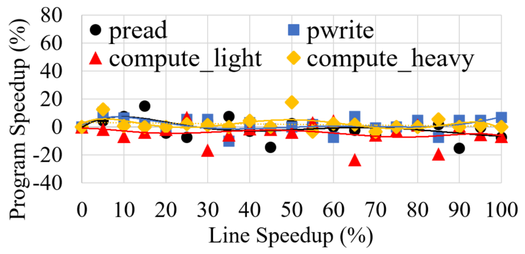
time if function B is not sped up. The expected execution time without speedup is 17, which is obtained by that whenever B is invoked, all the threads are delayed by 1. Hence, the application's virtual speedup is 1 (17 minus 16).

COZ. COZ [15] is an implementation of the causal profiler employing the virtual speedup method. It reports to a user with information about the application's bottlenecks and provides an estimation of performance improvement for each optimization point. To apply the virtual speedup technique, COZ employs sampling-based profiling using the Linux perf subsystem. Periodically, COZ reads sampling results, IPs and callchains of each thread. Then, when the target code line to speed up is being executed, COZ applies the virtual speedup technique by delaying the execution of other threads, hence forcing sleep of co-running threads.

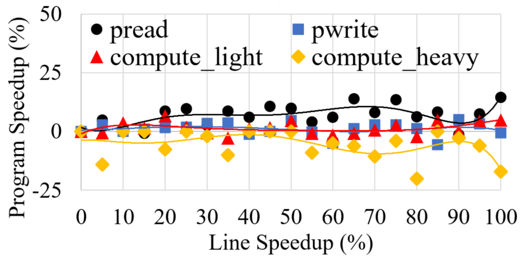
COZ carefully handles dependencies between threads and injects delays between threads with dependency in order to avoid incorrect estimation of virtual speedup. For example, if thread A is woken up by thread B, any injected delays to thread B while thread A is sleeping are considered the injected delays to thread A as well. This is because thread A is woken up after thread B has consumed its injected delays. In other words, if thread A is delayed by the same amount of injected delay after its wakeup, thread A experiences double delays from one source of delay injection. Accordingly, COZ manages dependencies arising from thread synchronization primitives (e.g., mutex, condition variable), and exempts delays during thread wakeups.

COZ employs two optimization methods to enhance its profiling performance. First, COZ processes multiple samples in batches. Second, it tries to skip consuming delays whenever all the threads need to consume the same amount of delay. This helps reduce the profiling time since otherwise, all the injected delays may increase the application's runtime significantly.

Profiling Result. While causal profiling is effective and informative in specifying bottlenecks and providing the estimated



(a) Case 1



(b) Case 2

Figure 5: Virtual speedup result of Figure 1.

outcome of bottleneck optimization, it lacks the capability of considering off-CPU events for profiling, often leading to incorrect profiling results. Figure 5 shows the profiling result of the example program in Figure 1 using COZ. In this figure, the x-axis represents the reduction in the execution time of a particular line, while the y-axis indicates the predicted overall runtime reduction of the program if that specific line speeds up by x%. For instance, a 0% program speedup indicates no performance improvement, whereas a 75% program speedup implies the runtime is reduced by 75%.

In Case 1, the actual bottleneck is `compute_heavy` but COZ fails to identify its potential performance improvement if it is optimized. Actually, COZ identifies marginal virtual speedup of all the four events in Case 1 as shown in Figure 5a. A similar phenomenon happens in Case 2 of the same example program as shown in Figure 5b. These results stem from the fact that COZ does not consider off-CPU events.

The causal profiling is an essential technique considering the ever-increasing complexity of applications on modern computer systems. However, the inability to profile off-CPU events is a critical limitation of existing causal profiling. The CPU performance improvement has stopped due to the end of Moore's law and Dennard scaling, and domain-specific accelerators, such as GPUs, FPGAs, and Smart SSDs, are gaining significant attention [25]. These heterogeneous computing environments make the behavior of applications more complicated with various off-CPU events for offloading computation to such accelerators. In addition, various low-latency I/O devices, such as CXL memory expander [16], RDMA-capable several-hundred gigabit network interface cards [49], flash-based or persistent memory-based solid-state drives [41, 59], are making application behavior increasingly complex. Traditional I/O-boundness or CPU-boundness is no longer a proper

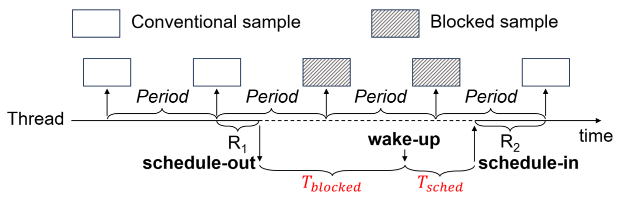


Figure 6: Blocked samples and conventional samples.

application classifier. The mixture of both I/O- and CPU-bound applications require sophisticated performance profiling methodology. Among these, profiling on-CPU and off-CPU events together is especially important for optimizing application performance.

3 Design and Implementation

In this section, we present our methodology to profile on-CPU and off-CPU events simultaneously. Our methodology begins with proposing a new method of sampling off-CPU events, called *blocked samples* (Section 3.1). Then, we present two profilers for accommodating blocked samples: *bperf*, an easy-to-use sampling-based profiler (Section 3.2) and *BCOZ*, a causal profiler that profiles both on- and off-CPU events simultaneously and estimates potential speedup of optimizations (Section 3.3).

3.1 Blocked Samples

Blocked samples captures information of blocking events of threads, such as waiting for I/O completion, waiting for synchronization (e.g., mutex, condition variable), waiting for CPU scheduling, etc. The conventional on-CPU event sampling is thread-oriented (e.g., *task-clock* of the Linux perf subsystem [17]). When a thread executes its instructions on the CPU, the event-based sampling periodically collects a sample, hence snapshot, of thread context (e.g., IP and callchain) as shown in Figure 6. When a thread is blocked, the sampling is paused until the thread is woken up and resumes its execution. Different from the conventional on-CPU sampling, blocked samples augments the missing samples while threads are blocked, providing the execution context of blocking periods (shaded boxes in Figure 6).

Each blocked sample contains four attributes for tracking an off-CPU event: *IP*, *callchain*, *weight*, and *type*. The IP is the address of the last instruction before a thread is blocked. This is actually the return address of invoking a CPU scheduler (e.g., `schedule` or `io_schedule` in Linux). The callchain is the call stack of functions from the main function of a thread to the current instruction before being blocked.

The length of a blocking event can be varied (e.g., a few microseconds for CPU time-sharing or hundreds of milliseconds for disk I/O). Hence, one blocking event can contain multiple blocked samples of the same properties (i.e., identical IP, callchain, etc.). We encode the number of repeats to the *weight* field, saving space and time when handling blocked samples.

The *type* field is used to categorize blocked samples into the following subclasses:

- **I/O:** The I/O subclass corresponds to an off-CPU event that occurs when a thread requests and waits for an I/O event. Specifically, when a current thread submits a synchronous I/ O request and is blocked by invoking a CPU scheduler (e.g., a `task_struct` with `in_iowait` set in Linux), this blocking event is categorized as the I/O subclass.
- **Synchronization:** The synchronization subclass refers to an off-CPU event where a thread is waiting for a lock or condition variable. To identify synchronization subclasses, we slightly modified a process control block (e.g., `task_struct` in Linux) to include a field (`in_lockwait`) that identifies lock waiting. The field is set/cleared when a thread sleeps/wakes using the kernel-supported synchronization primitives (e.g., `futex` of Linux).
- **Scheduling:** The scheduling subclass refers to the off-CPU event that occurs when a thread is runnable but not scheduled on a CPU.
- **Others:** This subclass refers to off-CPU events that does not correspond to the above subclasses. Typically, off-CPU events related to sleeping (e.g., `usleep`) are in this category.

