wPerf: Generic Off-CPU Analysis to Identify Bottleneck Waiting Events (technical report)

Fang Zhou, Yifan Gan, Sixiang Ma, Yang Wang
The Ohio State University

Abstract

This paper tries to identify waiting events that limit the maximal throughput of a multi-threaded application. To achieve this goal, we not only need to understand an event's impact on threads waiting for this event (i.e., local impact), but also need to understand whether its impact can reach other threads that are involved in request processing (i.e., global impact).

To address these challenges, wPerf computes the local impact of a waiting event with a technique called *cascaded re-distribution*; more importantly, wPerf builds a *wait-for graph* to compute whether such impact can indirectly reach other threads. By combining these two techniques, wPerf essentially tries to identify events with large impacts on all threads.

We apply wPerf to a number of open-source multi-threaded applications. By following the guide of wPerf, we are able to improve their throughput by up to 4.83×. The overhead of recording waiting events at runtime is about 5.1% on average.

1 Introduction

This paper proposes wPerf, a generic off-CPU analysis method to identify critical waiting events that limit the maximal throughput of multi-threaded applications.

Developers often need to identify the bottlenecks of their applications to improve their throughput. For a single-threaded application, one can identify its bottleneck by looking for the piece of code that takes the most time to execute, with the help of tools like perf [61] and DTrace [20]. For a multi-threaded application, this task becomes much more challenging because a thread could spend time waiting for certain events (e.g., lock, I/O, condition variable, etc.) as well as executing code: both execution and waiting can create bottlenecks.

Accordingly, performance analysis tools targeting multi-threaded applications can be categorized into two types: on-CPU analysis to identify bottlenecks created by execution and off-CPU analysis to identify bottlenecks created by waiting [57]. As shown in previous works, off-CPU analysis is important because optimizing waiting can lead to a significant improvement in performance [3, 4, 9, 14, 42, 70–72, 77].

While there are systematic solutions for on-CPU analysis (e.g., Critical Path Analysis [40] and COZ [16]), existing off-CPU analysis methods are either inaccurate or incomplete. For example, a number of tools can rank waiting events based on their lengths [36, 58, 75], but longer waiting events are not necessarily more important (see Section 2); some other tools design metrics to rank lock contention [2, 18, 76], which is certainly one of the most important types of waiting events, but other waiting events, such as waiting for condition variables or I/Os, can create a bottleneck as well (also see Section 2). As far as we know, no tools can perform accurate analysis for all kinds of waiting events.

To identify waiting events critical to throughput, the key challenge is a gap between the *local impact* and the *global impact* of waiting events: given the information of a waiting event, such as its length and frequency, it may not be hard to predict its impact on the threads waiting for the event (i.e., local impact). To improve overall application throughput, however, we need to improve the throughput of all threads involved in request processing (called *worker threads* in this paper). Therefore, to understand whether optimizing a waiting event can improve overall throughput, we need to know whether its impact can reach all worker threads (i.e., global impact). These two kinds of impacts are not always correlated: events with a small local impact usually have a small global impact, but events with a large local impact may not have a large global impact. As a result, it's hard to directly rank the global impact of waiting events.

To address this problem, we propose a novel technique called "wait-for graph" to compute which threads a waiting event may influence. This technique is based on a simple observation: if thread B never waits for thread A, either directly or indirectly, then optimizing A's waiting events would not improve B, because neither B's execution speed nor B's waiting time would be affected. Following this observation, wPerf models the application as a wait-for graph, in which each thread is a vertex and a directed edge from A to B means thread A sometimes waits for B. We can prove that if such a graph contains any *knots* with worker threads inside them, we must optimize at least one waiting event in each of these knots. Intuitively, this conclusion is a generalization of our observation: a knot is an inescapable section of the graph (see formal definition in Section 3.1), which means the worker threads in a knot never wait for outside threads, so optimizing outside events would not improve these worker threads. However, to improve overall throughput, we must improve all worker threads, which means we must optimize at least one event in the knot. In other words, each knot must contain a bottleneck.

A knot means there must exist cyclic wait-for relationship among its threads. In practice, such cyclic wait-for relationship can be caused by various reasons, such as blocking I/Os, load imbalance, and lock contention.

For complicated knots, wPerf refines them by trimming edges whose local impact is small, because events with little local impact usually have little global impact and thus optimizing them would have little impact on the application. For this purpose, the length of a waiting event can serve as a natural heuristic for its local impact, but using it directly may not be accurate when waiting events are nested. For example, if thread A wakes up B and then B wakes up C later, adding all C's waiting period to edge $C \rightarrow B$ is misleading because part of this period is caused by B waiting for A. To solve this problem, we introduce a technique called "cascaded redistribution" to quantify the local impact of waiting events: if thread A waits for thread B from t_1 to t_2, wPerf checks what B is doing during t_1 to t_2 and if B is waiting for another thread, wPerf will re-distribute the corresponding weight and perform the check recursively.

Given such local impact as a weight on each edge, wPerf can refine a complicated knot by continuously removing its edges with small weights, till the knot becomes disconnected, which allows wPerf to further identify smaller knots. wPerf repeats these two procedures (i.e., identify knots and refine knots) iteratively until the graph is simple enough, which should contain events whose local impact is large and whose impact can potentially reach all worker threads.

We apply wPerf to various open-source applications. Guided by the reports of wPerf, we are able to improve their throughput by up to 4.83×. For example, we find in ZooKeeper [34], using blocking I/Os and limiting the number of outstanding requests combined cause inefficiency when the workload is read-heavy: in this case, for each logging operation, ZooKeeper can only batch a small number of writes, leading to inefficient disk performance. wPerf's runtime overhead of recording waiting events is about 5.1% on average.

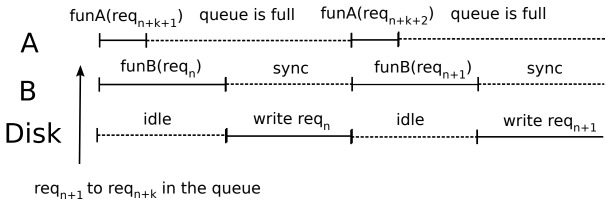
Thread A

```
while (true)
   recv req from network
   funA(req) //2ms
   queue.enqueue(req)
```

Thread B

```
while (true)
   req = queue.dequeue()
   funB(req) //5ms
   log req to a file
   sync //5ms
```

(a) Code (queue is a producer-consumer queue with max size k)



(b) Runtime execution.

Figure 1: An example of a multi-threaded program with a bottleneck waiting event.

2 Motivating Example

This section presents an example that motivates our work. As shown in Figure 1: since thread B needs to sync data to the disk (Figure 1a), B and the disk cannot process requests in parallel at runtime (Figure 1b). As a result, B and the disk combined take 10ms to process a request, which becomes the bottleneck of this application. As one can see, this application is saturated while none of its threads or disks are fully saturated. Furthermore, one can observe the following phenomena:

- Off-CPU analysis is important. In this example, on-CPU analysis like Critical Path Analysis [40] or COZ [16] can identify that *funB* and disk write are worth optimizing, which is certainly correct, but we should not neglect that the blocking pattern between B and the disk is worth optimizing as well: if we can change thread B to write to disk asynchronously, we could double the throughput of this application.
- While lock contention is well studied, we should not neglect other waiting events. The bottleneck of this example is not caused by contentions, but by waiting for I/Os. Replacing the disk with thread C and letting B wait for C on a condition variable can create a similar bottleneck.
- Longer waiting events are not necessarily more important. In other words, events with a large local impact may not have a large global impact. In this example, thread A spends 80% of its time waiting for B, which

is longer than the time B spends waiting for the disk, but it is because A has less work than B and is not the cause of the bottleneck.

Although a number of tools like off-CPU flame graph [58] have been developed to help off-CPU analysis, we are not aware of any tools that can answer the question which waiting events are important, when considering all kinds of waiting events. As a result, such investigation largely relies on the efforts of the developers. For the above simple example, it may not be difficult. In real applications, however, such patterns can become complicated, involving many more threads and devices (see Section 5). These phenomena motivate us to develop a new off-CPU analysis approach, which should be generic enough to handle all kinds of waiting events.