Implementation. Blocked samples are collected by extending the *task-clock* event in the Linux kernel's perf subsystem. The original sampling using task-clock collects samples by using a periodic timer (e.g., high-resolution timer). While a task is running on the CPU, the timer allows periodic sample collection. Meanwhile, when a task is off-CPU, hence blocked, the timer is paused until the task resumes execution.

Blocked samples augment the original task-clock-based sampling by including the blocking state of a thread. As the blocking state of a thread remains unchanged until the thread resumes its execution, our scheme captures the blocking state at the moment a thread resumes, rather than relying on the periodic timer. This is achieved by incorporating three task scheduling-related operations: *schedule-out*, *wake-up* and *schedule-in* functions.

First, when a thread is *scheduled out*, our scheme records the subclass of the thread's blocking, along with a timestamp marking the start of the off-CPU interval. Then, when the thread is *woken up*, our scheme records a timestamp to mark the end of the off-CPU interval. When the thread is eventually *scheduled in*, we record a timestamp and calculate the blocking interval ($T_{blocked}$) and the waiting interval for CPU scheduling (T_{sched}) as shown in Figure 6. For threads that are runnable but remain in the runqueue due to CPU contention, the off-CPU interval has no wake-up timestamp and hence belongs to the scheduling subclass of blocked samples. Finally, in the schedule-in function, if the blocking interval overlaps with one or more sampling points in time, a new sample is created. This sample contains the IP, callchain, weight and type attributes. For example in Figure 6, the two off-CPU events,

```
    # Overhead  Command  Shared Object       Symbol
    # ........  ........ ...........         ......
       64.54%   a.out    libpthread          [I] __libc_pread64
T1     14.94%   a.out    [kernel.kallsyms]   [I] __libc_pwrite64
        5.50%   a.out    a.out               [L] pthread_cond_wait
        2.59%   a.out    a.out               [.] compute_light
                            ...

    # Overhead  Command  Shared Object       Symbol
    # ........  ........ ...........         ......
T2     99.97%   a.out    a.out               [.] compute_heavy
        0.02%   a.out    libpthread          [L] pthread_cond_wait
                            ...
```

Figure 7: bperf sampling results of Case 1 in Figure 1.

$T_{blocked}$ and T_{sched} contain two sampling points. Hence, two blocked samples, one for blocking and the other for scheduling, are collected. If an off-CPU interval does not overlap with any sampling points, no blocked samples are collected. This approach minimizes the overhead of the collection of blocked samples, even with frequent off-CPU events, as the three hook points only perform timestamping.

A single blocking event can encompass multiple sampling points. This means that the blocking event can generate multiple blocked samples. However, since these samples share identical attributes, our scheme avoids replicating them. Instead, it encodes the repetition using the *weight* field of a blocked sample. This approach reduces both the space and time overhead associated with handling blocked samples.

3.2 bperf

bperf is an online profiling tool that profiles applications using sampling-based profiling and provides statistics of sampling results. *bperf* is an extension of the Linux *perf* tool [17] to support blocked samples. Similar to perf, bperf can be an online or offline tool as its sampling-based profiling can be attached and detached at any time while a program is running and its profiling overhead is generally low (1.6%), as demonstrated in Section 4.4.

Basically, treating blocked samples has no significant difference from handling conventional on-CPU samples. Samples are classified using their IP and callchain. Using the information, their statistics are reported such as overhead portion, function symbol, and the object file as shown in Figure 7.

We extend the Linux perf tool to (1) interpret the *weight* field of blocked samples and (2) annotate a subclass to blocked samples. Firstly, when bperf processes blocked samples, the weight field denotes the number of repetitions of the same event. Therefore, this repetition is taken into account when calculating the statistics. Secondly, bperf examines the subclass of blocked samples and annotates their subclass type in the reported result. Currently, bperf uses the following annotations, `I` for the I/O subclass, `L` for synchronization, `S` for scheduling and `B` for the rest. This information enables the user to identify and analyze the performance impact of blocked samples distinguished by each subclass.

Figure 7 shows the profling result of the example program in Figure 1 using bperf. As compared to the results of the original perf sampling (Figure 2), bperf provides on-CPU and off-CPU events together, thereby allowing developers to understand the overhead of various events more precisely. In Thread 1, `pread` incurs the largest overhead, followed by `pwrite`, and `pthread_cond_wait` ranks third. In Thread 2, `compute_heavy` dominates the thread's execution time.

The advantages of using bperf are two-fold. First, bperf can allow in-depth analysis of blocking events and their interactions inside the operating system kernel. For example, `fsync` is a complex operation that accompanies many types of disk writes. Without bperf, only a tiny amount of user-level and kernel-level codes are collected as on-CPU samples. With bperf, various off-CPU samples are collected to allow in-depth understanding of the `fsync` operation, such as write-back of data blocks, waking up and waiting for `jbd2` file system journaling thread, write-back of journal blocks and commit blocks. Consequently, bperf allows profiling of the interaction between kernel services (e.g., `fsync` call) and accompanying off-CPU events (e.g., data-block writes, synchronization with `jbd2`) with the detailed information of their callchains.

Second, the profiling results using bperf can provide the following performance optimization guidelines.

- When profiling results show a substantial overhead attributed to I/O subclasses, this suggests that the application spends a significant portion of time waiting for I/O operations, such as synchronous I/O. To enhance performance under these conditions, upgrading to faster I/O devices could be considered. Alternatively, adopting asynchronous I/O interfaces could help minimize the blocking time associated with I/O operations [43, 58].
- When profiling results attribute a large overhead of scheduling subclasses, the result indicates the application threads are spending a large fraction of time in CPU runqueues waiting for scheduling. An optimization guideline can be (1) adjusting the number of threads [58], (2) allocating more CPU resources to an application [47, 58], (3) pinning threads to cores to avoid the performance noise caused by CPU load balancers [36, 37], etc.
- When the overhead of the synchronization subclass is notable, performance improvement can be attained by optimizing the events executed within the critical section through application analysis [35].

3.3 BCOZ

This section introduces BCOZ, a causal profiler designed to identify performance bottlenecks. BCOZ is an offline tool that is designed to help programmers identify performance bottlenecks and improve the performance of their programs. At its core, BCOZ profiles on- and off-CPU events collected by blocked samples and estimates performance improvement through virtual speedup. BCOZ precisely identifies interactions between on- and off-CPU events with symbol-level information obtained from blocked samples. This section explores the challenges of accurately estimating the virtual speedup of off-CPU events and discusses various features that are useful for analyzing applications with off-CPU events to optimize performance.

Sampling Kernel Codes. Off-CPU events are tightly coupled with the operation of the operating system kernel codes. Such events occur through the use of operating system services, such as blocking system calls, synchronization primitives, and multi-tasking. In particular, applications with a high number of blocked samples tend to include frequent interactions with kernel [8, 9]. Hence, not only the period during which a thread is blocked but also the kernel operations for such kernel services are important for analyzing and estimating the speedup of their optimization. Accordingly, it is necessary to capture samples for kernel operations and support virtual speedup on those. The original COZ captures only user-space samples for its virtual speedup. However, BCOZ captures samples from not only user space but also kernel space to identify the target of virtual speedup. The target of virtual speedup is basically selected as a part of user-space code. Then, every sample includes their callchain. If the sample includes the instruction pointer of the kernel space, the callchain contains both user-space and kernel-space ones because bperf collects both of them. BCOZ considers them as a unified callchain and identifies the target of virtual speedup by traversing the callchain from the kernel space to the user space.

Virtual Speedup of Blocked Samples. Estimating the virtual speedup of blocked samples is the core of the causality analysis of off-CPU events. BCOZ needs special care when processing blocked samples in order to produce correct virtual speedup estimation. Recall that blocked samples are captured when a thread is scheduled in. Such blocked samples are queued to the perf subsystem and are reported to BCOZ. BCOZ handles blocked samples in batches similar to what COZ does. Processing blocked samples indicates that if a blocked sample is the target of virtual speedup, a proper delay is injected to other co-running threads as in COZ [15]. A blocked sample contains a callchain from the kernel space to the user space. Hence, if the user-space callstack contains target code lines to speed up, the blocked sample becomes the target of virtual speedup. For example, if `pread` is the target of virtual speedup, not only on-CPU samples such as library codes or kernel I/O stack but also off-CPU samples for disk I/Os issued by these code lines are the target of the virtual speedup. This is because such on-/off-CPU samples have a callchain whose user-space part contains `pread`.

Additionally, special care is required in handling dependencies during the virtual speedup of blocked samples. As blocked samples are processed after a blocking event, hence an I/O event, is finished, the processing of blocked samples can incorrectly inject delays to threads which have depen-

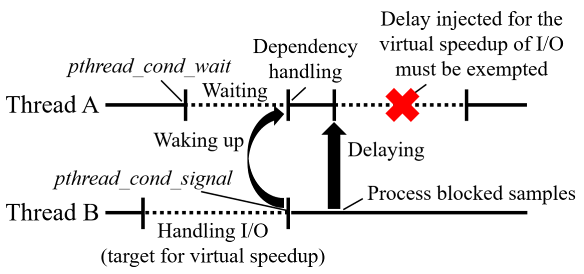


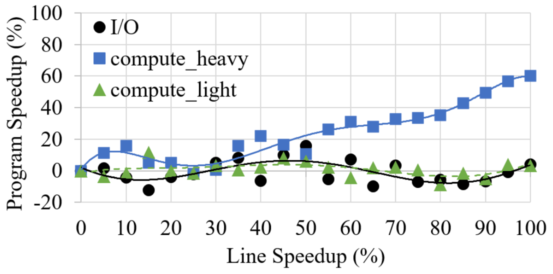
Figure 8: Illustration of virtual speedup when the target includes the off-CPU event while another thread waits for the completion of the target.

dency on the blocking event. For instance, as illustrated in Figure 8, thread A indirectly wait on I/O issued by thread B, and hence after thread B finishes its synchronous I/O, it wakes up thread A. The problem happens if the synchronous I/O becomes the target of speedup. When the blocked samples associated with the I/O are processed sometime after the wake-up operation, the delay injected to thread A, which is for the virtual speedup of the I/O event, makes thread A experience unnecessary delay. In other words, if thread A is injected with the delay, the situation is like the I/O is not sped up. This is because if the I/O event is sped up, the execution of threads waiting for the I/O event directly or indirectly should be boosted as the I/O waiting time is reduced. With the virtual speedup technique, such threads should not be delayed.

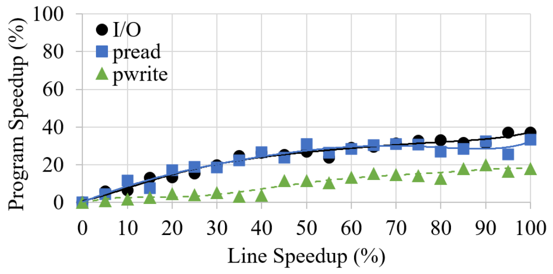
To this end, BCOZ processes blocked samples for virtual speedup before conducting the thread wakeup operations in synchronization primitives (e.g., `mutex_unlock` and `cond_signal`). In the example, the blocked samples of the I/O event are processed before waking up thread A. By doing so, once blocked samples contain the target for virtual speedup, the delays for the virtual speedup are not injected to thread A as thread A is not running. Therefore, the delays are exempted in thread A.

Subclass-Level Virtual Speedup. Optimizing off-CPU events sometimes requires different types of optimization attempts compared to optimizing code. For instance, enhancing the execution environment can often achieve greater performance gains than optimizing the application's code, such as upgrading from HDDs to flash-based SSDs in database applications [51]. Conversely, some applications may exhibit no or marginal performance improvement even when faster hardware devices are employed [23,30,32] due to application-side bottleneck. Similarly, the scheduling subclass off-CPU events have no particular code lines to attempt to optimize. These off-CPU events are challenging to optimize and may require prior knowledge of the optimization effects.

BCOZ provides a subclass virtual speedup technique designed to predict the performance gains from optimizing a specific type of off-CPU events. Hence, the target for virtual speedup is not selected from the application code but from the class of off-CPU events. Applying virtual speedup at subclass



(a) BCOZ results of Case 1 in Figure 1



(b) BCOZ results of Case 2 in Figure 1

Figure 9: Profiling results of Figure 1 using BCOZ.

granularity is straightforward. During the sample processing, instead of checking whether the sample's callchain includes the target code, it checks whether the *type* field of blocked samples (Section 3.1) matches the target subclass.

Please note that the synchronization subclass does not support this subclass-level virtual speedup. From an actual optimization perspective, it is not possible to speed up the lock waiting period itself. Instead, the optimization focuses on speeding up the operations in the critical section. Hence speeding up the lock waiting period is invalid but speeding up the critical section is valid. In other words, the virtual speedup of critical sections can be done when the critical section is selected as the target of speedup. Therefore, we exclude the synchronization subclass from the subclass-level virtual speedup technique.

Selecting Virtual Speedup Targets. BCOZ supports both automated virtual speedup target selection and explicit target designation, similar to COZ. The *automated target selection* conducts virtual speedup for multiple targets in a single run by monitoring frequently sampled code lines during execution and dynamically changing the targets [15]. Where BCOZ differs from COZ is that conducting virtual speedup is not limited to on-CPU events executed by the target code line, but also includes virtual speedup for off-CPU events of the target. Additionally, BCOZ can designate an off-CPU subclass rather than a code line as a target for subclass-level virtual speedup. If a particular off-CPU subclass dominates in the sampling result, users can designate such subclass as a virtual speedup target to assess whether optimizing for that off-CPU event can yield performance improvements.

Profiling Results using BCOZ. Figure 9 provides a summary of the virtual speedup results of the example program depicted in Figure 1. This result illustrates that the actual performance improvements can be achieved by optimizing `compute_heavy` in Case 1 and I/O operations (especially `pread`) in Case 2. Furthermore, the virtual speedup results indicate the actual performance gains are bounded by the point at which the critical path moves. Consequently, by filtering out the false bottlenecks, we can prevent unnecessary optimization efforts for users.

4 Evaluation

In this section, we illustrate the experience of application profiling with bperf and BCOZ. The goal of our evaluation is to answer the following questions: (1) Do bperf and BCOZ identify bottlenecks precisely? (2) Does the estimated virtual speedup align with the actual speedup? (3) As compared to other state-of-the-art profilers (i.e., COZ and wPerf), are bperf and BCOZ more useful?

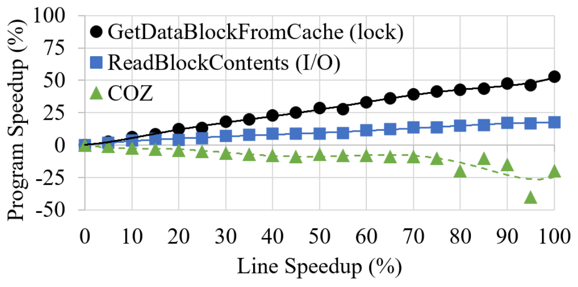
4.1 Experimental Setup

All experiments were conducted on a machine equipped with two Intel Xeon Gold 5218 CPUs (2.30 GHz, 16 physical cores), 375 GB DDR4 DRAM, and a flash-based SSD (PM983), which can deliver performance of up to 540 K I/O operations per second (IOPS). We modified the perf subsystem of the Linux kernel 5.3.7 to support blocked samples, which requires to modification of 295 lines of code. Furthermore, bperf was developed based on the perf tool of the Linux kernel. Finally, BCOZ is implemented based on the existing COZ code [13].