3 Identify Bottleneck Waiting Events

In this paper, we propose wPerf, a generic approach to identify bottleneck waiting events in multi-threaded applications. To be more specific, we assume the target application is processing requests from either remote clients or user inputs, and the goal of wPerf is to identify waiting events whose optimization can improve the application's throughput to process incoming requests.

wPerf models the target application as a number of threads (an I/O device is modeled as a pseudo thread). A thread is either executing some task or is blocked, waiting for some event from another thread. A task can be either a portion of an incoming request or an internal task generated by the application. A thread can be optimized by 1) increasing its speed to execute tasks; 2) reducing the number of tasks it needs to execute; or 3) reducing its waiting time. Since wPerf targets off-CPU analysis, it tries to identify opportunities for the third type.

To identify bottleneck waiting events, wPerf uses two steps iteratively to narrow down the search space: in the first step, it builds the wait-for graph to identify subgraphs that must contain bottlenecks. If these subgraphs are large, wPerf refines them by removing edges with little local impact.

In this section, we first present a few definitions, then explain the basic idea of wPerf in a simplified model, and finally extend the model to general applications.

3.1 Definitions

Definition 3.1. *Worker and background threads. A thread is a worker thread if its throughput of processing its tasks grows with the application's throughput to process its incoming requests; a thread is a background thread if its throughput does not grow with the throughput of the application.*

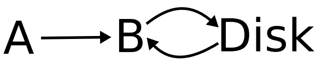


Figure 2: Wait-for graph of the application in Figure 1.

For example, threads that process incoming requests are obvious worker threads; threads that perform tasks like garbage collection or disk flushing are also worker threads, though they usually run in the background; threads that perform tasks like sending heartbeats are background threads.

This definition identifies threads that must be optimized to improve overall application throughput (i.e., worker threads), because they are directly or indirectly involved in processing incoming requests. In real applications, we find most of the threads are worker threads.

Definition 3.2. *Wait-for relationship. Thread A directly waits for thread B if A sometimes is woken up by thread B. Thread A indirectly waits for B if there exists a sequence of threads T_1, T_2, ... T_n such that $T_1 = A$, $T_n = B$, and T_i directly waits for $T_{(i+1)}$. Thread A waits for thread B if A either directly or indirectly waits for B.*

Definition 3.3. *Wait-for graph. We construct a wait-for graph for a multi-threaded application in the following way: each vertex is a thread and a directed edge from thread A to B means A directly waits for B.*

For example, Figure 2 shows the wait-for graph for the application shown in Figure 1. One can easily prove that A waits for B if there is a directed path from A to B.

Definition 3.4. *Knot and sink. In a graph, a knot is a nonempty set K of vertices such that the reachable set of each vertex in K is exactly set K; a sink is a vertex with no edges directed from it [32].*

Intuitively, knot and sink identify minimal inescapable sections of a graph. Note that by definition, a vertex with a self-loop but no other outgoing edges is a knot.

3.2 Identify bottleneck waiting events in a simplified model

In this simplified model, we make the following assumptions and we discuss how to relax these assumptions in the next section: 1) each application is running a fixed number of threads; 2) there are more CPU cores than the number of threads; 3) all threads are worker threads; 4) threads are not performing any I/O operations. Our algorithm uses two steps to narrow down the search space.

3.2.1 Step 1: Identifying knots

Our algorithm first narrows down the search space by identifying subgraphs that must contain bottlenecks, based on the following lemma and theorem.

Lemma 3.1. *If thread B never waits for A, reducing A's waiting time would not increase the throughput of B.*

Proof. If we don't optimize the execution of B, the only way to improve B's throughput is to give it more tasks, i.e., reduce its waiting time. However, since B never waits for A, optimizing A would not affect B's waiting time. Therefore, B's throughput is not affected. □

Theorem 3.2. *If the wait-for graph contains any knots, to improve the application's throughput, we must optimize at least one waiting event in each knot.*

Proof. We prove by contradiction: suppose we can improve the application's throughput without optimizing any events in a knot. On one hand, since all threads are worker threads, if overall throughput were improved, the throughput of each thread should increase (Definition 3.1). On the other hand, because a knot is an inescapable section of a graph, threads in the knot never wait for outside threads, so optimizing outside threads or events would not improve the throughput of threads in the knot (Lemma 3.1). These two conclusions contradict and thus the theorem is proved. □

For example, in Figure 2, thread B and the disk form a knot and thus at least one of their waiting events must be optimized to improve the application's throughput.

A graph must contain either knots or sinks or both [32]. A sink means the execution of the corresponding thread is the bottleneck, which is beyond the scope of off-CPU analysis. A knot means there must exist cyclic wait-for relationship among multiple threads, which can cause the application to saturate while none of the threads on the cycle are saturated. In practice, such cyclic wait-for relationship can happen for different reasons, among which the following ones are common:

- Lock contention. Multiple threads contending on a lock is probably the most common reason to cause a cyclic wait-for relationship. In this case, threads contending on the lock may wait for each other.
- Blocking operation. Figure 1 shows an example of this problem: since B needs to wait for the responses from the disk, and the disk needs to wait for new requests from B, there exists a cyclic wait-for relationship between B and the disk.
- Load imbalance. Many applications work in phases and parallelize the job in each phase [19, 69]. Imbalance across phases or imbalance across threads in the same phase can create a cycle. For example, suppose in phase 1, thread A executes three tasks and thread B executes one task; in phase 2, A executes one task and B executes three tasks: in this case, A needs to wait for B at the end of phase 1 and B needs to wait for A at the end of phase 2, creating a cycle.

3.2.2 Step 2: Refining knots

If a knot is small, the developers may manually investigate it and decide how to optimize. For a large knot, wPerf further narrows down the search space by removing edges whose optimization would have little impact on the application. However, accurately predicting the global impact of a waiting event is a challenging problem in the first place. To address this challenge, we observe that the local impact of a waiting event can be viewed as the *upper bound* of the global impact of this event: improvement to all threads naturally includes improvement to threads waiting for this event, so the local impact of an event should be at least as large as its global impact.

Following this observation, wPerf removes edges with a small local impact until the knot becomes disconnected. When disconnection happens, wPerf tries to identify smaller knots. wPerf repeats these two procedures—identifying knots and trimming edges with a small local impact—until the result is simple enough for developers. We discuss the termination condition in Section 4.3. By combining these two procedures, wPerf essentially tries to identify the edges with a large impact on all worker threads.

Since local impact marks the upper bound of global impact, knot refinement will not bring false negatives (i.e., removing important edges), which means the user will not miss important optimization opportunities. However, it may bring false positives (i.e., not removing unimportant edges), which requires additional effort from the user, but in our case studies, we find such additional effort is not significant, mainly because many edges with a large local impact are outside of the knot and thus are removed.

The total waiting time spent on an edge is a natural heuristic to quantify the local impact of the edge, but we find it may be misleading when waiting events are nested. To illustrate the problem, we show an example in Figure 3: thread C wakes up B at time t_1 and B wakes up A at time t_2. In practice, such nested waiting can happen in two ways: first, it is possible that C wakes up B and A simultaneously and B happens to execute first (e.g., C releases a lock that both A and B try to grab) and we call this type "symmetric waiting"; second, it is also possible that A's Task 3 depends on B's Task 2, which depends on C's Task 1. We call this type "asymmetric waiting". However, from the recorded waiting events, wPerf does not know which type it is, which means its solution to compute the edge weights should work for both types.

To motivate wPerf's solution, we show several options we have tried. The naive solution (Graph1) adds weight $(t_2 - t_0)$ to edge $A \rightarrow B$ and weight $(t_1 - t_0)$ to edge $B \rightarrow C$. This solution underestimates the importance of $B \rightarrow C$, because reducing the time spent on

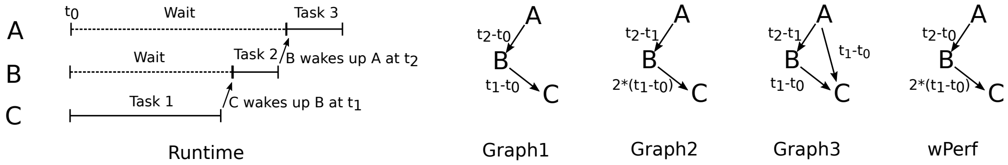


Figure 3: Building edges weights from length of waiting events (Graphs 1-3 are our failed attempts).

$B \rightarrow C$ can automatically reduce the time spent on $A \rightarrow B$. Graph2 moves the overlapping part $(t_1 - t_0)$ from $A \rightarrow B$ to $B \rightarrow C$, which increases the importance of $B \rightarrow C$, but it underestimates the importance of $A \rightarrow B$: in asymmetric waiting, it is possible to optimize $A \rightarrow B$ but not optimize $B \rightarrow C$, so it is inappropriate to assume optimizing $A \rightarrow B$ can only reduce the waiting time by $t_2 - t_1$. Graph3 draws a new edge $A \rightarrow C$ and moves the weight of $(t_1 - t_0)$ to the new edge, indicating that $(t_1 - t_0)$ is actually caused by waiting for C: this approach makes sense for symmetric waiting, but is confusing for asymmetric waiting, in which A does not directly wait for C. wPerf's solution is to keep weight $(t_2 - t_0)$ for edge $A \rightarrow B$, which means optimizing this edge can reduce A's waiting time by up to $(t_2 - t_0)$, and increases the weight of $B \rightarrow C$ by $(t_1 - t_0)$, which means optimizing this edge can lead to improvement in both B and A. wPerf's solution may seem to be unfair for symmetric waiting, but for symmetric waiting, A and B should have similar chance to be woken up first, so if we test the application for sufficiently long, the weights of $A \rightarrow B$ and $B \rightarrow C$ should be close.

Following this idea, wPerf introduces a cascaded redistribution algorithm to build the weights in the general case: at first, wPerf assigns a weight to an edge according to the waiting time spent on that edge. If wPerf finds while thread A is waiting for thread B, thread B also waits for thread C (length t), wPerf increases the weight of $(B \rightarrow C)$ by t. If C waits for other threads during the same period of time, wPerf will perform such adjustment recursively (see the detailed algorithm in Section 4.2).

3.3 Extending the model

Next, we extend our model by relaxing its assumptions.

Not enough CPUs. A thread may also wait because all CPU cores are busy (i.e., the thread is in "runnable" state). We can record the runnable time of each thread: if a thread in a knot is often in the runnable state, then the application may benefit from using more CPU cores or giving those bottleneck threads a higher priority.

I/Os. wPerf models an I/O device as a pseudo thread. If a normal thread sometimes waits for an I/O to complete, wPerf draws an edge from the normal thread to the corresponding I/O thread. If an I/O device is not fully utilized (see Section 4.1), wPerf draws an edge from the I/O thread to all normal threads that have issued I/Os to this device, meaning the device waits for new I/Os from these normal threads.

Busy waiting. Some threads use busy waiting to continuously check whether another thread has generated the events. A typical example is a spin lock. From the OS point of view, a thread that is busy waiting is not counted as waiting, because it is executing code; at the logical level, however, time spent on busy waiting should be counted as waiting time in our model. We discuss how to trace such events in Section 4.1.

Background threads. A knot consisting of only background threads does not have to be optimized to improve the application's throughput, because the throughput of a background thread does not grow with the application's throughput. Note that though not necessary, optimizing such a knot may still be beneficial. For example, suppose a background thread needs to periodically send a heartbeat, during which it needs to grab a lock and thus may block a worker thread. In this case, reducing the locking time of the background thread may improve the worker threads contending on the same lock, but it is not ncessary since optimizing those worker threads may improve the application's throughput as well. Therefore, wPerf reports such a knot to the user, removes the knot, and continues to analyze the remaining graph, because there may exist other optimization opportunities. wPerf uses the following heuristic to identify such a knot: if the knot does not contain any I/O threads and the sum of the CPU utilization of all threads in the knot is less than 100%, wPerf will report it, because this means some threads in the knot sleep frequently, which is a typical behavior of background threads.

Short-term threads. Some applications create a new thread for a new task and terminate the thread when the task finishes. Such short-term threads do not follow our definition of worker thread, because their throughput does not grow with the application's throughput. To apply our idea, wPerf merges such short-term threads into a virtual long-running thread: if any of the short-term threads is running/runnable, wPerf marks the virtual thread as running/runnable; otherwise, wPerf marks the virtual thread as blocked, indicating it is waiting for new tasks from the thread that is creating these short-term threads.

4 Design and Implementation

To apply the above ideas, wPerf incorporates three components: the recorder records the target application's and the OS's waiting events at runtime; the controller receives commands from the user and sends the commands to the recorder; the analyzer builds the wait-for graph from the recorded events offline and tries to identify knots or sinks. In this section, we present how recorder and analyzer work in detail.

4.1 Recording sufficient information

The responsibility of the recorder is to capture sufficient information to allow the analyzer to build the wait-for graph. Towards this goal, such information should be able to answer two questions: 1) if a thread is waiting, which thread is it waiting for? and 2) how long does a thread spend on waiting for another thread? The former will allow us to create edges in the wait-for graph, and the latter will allow us to compute weights for edges.

Profiling tools (e.g., perf [61], DTrace [20], ETW [1], etc.) can record events at different layers. We decide to record waiting events at low layers (i.e. CPU scheduling and interrupt handling) because events at lower layers usually can provide more accurate answers to the above two questions. Taking I/O waiting as an example, one option is to record the lengths of related system calls, but such information is not precise: it is possible that most of the time is indeed spent on waiting for I/Os to complete; it is possible that much time is spent on in-kernel processing, such as data copy; it is also possible that in the kernel, this system call contends with another thread (e.g., write to the same file). Recording at lower layers, on the other hand, can provide precise information.

Following this observation, wPerf uses *kprobe* [41] to record key waiting events in the kernel, with one exception about busy waiting. Since we implement wPerf on Linux, next we first present the background about how Linux performs scheduling and interrupt handling and then present what information wPerf records.

Background. A thread can be in different states: a thread is *running* if it is being executed on a CPU; a thread is *runnable* if it is ready to run but has not been scheduled yet, maybe because all CPUs are busy; a thread is *blocked* if it is waiting for some events and thus cannot be scheduled. While an application can block or unblock a thread through corresponding system calls, OS scheduling module decides which threads to run.

When an interrupt is triggered, CPU jumps to the predefined interrupt request (IRQ) function, preempting the current thread running on the CPU. An IRQ function is usually not executed in a thread context, so it is not controlled by scheduling, which means wPerf has to record IRQ events as well as scheduling events. An IRQ function can wake up a blocked thread: this is common when the thread is waiting for I/Os to complete.

Recording scheduling events. For CPU scheduling, wPerf records two key functions: *__switch_to* and *try_to_wake_up*. *try_to_wake_up* changes a thread's state from *blocked* to *runnable*, which can be invoked in functions like *pthread_mutex_unlock* or when an I/O completes (usually in an IRQ). For this function, wPerf records the timestamp, the thread ID of the thread to be woken up, and the entity (either a thread or an IRQ) that invokes the wakeup. *__switch_to* switches out a thread from a CPU and switches in another. The thread that is switched in must be in *running* state; the one that gets switched out could be either in *runnable* state, which means this switch is caused by CPU scheduling, or in *blocked* state, which means this switch is caused by events like *pthread_mutex_lock* or issuing an I/O. wPerf records the timestamp and the states of both threads.

Recording IRQ events. wPerf intercepts IRQ functions to record its starting time, ending time, its type, and which CPU it runs. To know IRQ type, wPerf intercepts soft IRQ functions defined in *interrupt.h*, each for a specific type of device. By utilizing the function name, wPerf can know what type of hardware device triggers the interrupt, but this approach has a limitation that it cannot distinguish different instances of the same type of devices. This problem could be solved if wPerf can record the IRQ number, which is unique to each device, but unfortunately in Linux, IRQ number is not observable to every IRQ function. Modifying Linux kernel could solve this problem, but our current implementation tries to avoid kernel modification for portability.

Recording information for I/O devices. wPerf models an I/O device as a pseudo I/O thread (Section 3.3). To build the wait-for graph, wPerf needs to know 1) how long a normal thread waits for an I/O thread and 2) how long an I/O thread waits for a normal thread. The recorded IRQ events can only answer the first question.

Since we cannot instrument the internal execution of a hardware device, we have designed an approximate solution to answer the second question: we assume an I/O device is waiting during its idle time; we draw an edge from the device to each normal thread that has issued an I/O to this device; and we distribute the device's idle time to different edges based on how much data each thread sends to the device, meaning the device is waiting for new I/Os from these threads in its idle time. To implement this mechanism, we need to estimate the idle time of each device.