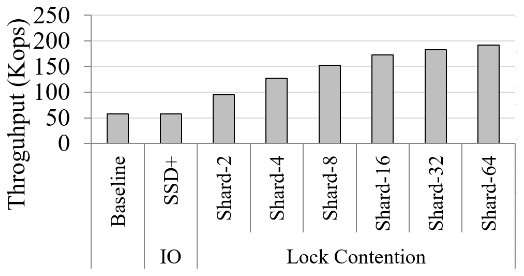
The application codes are complied to include frame pointers to correctly trace callchain. Hence, we disabled frame pointer omitting using `-fno-omit-frame-pointer` option. This is necessary for BCOZ (and COZ [15]).

We present virtual speedup results obtained through either the automated target selection mode or the subclass-level virtual speedup method, as explained in Section 3.3. In the automated target selection mode, the profiler produces virtual speedup graphs for frequently executed lines of code, from which we select the top-N results that demonstrate a positive speedup value. In the subclass-level virtual speedup method, a dominant off-CPU event is manually selected for conducting the virtual speedup profiling.

We measured performance using throughput, defined as the number of processed queries per second, where any performance enhancement reflects as an improvement in throughput. This is based on the assumption that reducing query processing time will naturally lead to increased query throughput, especially in environments where clients continuously submit queries, as in RocksDB. However, as the program speedup in the virtual speedup graphs indicates the percentage of the reduction in execution time, a simple conversion is necessary to translate it to the throughput improvement. A program speedup of y% means that the time it takes to process the same number of queries is now reduced to $(100 - y)\%$ of the original, indicating that the throughput has increased by $\frac{100}{100-y}$ times. For instance, 75% program speedup is translated as 4x throughput improvement.



(a) Causality analysis using BCOZ



(b) Optimization results

Figure 10: Results of (a) causality analysis, and (b) actual optimization in *Prefix Dist*.

4.2 Case Study: RocksDB

In this section, we provide our experience of profiling RocksDB, a widely used log-structured merge (LSM) tree-based key-value store. By profiling RocksDB in various system configurations, we aim to identify off-CPU event bottlenecks that were previously difficult to pinpoint. We also validate these bottlenecks by attempting actual optimizations or comparing them with findings from previous studies on RocksDB optimization.

Optimization 1: Block Cache Contention. As a first optimization, we identify and address the bottleneck of block cache operations in a read-intensive workload. Figure 10 shows the profiling results for read-only execution of *Prefix Dist* [10], an open-sourced real-world workload by Facebook. In this experiment, the key-value size is 91 bytes (48 bytes key and 43 bytes value), the block cache size is 10 GB, the workload runs eight worker threads, and the dataset is set to 1-billion key-value pairs. The workload performs using a single shard to reproduce the well-known lock contention problem of RocksDB's LRU-based block cache [6].

Figure 10a illustrates the virtual speedup results using by BCOZ (solid line) and COZ (dotted line). In the results, two operations, `GetDataBlockFromCache` and `ReadBlockContents` are identified as bottlenecks. The

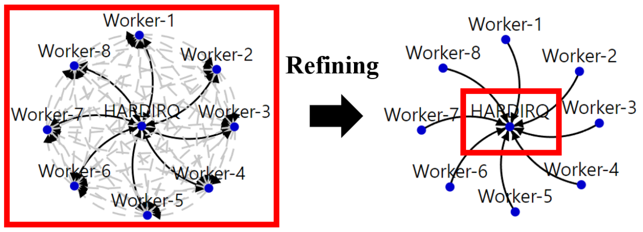
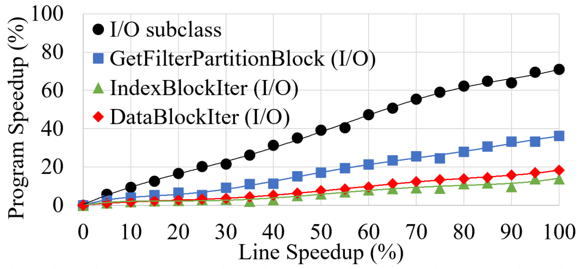
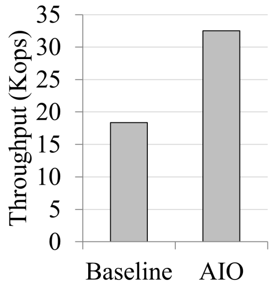


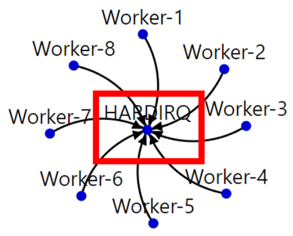
Figure 11: The wait-for graph and identified knots using wPerf



(a) Causality results



(b) Optimization result



(c) Knots in wait-for graph

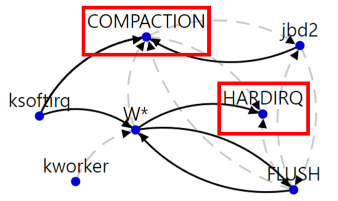
Figure 12: Results of (a) causality analysis, (b) actual optimization, and (c) wPerf in *all random*.

worker threads of RocksDB handles get operations by (1) looking up desired blocks (e.g., filter, index and data block) from the block cache (`GetDataBlockFromCache`) and (2) issuing block I/O requests to underlying disks upon a cache miss from the block cache (`ReadBlockContents`). The cache lookup operation is the real bottleneck since all the workers are contending on the lock of the block cache. As shown in Figure 10a, BCOZ shows up to 60% speedup when the cache lookup operation is optimized and up to 20% speedup when the block read I/O operation is optimized. Although the two operations accompany off-CPU events, COZ does not show any virtual speedup result for the two operations because it cannot take off-CPU events into account for profiling.

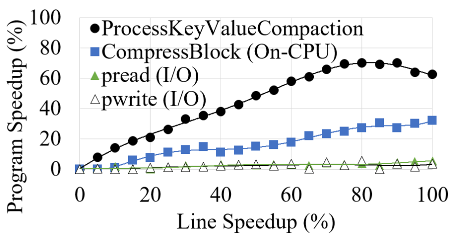
In order to verify the virtual speedup results, we conducted optimization of the two operations. First, we replaced the flash-based SSD with a faster one [44] which delivers performance of up to 1,500 K IOPS; this optimization is denoted as *SSD+*. We expect the speedup of `ReadBlockContents`. Figure 10b, however, shows no performance gain with SSD+ since the lock contention is the major bottleneck. Our second optimization is to apply sharding. The block cache is partitioned to multiple shards, which is denoted as `Shard-N` where N is the number of shards. As shown in the figure, Shard-N shows improved performance. The more shards there exist, the less lock contention occurs thereby showing more improved throughput. This tendency is shown in the virtual speedup analysis of BCOZ in Figure 10a.

To compare profiling capability of wPerf, we use wPerf to analyze the application again. Figure 11 shows the profiling results of wPerf, the initial knot (left) and the knot after trimming edges with a small global impact [58] (right). After trimming the low-weight edges (right), the wait-for graph identifies only the I/O (`HARDIRQ`) as the bottleneck. This profiling result requires a programmer's additional efforts to identify which user-level functions incur such bottlenecks. In addition, improving the I/O performance (SSD+) does not lead to actual performance improvement, which may increase the user's burden of profiling. The wait-for graph before trimming (left) is too complex but provides two bottleneck points: (1) worker --> HARDIRQ edge indicates the I/O waiting of worker threads and (2) worker <--> worker edge indicates lock waiting between worker threads. Therefore, the strategy that the user can try is to optimize the I/O event of the worker thread or to solve the lock contention. However, wPerf does not provide the potential speedup when the bottlenecks are optimized. One lucky programmer may attempt to optimize the lock contention and success to improve the performance. Meanwhile, one unlucky programmer may attempt to optimize the block read I/O operations and may not be able to improve the performance. Therefore, the missing causality analysis of wPerf can increase the burdens of users.