For a disk, we record its used bandwidth and I/Os per second (IOPS). We use the bandwidth to estimate the disk's idle time under sequential I/Os and use the IOPS to estimate its idle time under random I/Os. The case about network interface card (NIC) is more complicated because its capacity is not only limited by the NIC device, but also by the network infrastructure or the remote service. Our current implementation uses the NIC's maximal bandwidth as an upper bound to estimate the NIC's idle time. If the user has a better knowledge about the link bandwidth or the capacity of the remote service, wPerf can use these values for a better estimation.

Recording information for busy waiting. From the OS point of view, a thread that is performing busy waiting is in *running* state but logically it is in *blocked* state. Since such waiting and waking up do not involve kernel functions, recording events in kernel cannot capture them. To make things worse, there is no well-defined interface for such mechanism: some applications use spinlock provided by *pthread* while others may implement their own mechanisms (e.g., MySQL [51]). Previous studies have shown that, although such mechanisms are error prone, they are quite popular [74].

wPerf has no perfect solution to this problem. Instead, it relies on the developers' knowledge. wPerf provides two tracing functions *before_spin* and *after_spin* to developers, so that they can insert these tracing functions at appropriate places. In practice, a developer does not need to trace every of such functions. Instead, he/she can first find frequent ones with on-CPU analysis tools, and then instrument these frequent ones.

Removing false wakeup. A false wakeup is a phenomenon that a thread is woken up but finds its condition to continue is not satisfied, so it has to sleep again. For example, a ticket selling thread A may broadcast to threads B and C, claiming it has one ticket. In this case, only one of B and C can get the ticket and continue. Suppose B gets the ticket: though wPerf can record an event A waking up C, adding weight to edge $C \rightarrow A$ is misleading, because C's condition to continue is not satisfied.

Similar as the case for busy waiting, wPerf provides a tracing function to developers, which can declare a wakeup event as a false one. The developer can insert it after a wakeup, together with a condition check. During analysis, wPerf will remove the pair of wakeup and waiting events that encapsulate this declaration. Once again, the developer only needs to identify significant ones.

Recording call stacks. Developers need to tie events to source code to understand the causes of waiting. For this purpose, wPerf utilizes perf [61] to sample the call stacks of the scheduling and IRQ events as mentioned above. By comparing the timestamp of a call stack with the timestamps of recorded events, wPerf can affiliate a call stack to an edge in the wait-for graph to help developers understand why each edge occurs. Note that getting accurate call stacks requires additional supports, such as enabling the sched_schedstats feature in kernel and compiling C/C++ applications with the -g option. For Java applications, we need to add the -XX:+PreserveFramePointer option to the JVM and attach additional modules like perf-map-agent [62] or async-profiler [6] (wPerf uses perf-map-agent). We are not aware of supports for Python applications yet.

Minimizing recording overhead. To reduce recording overhead, we apply two classic optimizations: 1) to reduce I/O overhead, the recorder buffers events and flushes the buffers to trace files in the background; 2) to avoid contentions, the recorder creates a buffer and a trace file for each core. Besides, we meet two challenges.

First, recording busy waiting and false wakeup events can incur a high overhead in a naive implementation. The reason is that these events are recorded in the user space, which means a naive implementation needs to make system calls to read the timestamp and the thread ID of an event: frequent system calls are known to have a high overhead [66]. To avoid reading timestamps from the kernel space, we use the virtual dynamic shared object (vDSO) technique provided by Linux to read current time in the user space; to avoid reading thread ID from the kernel space, we observe the *pthread* library provides a unique *pthread* ID (PID) for each thread, which can be retrieved in the user space. However, recording only PIDs is problematic, because PID is different from the thread ID (TID) used in the kernel space. To create a match between such two types of IDs, the recorder records both PID and TID for the first user-space event from each thread and records only PIDs afterwards.

Second, Linux provides different types of clocks, but the types supported by vDSO and perf have no overlap, so we cannot use a single type of clock for all events. To address this problem, the recorder records two clock values for each kernel event, one from the vDSO clock and one from the perf clock. This approach allows us to tie perf call stacks to kernel events and to order user-space events and kernel events. However, this approach cannot create an accurate match between perf call stacks and user-space events, so we decide not to record call stacks for user-space events: this is fine since the user needs to annotate these events anyway, which means he/she already knows the source code tied to such events.

4.2 Building the wait-for graph

Based on the information recorded by the recorder, wPerf's analyzer builds the wait-for graph and computes the weights of edges offline in two steps.

In the first step, the analyzer tries to match *wait* and *wakeup* events. A *wait* event is one that changes a thread's state from "running" or "runnable" to

```
1   input: w is a waiting segment.
2   w.start: starting time of this segment
3   w.end: ending time of this segment
4   w.ID: thread ID of this segment
5   w.wakerID: the thread that wakes up this
        segment

7   function cascade(w)
8      add weight (w.end-w.start) to edge
           w.ID → w.wakerID
9      find all waiting segments in w.wakerID
           that overlap with [w.start w.end)
10     for each of these segments
11        if segment.start < w.start
12           segment.start = w.start
13        if segment.end > w.end
14           segment.end = w.end
15        cascade(segment)
```

Figure 4: Pseudocode of cascaded re-distribution.

"blocked"; a *wakeup* event is one that changes a thread's state from "blocked" to "runnable". For each *wait* event, the analyzer searches for the next *wakeup* event that has the waiting thread's ID as the argument.

Such matching of *wait* and *wakeup* events can naturally break a thread's time into multiple segments, in either "running/runnable" or "waiting" state. The analyzer treats running and runnable segments in the same way in this step and separates them later. At the end of this step, the analyzer removes all segments which contain the false wakeup event, by removing the *wakeup* and *wait* events that encapsulate the event.

In the next step, the analyzer builds the wait-for graph using the cascaded re-distribution algorithm (Figure 3). As shown in Figure 4, the analyzer performs a recursive algorithm for each waiting segment: it first adds the length of this segment to the weight of edge *w.ID* → *w.wakerID* (line 8) and then checks whether thread wakerID is waiting during the same period of time (line 9). If so, the analyzer recursively calls the *cascade* function for those waiting segments (line 15). Note that the waiting segments in wakerID will be analyzed as well, so their lengths are counted multiple times in the weights of the corresponding edges. This is what cascaded re-distribution tries to achieve: nested waiting segments that cause multiple threads to wait should be emphasized, because optimizing such segments can automatically reduce waiting time of multiple threads.

After building the wait-for graph, the analyzer applies the algorithms described in Section 3: the analyzer first applies the Strongly Connected Component (SCC) algorithm to divide the graph into multiple SCCs and finds SCCs with no outgoing edges: an SCC with no outgoing edges is either a knot or a sink. If a knot is still complex, the analyzer repeatedly removes the edge with the lowest weight, until the knot becomes disconnected. Then the analyzer identifies knots or sinks again. The analyzer repeats this procedure till the developer finds the knot understandable. Finally, the analyzer checks whether the remaining threads contain any runnable segments: if so, the application may benefit from using more CPU cores or giving higher priority to these threads.

The analyzer incorporates two optimizations:

Parallel graph building. Building the wait-for graph could be time consuming if the recorded information contains many events. The analyzer parallelizes the computation of both steps mentioned above. In the first step, the analyzer parallelizes the matching of events and separation of segments: this step does not require synchronization because the event list is read-only and the output segment information is local to each analyzer thread. In the second step, the analyzer parallelizes the cascaded re-distribution for each segment: this phase does not require synchronization either because the segmentation information becomes read-only and we can maintain a local wait-for graph for each analyzer thread and merge all local graphs when all threads finish.

Merging similar threads. Many applications create a number of threads to execute similar kinds of tasks. wPerf merges such threads into a single vertex to simplify the graph. To identify similar threads, wPerf's utilizes the recorded call stacks: the analyzer merges two threads if their distributions of call stacks are similar. Note that in the original wait-for graph, a vertex should never have a self-loop because a thread should not wait for itself, but after merging similar threads, a self-loop can happen if similar threads wait for each other.

4.3 Using wPerf

First, the user needs to run the target application and use the wPerf recorder to record events. wPerf provides commands to start and stop recording at any time. If the user observes significant busy waiting or false wakeup during the experiment, he/she should annotate those events and re-run the experiment.

Then the user needs to run the analyzer on the recorded events. The analyzer provides both a graphic output and a text output to present the bottleneck. In this step, the user can set up the termination condition of knot refinement. By default, the refinement terminates when the remaining graph is either a single vertex or a simple cycle. In addition, the user can instruct the refinement to terminate when the smallest weight in the remaining graph is larger than a threshold. The user should set this threshold based on how much improvement he/she targets, since the weight of an edge represents the upper bound of the improvement one may gain by optimizing the edge.

In the third step, the user needs to investigate the knot to identify optimization opportunities. To facilitate such investigation, wPerf allows the user to query the call stacks attached to each edge to understand how each edge

	Problem	Speedup	Known fixes?	Involved techniques
HBase [5]	Blocking write	2.74×	Yes	VI, M-SHORT, M-SIM, FW
ZooKeeper [34, 79]	Blocking write	4.83×	No	VI
HDFS [29, 65]	Blocking write	2.56×	Yes	VI, M-SIM
NFS [56]	Blocking read	3.9×	No	VI, M-SIM
BlockGrace [10, 73]	Load imbalance	1.44×	No	M-SHORT, M-SIM
Memcached [47]	Lock contention	1.64×	Partially	VI, M-SIM
MySQL [51]	Lock contention	1.42×	Yes	VI, M-SIM, BW

Table 1: Summary of case studies. (Speedup = $\frac{ImprovedThroughput}{OriginalThroughput}$; VI: virtual I/O threads; M-SHORT: merging short-term threads; M-SIM: merging similar threads; BW: tracing busy waiting; FW: tracing false wakeup)

is formed. This step requires the user's efforts, and our experience is that for one who is familiar with the target application, this step usually takes no more than a few hours. One reason that simplifies this step is that many edges are caused by a thread waiting for new tasks from another thread (e.g., $Disk \rightarrow B$ in Figure 1), which are usually not optimizable.

Finally, the user needs to optimize the application. Similar as most other profiling tools, wPerf does not provide any help in this step. Based on our experience (Section 5), we have summarized a few common problems and potential solutions, most of which are classic: for blocking I/Os, one could consider using non-blocking I/Os or batching I/Os; for load imbalance, one could consider fine-grained task scheduling; for lock contention, one could consider fine-grained locking. However, since most of such optimizations will affect the correctness of the application, the user needs to investigate whether it is possible and how to apply them. In our case studies, the required user's efforts in this step vary significantly depending on the optimization, ranging from a few minutes to change a configuration option to a few weeks to re-design the application.

Taking the application in Figure 1 as an example, wPerf will output a wait-for graph like Figure 2, in which B and the disk form a knot. The user can then query the call stacks of edges $B \rightarrow Disk$ and $Disk \rightarrow B$; wPerf will show that $B \rightarrow disk$ is caused by the *sync* call in thread B and $Disk \rightarrow B$ is caused by the disk waiting for new I/Os from B. The user will realize that $Disk \rightarrow B$ is not optimizable and thus will focus on the *sync* call.

5 Case Study

To verify the effectiveness of wPerf, we apply wPerf to a number of open-source applications (Section 5.1): we try to optimize the events reported by wPerf and see whether such optimization can lead to improvement in throughput. We find some problems are already fixed in newer versions of the applications or online discussions, which can serve as a direct evidence of wPerf's accuracy. Table 1 summarizes our case studies. Note that we have avoided complicated optimizations because how to optimize is not the contribution of wPerf, and thus there may exist better ways to optimize the reported problems.

Furthermore, as a comparison, we run three existing tools on the same set of applications and present their reports (Section 5.2). Finally, we report the overhead of online recording and offline analysis (Section 5.3).

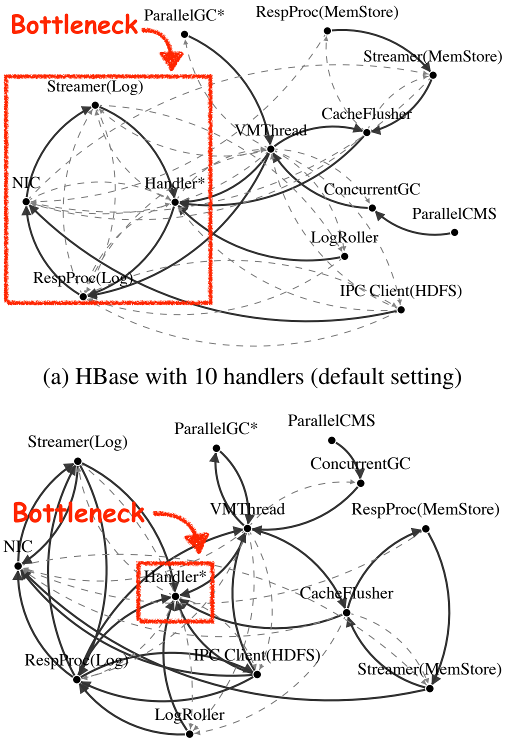
We run all experiments in a cluster with 21 machines: one machine is equipped with two Intel Xeon E5-2630 8-core processors (2.4GHz), 64GB of memory, and a 10Gb NIC; 20 machines are equipped with an Intel Xeon E3-1231 4-core processor (3.4GHz), 16GB of memory, and a 1Gb NIC each.

For each experiment, we record events for 90 seconds. We set the analyzer to terminate when the result graph is a single vertex or a simple cycle or when the lowest weight of its edges is larger than 20% of the recording time (i.e., 18). We visualize the wait-for graph with D3.js [17], and we use solid lines to draw edges whose weights are larger than 18 and use dashed lines to draw the other edges. Since D3.js cannot show a self-loop well, we use "*" to annotate threads with self-loops whose weights are larger than 18. We record all edge weights in Section A. wPerf uses a thread ID to represent each thread, and for readability, we manually check the call stacks of each thread to find its thread name and replace the thread ID with the thread name. We set perf sampling frequency to be 100Hz, which allows perf to collect sufficient samples with a small overhead.

5.1 Effectiveness of wPerf

HBase. HBase [5] is an open-source implementation of Bigtable [12]. It provides a key-value like interface to users and stores data on HDFS. We first test HBase 0.92 with one RegionServer, which runs on HDFS with three DataNodes. We run a write workload with a key size of 16 bytes and a value size of 1024 bytes.

With the default setting, HBase can achieve a throughput of 9,564 requests per second (RPS). Figure 5a shows



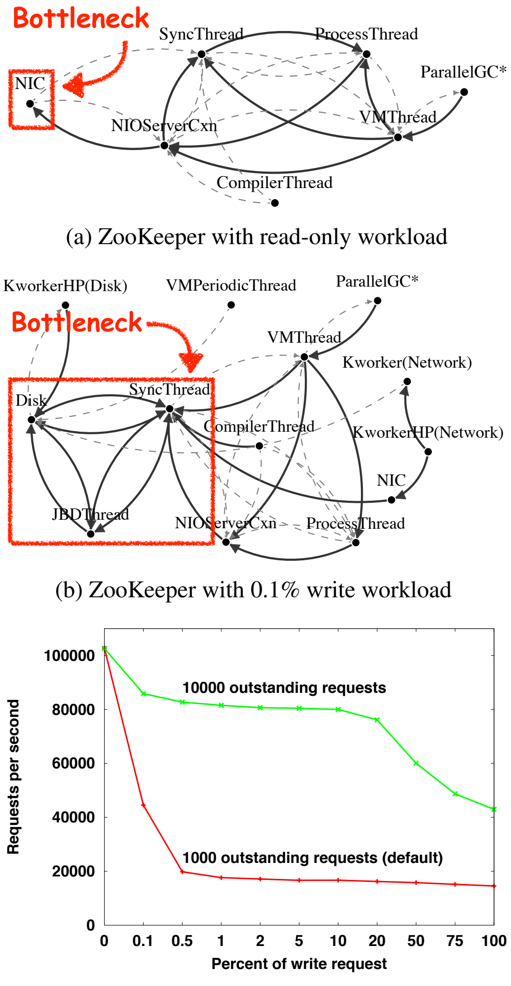
(a) HBase with 10 handlers (default setting)

(b) HBase with 60 handlers

Figure 5: Wait-for graphs of HBase. For readability, we sort edges by their weights and only show the top 40.

the wait-for graph, in which wPerf identifies a significant cycle among HBase Handler threads, HDFS Streamer threads, the NIC, and HDFS ResponseProcessor threads. This cycle is created for the following reason: the Handler threads flushes data to the Streamer threads; the Streamer threads send data to DataNodes through the NIC; when the NIC receives the acknowledgements from the DataNodes, it wakes up the ResponseProcessors; and finally the ResponseProcessors notify the Handlers that a flushing is complete. The blocking flushing pattern, i.e., the Handlers must wait for notification of flushing complete from the ResponseProcessor, is the fundamental reason to create the cycle. The HBase developers are aware that blocking flush is inefficient, so they create multiple Handlers to flush in parallel, but the default number of 10 Handlers is too small on a modern server.