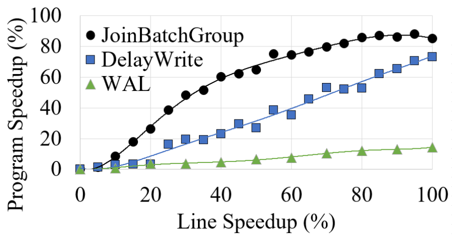
Optimization 2: Block Read Operation. After resolving the block cache contention by sharding, we profiled RocksDB again with the *all random* workload in order to identify and optimize off-CPU events. The configurations of the workload remain unchanged, except that the block cache size is reduced to 128 MB to incur a large amount of off-CPU events, hence disk read I/Os. Figure 12 shows the profiling result of the workload using BCOZ and wPerf. The profiling results of BCOZ demonstrate the three bottleneck points, `IndexBlockIter`, `GetFilterPartitionBlock`, `DataBlockIter`, which handle index blocks, filter blocks and data blocks, respectively. All the operations are off-CPU-intensive operations. In addition, the I/O subclass-level virtual speedup is applied and depicted as `I/O subclass`. As shown in the figure, each of the three operations shows from 15% to 40% of speedup and their aggregated speedup is more than 70% (I/O subclass). Among the three operations, `GetFilterPartitionBlock` shows the largest expected speedup. An LSM tree has multiple levels and a key-value entry can exist in any of the levels. In each level, a key-existence test is done using a bloom filter. Therefore, `GetFilterPartitionBlock` performs the key existence testing using the filter blocks that need to be fetched from the disk. Since a single get operation traverses multiple



(a) Knots in the wait-for graph of wPerf.



(b) Compaction thread.



(c) Worker thread.

Figure 13: Results of (a) wPerf, (b) compaction thread, and (c) worker thread in *fillrandom*.

levels of the LSM tree, optimizing the filter read I/O operation is expected to show the largest performance gain.

Our optimization strategy is to apply asynchronous I/O to fetch filter blocks of the next levels of the LSM tree. Hence, for a get operation, our optimization performs a read of the filter blocks speculatively through the LSM tree. Hence, while the first level is processing the key existence test, the filter blocks of successive levels are speculatively fetched thereby reducing the read I/O waiting time. Our experimental results show that our optimization results in 77.3% of performance gain as shown in Figure 12b.

We compared the profiling result of BCOZ with wPerf. The wait-for graph of wPerf is shown in Figure 12c. As shown in the figure, the knot is `HARDIRQ`, which denotes the disk. However, wPerf provides no more information on the bottleneck. Hence, the user's additional effort is necessary to specify the bottleneck. In addition, the lack of the causality analysis causes users to hesitate about which of the I/O operations to speed up. However, BCOZ provides the causality analysis to off-CPU events and illustrates that the filter I/O operations have the largest potential speedup. This analysis is followed by successful optimization of the filter I/O operations by parallelizing filter I/O operations by using asynchronous I/O.

Optimization 3: Write-intensive Workload. Our next optimization attempt is the write-intensive workload, *fillrandom* against RocksDB. For this experiment, we utilized a key-value size of 1 KB, with 16 worker threads concurrently writing a total of 10 million records. Figure 13 shows the profiling result using BCOZ and wPerf. First, wPerf identifies two potential bottlenecks: the `COMPACTION` thread and the `HARDIRQ` I/O thread. The initial wait-for graph was too complex and we applied the *merging similar threads* technique [58] to obtain the wait-for graph of the figure. From the wait-for graph, we can identify two bottlenecks: (1) the compaction operation of the LSM tree and (2) the write I/O operations of the worker threads (`W*`, the merged worker threads). Since the worker threads perform write-ahead-logging (WAL), WAL write I/Os can be the bottleneck. For the compaction, wPerf does not specify among the operations of compaction the significant overhead. In addition, wPerf reports that `HARDIRQ` shows a bigger global impact than `COMPACTION`.

Meanwhile, BCOZ provides more informative analysis

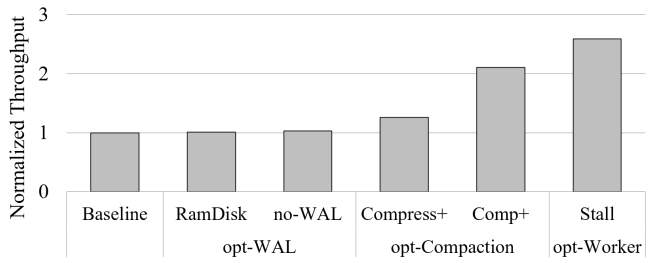
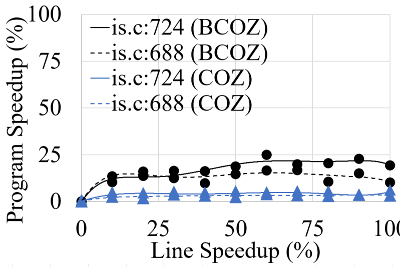


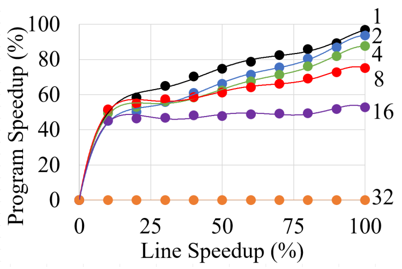
Figure 14: Results of actual optimizations in *fillrandom*.

results. First, BCOZ identifies the compression operation (`CompressBlock`), which is on-CPU, as the significant bottleneck as shown in Figure 13b. Although BCOZ can estimate the potential speedup of I/O operations (`pread` and `pwrite`), their speedup is expected to be marginal. Second, BCOZ expects marginal speedup for the WAL operations as shown in Figure 13c. Third, BCOZ analyzes that the contention between the worker threads (`JoinBatchGroup`) is the bottleneck and its optimization can potentially improve the performance. Finally, BCOZ identifies the write throttling (`DelayWrite`) of the RocksDB's memtable write policy as the bottleneck.

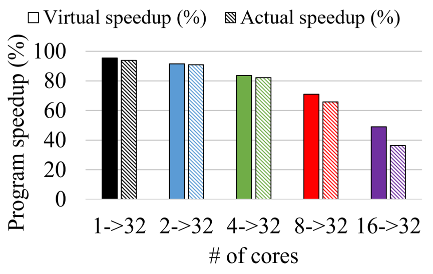
The actual optimizations of the identified bottlenecks are performed as follows. First, the WAL operation is optimized by (1) using a *RamDisk* as the WAL storage and (2) disabling WAL (*no-WAL*). Second, the compaction operation is optimized by (1) disabling compression (*Compress+*) and (2) allocating many compaction threads (*Comp+*). Figure 14 shows the performance of the RocksDB *fillrandom* workload when the optimizations are applied. Third, the writeback stall is relieved by increasing the number of maximum memtables from 2 to 16 (*Stall*). As shown in the figure, WAL-related optimizations show marginal performance gain. This result indicates that BCOZ estimates the potential speedup correctly. In addition, *Compress+* and *Comp+* show improved performance, which is also predicted by BCOZ. Also, *Stall* shows the largest performance gain, which is estimated by BCOZ in Figure 13c. In the meantime, wPerf only identifies `COMPACTION` and `HARDIRQ` as the potential bottleneck points. `HARDIRQ` has shown marginal performance since worker threads are reported to wait for the I/O thread but accelerating WAL opera-



(a) Line-level virtual speedup



(b) Subclass-level virtual speedup



(c) Virtual speedup vs. actual speedup

Figure 15: Virtual speedup results of NPB-is under the CPU contention.

tions show no performance gain. From these results, BCOZ is effective in specifying performance bottlenecks.