We increase the number of Handlers and HBase can achieve a maximal throughput of 13,568 RPS with 60 Handlers. Figure 5b shows the new wait-for graph, in which wPerf identifies the Handlers as the main bottleneck. Comparing to Figure 5a, the edge weight of Handler → ResponseProcessor drops from 87.4 to 16.5: this is because overlapping more Handlers make them spend more time in runnable state. The setting of Handler count has been discussed online [27, 28].



(a) ZooKeeper with read-only workload

(b) ZooKeeper with 0.1% write workload

(c) Throughput of ZooKeeper.

Figure 6: Wait-for graphs and throughput of ZooKeeper.

In Figure 5b, wPerf identifies a significant self-loop inside Handlers. Such waiting is caused by contentions among Handlers. We find that HBase 1.28 has incorporated optimizations to reduce such contentions and our experiments show that it can improve the throughput to 26,164 RPS. Such results confirm the report of wPerf: fixing the two bottlenecks reported by wPerf can bring a total of 2.74× speedup.

ZooKeeper. ZooKeeper [34, 79] is an open-source implementation of Chubby [11]. We evaluate ZooKeeper 3.4.11 with a mixed read-write workload and 1KB key-value pairs. As shown in Figure 6c, we find a performance problem that even adding 0.1% write can significantly degrade system throughput from 102K RPS to about 44K RPS. We use wPerf to debug this problem.

As shown in Figure 6a, for the read-only workload, wPerf identifies NIC as the major bottleneck, which is reasonable because the NIC's max bandwidth is 1Gbps: this is almost equal to 102K RPS. For the workload with 0.1% write (Figure 6b), however, wPerf identifies the key bottleneck is a knot consisting of the SyncThread in ZooKeeper, the disk, and the journaling thread in the file system. As shown in the knot, the disk spends a lot of time waiting for the other two, which means the disk's bandwidth is highly under-utilized.

We investigate the code of SyncThread. SyncThread needs to log write requests to disk and perform a blocking *sync* operation, which explains why it needs to wait for the disk. Sync for every write request is obviously inefficient, so ZooKeeper performs a classic batching optimization that if there are multiple outstanding requests, it will perform one sync operation for all of them. In ZooKeeper, the number of requests to batch is limited by two parameters: one is a configuration option to limit the total number of outstanding requests in the server (default value 1,000), which is used to prevent out of memory problems; the other is a hard-coded 1,000 limit, which means the SyncThread will not batch more than 1,000 requests. However, we find both limits count both read and write requests, so if the workload is dominated by reads, the SyncThread will only batch a small number of writes for each sync, leading to inefficient disk access.

We try a temporary fix to raise this limit to 10,000, by modifying both the configuration file and the source code. As shown in Figure 6c, such optimization can improve ZooKeeper's throughput by up to 4.83X. However, a fixed limit may not be a good solution in general: if the workload contains big requests, a high limit may cause out of memory problems; if the workload contains small requests, a low limit is bad for throughput. Therefore, it may be better to limit the total size of outstanding requests instead of limiting the total number of them.

HDFS NameNode. HDFS [29, 65] is an open-source implementation of Google File System [23]. It incorporates many DataNodes to store file data and a NameNode to store system metadata. Since NameNode is well-known to be a scalability bottleneck [64], we test it with a synthetic workload [63]: we run MapReduce TeraSort over HDFS 2.7.3, collect and analyze the RPC traces to NameNode, and synthesize traces to a larger scale.

With the default setting, NameNode can reach a maximal throughput of 3,129 RPCs per second. As shown in Figure 7, wPerf identifies the bottleneck is a cycle between Handler threads and the disk. Our investigation shows that its problem is similar to that of ZooKeeper: Handler threads need to log requests to the disk and to improve performance, NameNode batches requests from all Handlers. Therefore, the number of requests to be batched is limited by the number of Handlers. The default setting of 10 Handlers is too small to achieve good disk performance. By increasing the number of Handlers, NameNode can achieve a throughput of about 8,029 RPCs per second with 60 handlers. This problem has been discussed online [53, 54].

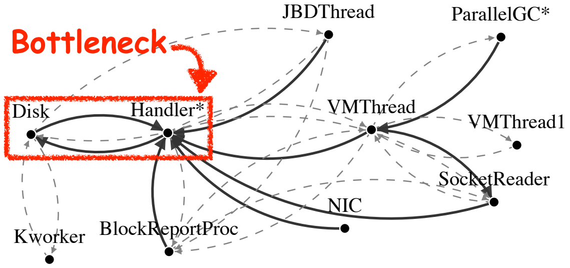


Figure 7: Wait-for graphs of HDFS NameNode.

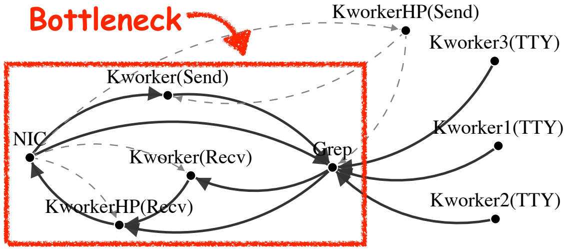


Figure 8: Wait-for graph of running grep over NFS.

NFS. Networked File System (NFS) [56] is a tool to share files among different clients. We set up an NFS server 3.2.29 on CloudLab [15] and set up one NFS client 2.7.5 on our cluster. We test its performance by storing Linux 4.1.6 kernel source code on it and running "grep".

As shown in Figure 8, wPerf identifies a cycle among the grep process, the kernel worker threads, and the NIC. The reason is that grep performs blocking read operations. As a result, grep needs to wait for data from the receiver threads, and the sender threads need to wait for new read requests from grep. This problem can be optimized by either performing reads in parallel or prefetching data asynchronously. We create two NFS instances, distribute files into them, and run eight grep processes in parallel: this can improve the throughput by 3.9×.

BlockGrace. BlockGrace [10, 73] is an in-memory graph processing system. It follows the classic Bulk Synchronous Parallel (BSP) model [69], in which an algorithm is executed in multiple iterations: in each iteration, the algorithm applies the updates from the last iteration and generates updates for the next iteration. We test BlockGrace with its own Single-Source Shortest Path (SSSP) benchmark and with 32 worker threads.

wPerf identifies a cycle between the main thread and the computation threads. Since the wait-for graph is simple, consisting of only these two types of threads, we do not show it here. Our investigation shows the primary reason is the main thread needs to perform initialization work for the computation threads, so the computation threads need to wait for initialization to finish and the main thread then waits for all computation threads to finish. To solve this problem, we let the computation threads perform initialization in parallel: this can improve the throughput by 34.19%.

Then we run wPerf again and find the cycle still exists, but the weight of (computation thread → main thread) is reduced. Our investigation shows the secondary reason is the load imbalance among computation threads. To alleviate this problem, we apply fine-grained task scheduling and implement long running computation threads (original BlockGrace creates new computation threads in each iteration): these techniques can further improve the throughput by 17.82% (44.14% in total).

Memcached. Memcached [47] is a popular in-memory key-value store. We ran the memaslap benchmark [46] to test its performance. We configure memaslap to use a window of 1024 keys per client and set overwrite ratio to be 0% (we tried different overwrite ratio and it does not have a significant impact on result).