Profiling Accuracy. Although BCOZ is proven to effectively identify performance bottlenecks, predicting the potential speedup accurately is sometimes difficult and intricate. For example, BCOZ predicts that optimizing the block cache lock contention can lead to 50% of program speedup (or doubling the throughput) as shown in Figure 10a. However, after relieving the block cache contention by applying sharding, the performance has been improved by four times, which is more than two times higher than the expected improvement. This is because sharding not only relieves the lock contention of cache lookup (`GetDataBlockFromCache`) but also reduces the contention of other operations (e.g., `PutDataBlockToCache`). Furthermore, certain optimizations may speed up a target code line but increase the amount of other events. For example, BCOZ predicts that optimizing `CompressBlock` will result in a 32% program speedup (or 1.47x throughput improvement) in Figure 13b, but the actual throughput improvement is 1.26x. This is because, by disabling compression, `CompressBlock` is no longer invoked, but this optimization has the side effect of increasing the total amount of disk I/Os.

Therefore, virtual speedup results can under- or over-estimate actual speedup. However, the strength of BCOZ (and COZ) lies in their ability to precisely identify specific lines that could potentially lead to actual speedup when optimized. Our evaluation results have shown that optimizing these predicted lines can indeed lead to actual speed improvements.

Takeaway. During the profiling of RocksDB, we have identified bottlenecks that align with those mentioned in existing RocksDB optimization studies. Firstly, we examined the overhead and virtual speedup results of the compaction, which involved a mix of off-CPU events (I/O subclass) and on-CPU events. The potential for performance improvement through compaction optimization, as indicated in numerous studies [2,12,28,42,48], was validated using bperf and BCOZ. The significance of the proposed profiling techniques has become evident as the existing Linux perf tool or COZ could not easily or accurately predict the bottlenecks in the absence of blocked samples. Furthermore, the necessity to optimize operations related write stalls and batched group writing was also identified by BCOZ. These operations have been also identified as bottlenecks in previous studies [11,18,24,29,34,52,56]. Finally, we have examined the bottleneck caused by I/O subclass off-CPU events during the execution of RocksDB and validated the performance improvement.

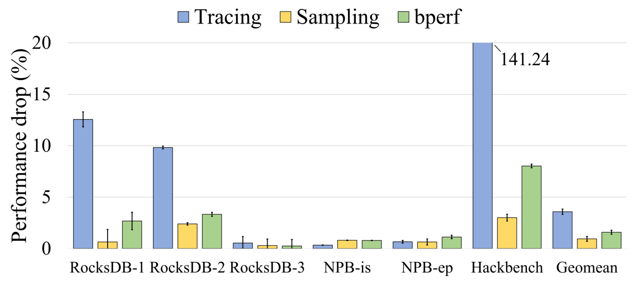
4.3 Case Study: NAS Parallel Benchmark

In this section, we evaluate the effectiveness of the scheduling subclass and subclass-level virtual speedup. For this experiment, we use a compute-intensive workload, *is* (integer sort) from the NAS parallel benchmark [5].

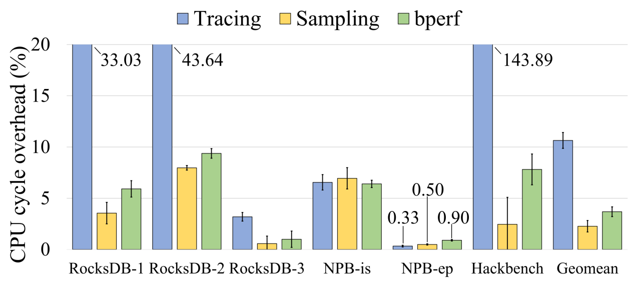
As discussed in Section 3.3, virtual speedup of application code lines can be ineffective if application threads are contending on the CPU cores. To demonstrate this situation, we intentionally controlled the number of CPU cores assigned to the NPB-is workload. The workload is configured to run 32 threads and the number of cores is varied from 1 to 32 cores. Figure 15a illustrates the profiling result of the main computation code lines using COZ and BCOZ when the number of cores is limited to one. As shown in the figure, COZ has estimated marginal performance improvement. On the contrary, BCOZ has predicted potential performance improvement if these code lines are optimized. Under high CPU contention (32 threads vs. 1 core), off-CPU events are frequent as the threads are frequently scheduled out due to the high CPU contention. In this case, BCOZ is able to estimates the optimization opportunity that when such scheduling subclass off-CPU events are removed, the performance can be improved.

Figure 15b presents the profiling results for the scheduling subclass-level virtual speedup as the number of cores increases from 1 to 32. With the highest CPU contention (using only one core), the estimated program speedup is at its peak. As the number of cores increases, thereby reducing CPU contention, the estimated program speedup decreases. Since the number of assigned CPU cores is fewer than the program's 32 threads, these profiling results seems valid.

To validate the virtual speedup profiling results, we measure the program performance with varying the number of cores. This approach reflects the optimization strategy of allocating additional CPU cores to mitigate CPU contention. Fig-



(a) Performance overhead



(b) CPU cycle overhead

Figure 16: Overhead analysis results of bperf.

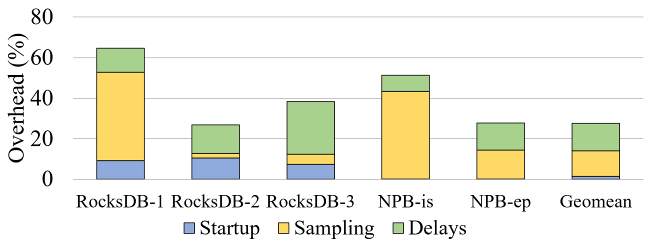


Figure 17: Overhead breakdown results of BCOZ.

ure 15c shows both the virtual and actual program speedups as the number of cores changes from X to 32 (X->32), transitioning from high CPU contention to no CPU contention. As shown in the figure, the actual speedup generally corresponds with the speedup predicted by BCOZ. These results confirms that it is important to correctly profile off-CPU events in highly parallel workload, and BCOZ leverages blocked samples to provide valuable profiling results to users.

4.4 Profiling Overhead

bperf. We compare the overhead of bperf with existing profiling techniques, (1) *tracing* which profiles only off-CPU events (`sched_switch` and `sched_wakeup`) using Linux perf's tracing mode [17, 38, 58], and (2) *sampling* which samples only on-CPU events using Linux perf's sampling mode (`task_clock`) [17]. All these tools have the same goal of profiling statistical overheads of target programs, but their profiling coverage are different: *tracing* focuses on off-CPU events, *sampling* focuses on on-CPU events, and *bperf* covers both on- and off-CPU events. For the sampling methods, the sampling period is set to 1 ms, which is identical in all the experiments in the evaluation section.

Figure 16 shows the overhead of the three profilers with the workloads used in the experiments as well as additional workloads, *NPB-ep* [5] and *hackbench* [22] to cover on-CPU-intensive and off-CPU-intensive cases, respectively. Among the workloads, *RocksDB-X* indicates the RocksDB workload used in Optimization-*X* in Section 4.2. We measured two types of overhead. First, the performance overhead refers to the performance drop of profiled applications (runtime increase for hackbench and throughput decrease for the rest of applications) as compared to the baseline, which runs without any profilers (Figure 16a). Second, the extra CPU overhead refers to the additional CPU cycles consumed by using the profilers as compared to the baseline (Figure 16b). The values shown in the figures are the average of 10 runs.

Overall, the three profilers show acceptable performance drops, no more than 4% on average. Tracing, sampling and bperf have shown the average performance drop by 3.6%, 0.9% and 1.6%, respectively. Specifically, tracing shows notable performance drops with the workloads showing frequent off-CPU events (i.e., RocksDB-1, 2 and hackbench). Tracing records profiling information (i.e., IP and callchain) on every thread state transitions, leading to high overhead when off-CPU events are frequent. In contrast, bperf hooks into all the state transitions like tracing but only records timestamps for each transition. Profiling information is recorded only if an off-CPU interval overlaps with sampling points, resulting in low profiling overhead.

As compared to sampling, bperf enables profiling of off-CPU events at a low cost. bperf shows the additional performance overhead by only 0.7% (1.6% for bperf – 0.9% for perf) and extra CPU cycles by 1.4% (3.7% for bperf – 2.3% for perf). Considering that the profiling capability of bperf is greater than perf, we believe these overheads are acceptable.

As the mechanism of bperf inherits from perf, it can be attached and detached at any time while the profiling target application is running. As the overhead of bperf is low, we believe it can be an online tool for collecting statistical overheads of applications in production.

BCOZ. Figure 17 illustrates the overhead of profiling applications with BCOZ. The BCOZ overhead is categorized into three parts: startup, sampling, and delays. Startup represents the overhead of collecting debug information, which is the duration between BCOZ's bootstrapping and the application's main function (i.e., `libc_start_main()`), on average 1.4%. Sampling encompasses the overhead incurred when there's no virtual speedup delay (i.e., 0% line speedup) consisting of the bperf's sampling overhead and BCOZ's intervention to read samples and verify whether they include speedup targets, on average 12.6%. Finally, delays entail additional overhead when BCOZ is fully enabled, averaging 13.6%. The delays overhead does not refer to the amount of delays injected for virtual speedup, but the increase in the end-to-end execution time of the application.

The overhead of BCOZ is not light considering its end-to-end overhead of 27.6% on average and up to 64.7%. However, such profilers show similar performance overheads. For example, COZ has demonstrated its overhead of 17.5% on average and up to 65% [15]. As these profilers insert additional delays while the application is running, the end-to-end execution time can be increased. These overheads can be reduced when the inserted delays are limited, cooling-off times are inserted between virtual speedup experiments, etc [15]. Such remedies can be effective in reducing the profiling overhead, but it remains uncertain whether the results provided are sufficient to identify bottlenecks and their potential for performance gain.

5 Related Work

On-CPU Event Profilers. Conventional profilers [17, 19–21, 33] that rely on existing on-CPU events (such as CPU usage and execution time) face challenges when identifying bottlenecks in modern applications. This is primarily because the event with the longest execution time in a multi-threaded application does not necessarily represent the critical performance path, and they do not account for off-CPU events. In the context of multi-threaded applications, there are causal profiling studies aimed at analyzing the impact of optimizing each individual event on overall application performance [1, 4, 7, 15, 45, 53, 54]. COZ [15], for instance, provides performance improvement predictions by applying virtual speedup to each event using the sampling results from the Linux perf subsystem. However, COZ's virtual speedup is limited to on-CPU events sampled by the Linux perf subsystem. It is not capable of estimating virtual speedup of events that include off-CPU events.

Off-CPU Event Profilers. Existing studies have focused on analyzing off-CPU event bottlenecks [27,35,38,39,55,57,58]. Some studies analyze application bottlenecks by measuring the duration of off-CPU events [27,39,55]. However, in multi-threaded applications, the longest event may not always represent the critical path of the application [15, 58]. Furthermore, nested off-CPU events can have varying performance impact on event duration and overall application performance [58]. Therefore, analyzing performance using the duration of off-CPU events leads to incorrect conclusions.

Other studies identify application bottlenecks by specifically targeting off-CPU events related to synchronization [35, 57]. However, as mentioned earlier, the off-CPU bottlenecks in modern applications are diverse and encompass various aspects, including device I/O. Therefore, relying on profiling specific off-CPU events has limitation of supporting the various applications.

wPerf [58] is a state-of-the-art study focused on analyzing off-CPU bottlenecks in applications, wait-for graphs are constructed to identify off-CPU events that act as bottlenecks. However, as discussed in Section 2.2, wPerf has several limitations. wPerf does not precisely pinpoint the performance bottleneck of applications. Also, wPerf lacks the capability of causality analysis, so it could not analyze the actual impact on application performance when optimizing a performance bottleneck. Finally, wPerf identifies bottlenecks but misses detailed information, requiring additional efforts from developers to understand the exact performance bottleneck.

6 Conclusion

Existing profilers face limitations when it comes to identifying modern application bottlenecks that involve a mix of on- and off-CPU events. These profilers treat on- and off-CPU events as separate dimensions, making it difficult to perform comprehensive profiling and interpret the results. Moreover, even if the bottleneck of an application is identified, it remains uncertain whether optimizing the bottleneck will result in actual performance improvements. To address this problem, this paper introduces a sampling technique called blocked samples, which enables the identification of application bottlenecks by integrating on- and off-CPU events within the same dimension. We present bperf, a Linux perf tool that utilizes the proposed blocked samples technique to identify application bottlenecks based on event execution time, and BCOZ, a causal profiler that offers a virtual speedup for off-CPU events. By profiling the RocksDB application using these two profilers, we are able to uncover previously unidentified bottlenecks related to I/O and synchronization tasks. Furthermore, by virtually speeding up these tasks, we identify optimization possibilities that were overlooked in existing RocksDB optimization studies.

We plan to extend blocked samples to include richer information for profiling. The current blocked samples consider the operations inside an I/O device as a black box. However, I/O devices may have their internal operations, which can be the hint of performance optimization opportunities for applications. For example, disk-internal events, such as garbage collection, and valid page copying, are important events for storage applications to establish their optimization strategies. In this regard, we plan to augment blocked samples with I/O device-internal operations thereby allowing applications to employ expanded optimization strategies.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, for their valuable comments and feedback. This work was supported in part by Samsung Electronics and by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS-2023-00321688). Jinkyu Jeong is the corresponding author.

Availability

The source code is available at https://github.com/s3yonsei/blocked_samples.

References

[1] Minwoo Ahn, Donghyun Kim, Taekeun Nam, and Jinkyu Jeong. Scoz: A system-wide causal profiler for multicore systems. *Software: Practice and Experience*, 51(5):1043–1058, 2021.

[2] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. Silk: Preventing latency spikes in log-structured merge key-value stores. In *USENIX Annual Technical Conference*, pages 753–766, 2019.

[3] BCC (BPF Compiler Collection). https://github.com/iovisor/bcc/tree/master.

[4] Zachary Benavides, Keval Vora, and Rajiv Gupta. Dprof: distributed profiler with strong guarantees. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–24, 2019.

[5] NAS Parallel Benchmarks. Nas parallel benchmarks. *CG and IS*, 2006.

[6] Block Cache. https://github.com/facebook/rocksdb/wiki/Block-Cache.

[7] Nader Boushehrinejadmoradi, Adarsh Yoga, and Santosh Nagarakatte. A parallelism profiler with what-if analyses for openmp programs. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 198–211. IEEE, 2018.

[8] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M Frans Kaashoek, Robert Tappan Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yue-hua Dai, et al. Corey: An operating system for many cores. In *OSDI*, volume 8, pages 43–57, 2008.

[9] Silas Boyd-Wickizer, Austin T Clements, Yandong Mao, Aleksey Pesterev, M Frans Kaashoek, Robert Tappan Morris, Nickolai Zeldovich, et al. An analysis of linux scalability to many cores. In *OSDI*, volume 10, pages 86–93, 2010.

[10] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[11] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. Spandb: A fast, cost-effective lsm-tree based kv store on hybrid storage. In *FAST*, volume 21, pages 17–32, 2021.

[12] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. Splinterdb: Closing the bandwidth gap for nvme key-value stores. In *2020 USENIX Annual Technical Conference*, 2020.

[13] Source code of COZ. https://github.com/plasma-umass/coz.

[14] cpudist in bcc. https://github.com/iovisor/bcc/blob/master/tools/cpudist.py.

[15] Charlie Curtsinger and Emery D Berger. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 184–197, 2015.