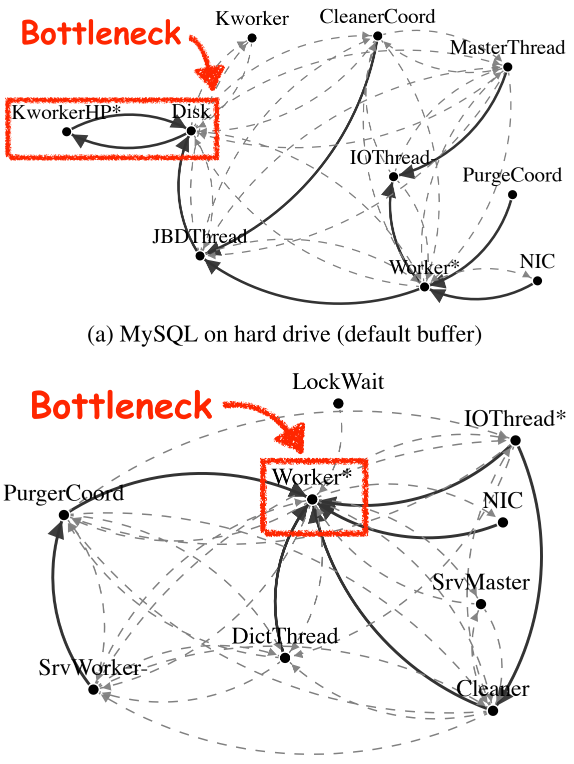
We first run the experiment on Memcached 1.4.36, which is the newest version at that moment. It can achieve a maximal throughput of 354K RPS. wPerf finds a knot with a single vertex, which is merged from multiple worker threads and has a self-loop. Since this wait-for graph is simple, we do not show it here. The recorded stack trace shows that the waiting is mainly caused by lock contention on the LRU list and on slab memory allocator.

To optimize the first lock contention, we change memcached to use spinlock for LRU related operations, because these operations are usually short. To reduce the second contention, we deploy two memcached servers on one machine, each with half number of worker threads. These two optimizations combined can increase the throughput of Memcached by 35%, to 547K RPS.

We find Memcached 1.5.2 has incorporated an optimization to reduce LRU-related contention, by using a separate thread to search and mark deleted entries in the LRU list, so that a worker thread can quickly find an empty entry. This optimization improves Memcached's throughput to about 527K RPS. In this version, we find our first optimization is not effective anymore, because LRU-related contention is already reduced; our second optimization to run two Memcached servers, on the other hand, can improve the throughput to 580K RPS.

MySQL. MySQL [51] is an open-source transactional database system widely used in practice.

MySQL has a complicated internal design: it creates a worker thread for each client, which reads the client's commands, executes them, and sends the replies to the client. Besides, MySQL creates a number of other threads, including a default number of eight I/O threads to perform asynchronous reads when worker threads need to read a page from the storage; four page cleaner threads to asynchronously write dirty pages to the storage; four purge threads to purge log space, a health monitor thread; and a thread to collect statistics. In all experiments, wPerf merges all worker threads into a single vertex and all I/O threads into a single vertex.



(a) MySQL on hard drive (default buffer)

(b) MySQL on RAM-disk (default buffer)

Figure 9: Wait-for graphs of MySQL experiments.

We ran the TPC-C benchmark [68] over MySQL. TPC-C simulates an online transaction processing system: it creates a number of warehouses and a number of clients for each warehouse. The client can browse and purchase items from a warehouse.

We start by running experiments using the default setting of MySQL and storing data on a hard drive. We can gain a maximal throughput of 66.133 transactions/sec. wPerf identifies the bottleneck is the disk subsystem, which is as expected, since hard-drive is well-known to cause I/O bottlenecks.

Next we create a RAM-disk and set MySQL to store all data in the RAM-disk. This time we can gain a maximal throughput of 2681.815 transactions/sec. wPerf identifies a knot consisting of the worker thread (it has a self-loop). The corresponding call stacks recorded by wPerf show that such wait-for relationship is caused by contention for multiple reasons: the first is contention on the locks of page buffers. This problem has been reported in developers forum [52]. We increase MySQL's buffer size to 8GB (default buffer size is 128MB) so that

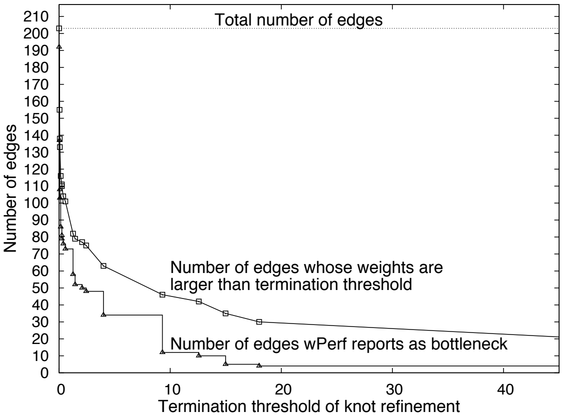


Figure 10: Changing the termination condition for HBase.

all data can be buffered in memory. This time we can gain a throughput of 3806.075 transactions/sec. The second reason is contention on rows. Since previous works [71, 72] have studied how to optimize the concurrency control of MySQL, we decide not to further optimize.

Effects of termination condtion. The user can terminate the knot refinement when the minimal weight of the edges in the knot is larger a threshold. We use threshold 18 in previous experiments and Figure 10 uses HBase as an example to study how this threshold affects wPerf. The gap between the top line (i.e., total number of edges) and the middle line (i.e., number of edges whose weights are larger than the termination threshold) represents the number of edges eliminated because of their small weights. As one can see, only using weights as a heuristic can eliminate many edges, but even with a large threshold, there are still 20-30 edges remaining. The gap between the middle line and the bottom line (i.e., number of edges wPerf reports as bottleneck) represents the number of edges eliminated by knot identification, i.e, these edges have large weights but are outside of the knot. By combining weights (i.e., cascaded re-distribution) and knot identification, wPerf can narrow down the search space to a small number of edges. For other applications, we observe the similar trend in ZooKeeper, HDFS NameNode, NFS, and MySQL experiments; for BlockGrace and Memcached experiments, we do not observe such trend because their wait-for graphs are simple and need little refinement.

Summary. By utilizing wPerf, we are able to identify bottleneck waiting events in a variety of applications and improve their throughput, which confirms the effectiveness of wPerf. Though most of the problems we find are classic ones, they raise some new questions: many problems are caused by inappropriate setting (e.g., number of threads, number of outstanding requests, task granularity, etc.) and no fixed setting can work well for all workloads, so instead of expecting the users to find the best setting, it may be better for the application to change such setting adaptively according to the workload.

	COZ	Flame graph	SyncPerf
HBase	-	Yes	-
ZooKeeper	-	No	-
HDFS	-	No	-
NFS	No	Yes	No
BlockGrace-1	Yes	Yes	No
BlockGrace-2	No	Yes	No
Memcached	Maybe	No	Yes
MySQL	Maybe	No	*

Table 2: Can other tools identify similar problems? (- the tool does not support Java; * experiment reports errors.)

5.2 Comparison to existing tools

As a comparison, we test one on-CPU analysis tool (COZ) and two off-CPU analysis tools (perf and SyncPerf) on the same set of applications. Since COZ and SyncPerf currently do not support Java, we run them only on NFS, BlockGrace, Memcached, and MySQL. We summarize their reports in Table 2 and record all their detailed reports in Appendix B to D.

COZ. To compute how much improvement we can gain by optimizing a certain piece of code, COZ [16] *virtually speeds up* the target piece of code by keeping its speed unchanged and slowing down other code when the target code is running. After the experiment is finished, COZ adjusts the measured throughput to compensate for this slowdown.

COZ is designed for on-CPU analysis, and when we try to use it to analyze off-CPU events, we meet two problems: first, COZ's implementation can only virtually speed up execution on the CPU but cannot virtually speed up I/O devices and thus it does not report any bottlenecks related to I/Os. For example, in the grep over NFS experiment (Figure 11) , COZ suggests us to optimize code in *kwset.c*, which is grep's internal data structure, but does not report anything related to I/Os. However, we believe there is nothing fundamental to prevent COZ from implementing virtual speed up for I/O devices. The second problem, however, is fundamental: the virtual speed up idea does not work well with waiting events, because in many cases, slowing down other events will automatically slow down a waiting event, which breaks COZ's idea to keep the speed of the target event unchanged. Taking the application in Figure 1 as an example, suppose we want to investigate how much improvement we can gain by removing the "sync" call: following COZ's idea, we should keep the length of "sync" unchanged and slow down the disk write, but this will automatically increase the length of "sync". For this reason, we do not find an accurate way to apply COZ to off-CPU events.