[16] CXL memory expander. https://semiconductor.samsung.com/us/news-events/tech-blog/expanding-the-limits-of-memory-bandwidth-and-density-samsungs-cxl-dram-memory-expander/.

[17] Arnaldo Carvalho De Melo. The new linux 'perf' tools. In *Slides from Linux Kongress*, volume 18, 2010.

[18] Chen Ding, Ting Yao, Hong Jiang, Qiu Cui, Liu Tang, Yiwen Zhang, Jiguang Wan, and Zhihu Tan. Trianglekv: Reducing write stalls and write amplification in lsm-tree based kv stores with triangle container in nvm. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):4339–4352, 2022.

[19] DTrace. http://dtrace.org/blogs/.

[20] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, pages 120–126. ACM, 1982.

[21] Brendan Gregg. The flame graph. *Communications of the ACM*, 59(6):48–57, 2016.

[22] Hackbench. https://github.com/linux-test-project/ltp/blob/master/testcases/kernel/sched/cfs-scheduler/hackbench.c.

[23] Jun He, Kan Wu, Sudarsun Kannan, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Read as needed: Building wiser, a flash-optimized search engine. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 59–73, 2020.

[24] Kewen He, Yujie An, Yijing Luo, Xiaoguang Liu, and Gang Wang. Flatlsm: Write-optimized lsm-tree for pm-based kv stores. *ACM Transactions on Storage*, 2023.

[25] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, jan 2019.

[26] JCOZ. https://github.com/Decave/JCoz.

[27] Jprofiler. https://www.ej-technologies.com/products/jprofiler/overview.html.

[28] Dongui Kim, Chanyeol Park, Sang-Won Lee, and Beomseok Nam. Bolt: Barrier-optimized lsm-tree. In *Proceedings of the 21st International Middleware Conference*, pages 119–133, 2020.

[29] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young-ri Choi, Alan Sussman, and Beomseok Nam. Listdb: Union of write-ahead logs and persistent skiplists for incremental checkpointing on persistent memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 161–177, 2022.

[30] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the performance of fast nvm storage with udepot. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 1–15, 2019.

[31] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W Lee, and Jinkyu Jeong. Asynchronous i/o stack: A low-latency kernel i/o stack for ultra-low latency ssds. In *USENIX Annual Technical Conference*, pages 603–616, 2019.

[32] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 447–461, 2019.

[33] John Levon and Philippe Elie. Oprofile: A system profiler for linux, 2004.

[34] Junkai Liang and Yunpeng Chai. Cruisedb: An lsm-tree key-value store with both better tail throughput and tail latency. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1032–1043. IEEE, 2021.

[35] Tongping Liu, Guangming Zeng, Abdullah Muzahid, et al. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 298–313. ACM, 2017.

[36] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[37] Alessandro Morari, Roberto Gioiosa, Robert W Wisniewski, Francisco J Cazorla, and Mateo Valero. A quantitative analysis of os noise. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 852–863. IEEE, 2011.

[38] Off-CPU analysis. https://www.brendangregg.com/offcpuanalysis.html.

[39] Off-CPU Flame Graph. https://www.brendangregg.com/FlameGraphs/offcpuflamegraphs.html.

[40] offcputime in bcc. https://github.com/iovisor/bcc/blob/master/tools/offcputime.py.

[41] INTEL. Breakthrough performance for demanding storage workloads. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-905p-product-brief.pdf.

[42] Fengfeng Pan, Yinliang Yue, and Jin Xiong. dcompaction: Delayed compaction for the lsm-tree. *International Journal of Parallel Programming*, 45:1310–1325, 2017.

[43] Jongwon Park and Jinkyu Jeong. Speculative multi-level access in lsm tree-based kv store. *IEEE Computer Architecture Letters*, 21(2):145–148, 2022.

[44] Specification of pm1735 nvme ssd. https://semiconductor.samsung.com/us/ssd/enterprise-ssd/pm1733-pm1735/.

[45] Behnam Pourghassemi, Ardalan Amiri Sani, and Aparna Chandramowlishwaran. What-if analysis of page load time in web browsers using causal profiling. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(2):27, 2019.

[46] Behnam Pourghassemi, Ardalan Amiri Sani, and Aparna Chandramowlishwaran. Only relative speed matters: Virtual causal profiling. *ACM SIGMETRICS Performance Evaluation Review*, 48(3):113–119, 2021.

[47] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-aware thread management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, 2018.

[48] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 497–514, 2017.

[49] Connectx nics. https://www.nvidia.com/en-us/networking/ethernet-adapters/.

[50] RocksDB. https://github.com/facebook/rocksdb.

[51] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance analysis of nvme ssds and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, pages 1–11, 2015.

[52] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. Matrixkv: reducing write stalls and write amplification in lsm-tree based kv stores with a matrix container in nvm. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, pages 17–31, 2020.

[53] Adarsh Yoga and Santosh Nagarakatte. A fast causal profiler for task parallel programs. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 15–26, 2017.

[54] Adarsh Yoga and Santosh Nagarakatte. Parallelism-centric what-if and differential analyses. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 485–501, 2019.

[55] Yourkit (Java and .NET profiler). https://www.yourkit.com/.

[56] Jinghuan Yu, Sam H Noh, Young-ri Choi, and Chun Jason Xue. Adoc: Automatically harmonizing dataflow between components in log-structured key-value stores for improved performance. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 65–80, 2023.

[57] Tingting Yu and Michael Pradel. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 389–400, 2016.

[58] Fang Zhou, Yifan Gan, Sixiang Ma, and Yang Wang. wperf: Generic off-cpu analysis to identify bottleneck waiting events. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 527–543, 2018.

[59] SAMSUNG. Ultra-low latency with SAMSUNG Z-NAND SSD. https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf.

A Artifact Appendix

A.1 Abstract

Blocked samples is a profiling technique based on sampling, that encompasses both on- and off-CPU events simultaneously. Based on blocked samples, we present two profilers: *bperf*, an easy to-use sampling-based profiler and *BCOZ*, a causal profiler that profiles both on- and off-CPU events simultaneously and estimates potential speedup of optimizations.

A.2 Scope

Our artifact can be used to identify bottlenecks across various applications and pinpoint code lines in need of optimization. Particularly, our approach is an efficient profiling technique for applications where both on- and off-CPU events are mixed.

A.3 Contents

Our artifact consists of three subdirectories: `blocked_samples` (source code of Linux kernel with bperf), `bcoz` (source code of BCOZ), and `osdi24_ae` (OSDI'24 artifacts evaluation). Descriptions of each subdirectory are as follows.

blocked_samples. This directory includes an extended Linux perf subsystem for blocked samples. Blocked samples is a profiling technique based on sampling, that encompasses both on- and off-CPU events simultaneously. Furthermore, the original Linux perf tool is replaced with our bperf (`blocked_samples/tools/perf`).

bcoz. This directory includes source code of BCOZ. BCOZ is a causal profiler that leverages the concept of virtual speedup for both on-CPU and off-CPU events using blocked samples. At its core, BCOZ profiles on-/off-CPU events (i.e., blocked samples) collected by our extended Linux perf subsystem and estimates performance improvement through virtual speedup. BCOZ is extended from COZ [15], a causal profiler for only on-CPU events.

osdi24_ae. This directory is for the OSDI '24 artifacts evaluation. It includes instructions for reproducing the experimental results in the paper.

The instructions in the *Getting Started with Blocked Samples* section of `README.md` in the root directory help verify whether blocked samples functions correctly.

A.4 Hosting

The GitHub repository for the artifacts is available on https://github.com/s3yonsei/blocked_samples.

A.5 Requirements

The Linux kernel version for blocked samples is 5.3.7 and we have verified that blocked samples operates correctly on the Ubuntu 20.04 LTS server. We will soon release the support for blocked samples on the latest Linux kernel version in the repository.