That said, we find on-CPU and off-CPU analysis are not completely orthogonal, so COZ can provide hints to off-CPU analysis in certain cases. For example, in the BlockGrace experiment, the first bottleneck (BlockGrace-1) is caused by the computation threads waiting for the main thread to perform initialization: while wPerf identifies this bottleneck as a knot consisting of the main thread and the computation threads, COZ identifies that the initialization code is worth optimizing (e.g., Graph.cpp:97 in Figure 11). Both reports can motivate the user to parallelize the initialization phase. The second bottleneck (BlockGrace-2), however, is caused by load imbalance among worker threads. While wPerf identifies a knot again, which motivates us to improve load balance, COZ reports the code in the computation threads is worth optimizing (e.g., Scheduler.cpp:333 and Graph.cpp:113 in Figure 11), which is certainly correct but misses the opportunity to improve load balance. Lock contention (e.g., in Memcached and MySQL) is another example: COZ can identify that execution in a critical section is worth optimizing (e.g., item.c:463 in Figure 13 and ib0mutex.h:706 in Figure 14). In this case, an experienced programmer may guess that reducing contention with fine-grained locking may also help, but without additional information, such guess may be inaccurate because long execution in the critical section can create a bottleneck as well even if there is almost no contention.

In summary, COZ can identify bottleneck on-CPU events, which wPerf cannot identify, but when regarding off-CPU events, COZ can at most provide some hints while wPerf can provide more accurate reports. Therefore, COZ and wPerf are mainly complementary.

Off-CPU flame graph. perf's off-CPU flame graph [58] can output all calls stacks causing waiting and aggregate them based on their lengths. However, it does not tell which events are important. One can focus on long events: for HBase (Figure 15), grep over NFS (Figure 18), and BlockGrace (Figure 19), the longest events happen to be the same as the ones reported by wPerf; for the others (Figure 16, Figure 17, Figure 20, and Figure 21), the longest ones are not the same as the ones reported by wPerf, and such unimportant but long waiting are usually caused by threads waiting for some rare events, such as JVM's garbage collection threads or threads waiting for new connections.

SyncPerf. SyncPerf [2] reports long or frequent lock contentions. For Memcached (Figure 24 items.c), it reports similar problems as wPerf, but for the other systems, it does not: for grep over NFS (Figure 22), SyncPerf does not report anything because grep does not have contention at all; for BlockGrace (Figure 23), SyncPerf reports asymmetric contention, but the key problem is imbalance among threads. We fail to run SyncPerf with MySQL, and the SyncPerf developers confirmed that it is probably because MySQL uses some functions SyncPerf does not fully implement. Note that the SyncPerf paper reported contentions in MySQL, so our problem may be caused by different versions of MySQL or glibc, etc. and if fixed, we believe SyncPerf and wPerf should identify similar bottlenecks.

	Slowdown	Trace size	Analysis
HBase	2.84%	1.4GB	110.6s
ZooKeeper	3.37%	393.9MB	23.8s
HDFS	3.40%	64.8MB	10.9s
NFS	0.77%	3.6MB	5.1s
BlockGrace	8.04%	110.7MB	14.7s
Memcached	2.43%	2.7GB	160.0s
MySQL	14.64%	7.4GB	271.9s

Table 3: Overhead of wPerf (recording for 90 seconds)

5.3 Overhead

Table 3 reports overhead of wPerf. At runtime, wPerf incurs an average overhead of 5.1% for recording events. For BlockGrace and MySQL, the two applications with relatively large overhead, we further decouple the sources of their overhead: for BlockGrace, recording events in kernel and recording call stacks with perf incur 3.1% and 4.9% overhead respectively; for MySQL, recording events in kernel, recording call stacks with perf, and recording events in user space incur 0.5%, 4.2%, and 9.9% overhead respectively.

As shown in Table 3, the trace size and analysis time vary significantly depending on the number of waiting events in the application; the analysis time further depends on the number of nested waiting events. wPerf's parallel analysis helps significantly: for example, for HBase, with 32 threads, it reduces analysis time from 657.1 seconds to 110.6 seconds.

Besides, wPerf needs users' efforts to insert tracing functions for false wakeup and busy waiting events: we inserted 7 lines of code in HBase to trace false wakeup events and 12 lines of code in MySQL to trace busy waiting events; we do not modify the other applications since these two events are not significant in them.

6 Related Work

Performance analysis is a broad area: some works focus on identifying key factors to affect throughput [20, 59, 61] and others focus on latency-related factors [13, 33]; some works focus on a few abnormal events [7, 43, 45, 78] and others focus on factors that affect average performance [13, 20, 21, 37, 59, 61]. wPerf targets identifying key factors that affect the average throughput of the application. Therefore, this section mainly discusses related work in this sub-area.

As mentioned earlier, tools in this sub-area can be categorized into on-CPU analysis and off-CPU analysis.

On-CPU analysis. For single-threaded applications, traditional performance profilers measure the time spent in different call stacks and identify functions that consume most time. Following this idea, a number of performance profilers (e.g., perf [61], DTrace [20], oprofile [59], yourkit [75], gprof [25, 26], etc.) have been developed and applied in practice. Two approaches are widely used: the first is to periodically sample the call stack of the target application and use the number of samples spent in each function to approximate the time spent in each function; the second is to instrument the target application and trace certain function calls [44, 55].

For multi-threaded programs, a number of works try to identify the critical path of an algorithm [22, 30, 31, 48–50, 60, 67] and pieces of code that often do not execute in parallel [35, 38, 39]. COZ [16] can further estimate how much improvement we can gain by optimizing a certain piece of code, as discussed in Section 5.2.

Off-CPU analysis. To identify important waiting events, many existing tools (e.g., perf [61], yourkit [75], jprofiler [36], etc.) can rank waiting events based on their aggregated lengths. However, as shown in Section 2, long waiting events are not necessarily important.

A number of tools design metrics to identify important lock contentions [2, 8, 18, 24, 76]. For example, Freelunch [18] proposes a metric called "critical section pressure" to identify important locks; SyncProfiler [76] proposes a graph-based solution to rank critical sections; SyncPerf [2] considers both the frequency and length of contentions. However, they are not able to identify problems unrelated to contention.

SyncProf [76] and SyncPerf [2] can further identify the root cause of a problem and make suggestions about how to fix the problem. Similar as many other tools, wPerf does not provide such diagnosis functionality.

7 Conclusion and Future Work

To identify waiting events that limit the application's throughput, wPerf uses cascaded re-distribution to compute the local impact of a waiting event and uses wait-for graph to compute whether such impact can reach other threads. Our case studies show that wPerf can identify problems other tools cannot find. In the future, we plan to extend wPerf to distributed systems, by connecting wait-for graphs from different nodes.

References

[1] Event Tracing for Windows (ETW). https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/event-tracing-for-windows--etw-.

[2] Alam, M. M. u., Liu, T., Zeng, G., and Muzahid, A. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 298–313.

[3] Ananthanarayanan, G., Ghodsi, A., Shenker, S., and Stoica, I. Effective straggler mitigation: Attack of the clones. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 185–198.

[4] Ananthanarayanan, G., Kandula, S., Greenberg, A., Stoica, I., Lu, Y., Saha, B., and Harris, E. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 265–278.

[5] Apache HBASE. http://hbase.apache.org/.

[6] aync-profiler: Sampling CPU and HEAP Profiler for Java Featuring AsyncGetCallTrace and perf events. https://github.com/jvm-profiling-tools/async-profiler/releases.

[7] Attariyan, M., Chow, M., and Flinn, J. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 307–320.

[8] Bach, M. M., Charney, M., Cohn, R., Demikhovsky, E., Devor, T., Hazelwood, K., Jaleel, A., Luk, C.-K., Lyons, G., Patil, H., and Tal, A. Analyzing parallel programs with pin. *Computer 43*, 3 (Mar. 2010), 34–41.

[9] Bhat, S. S., Eqbal, R., Clements, A. T., Kaashoek, M. F., and Zeldovich, N. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 69–86.

[10] Blockgrace. https://github.com/wenleix/BlockGRACE.

[11] Burrows, M. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 335–350.

[12] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2 (June 2008), 4:1–4:26.

[13] Chow, M., Meisner, D., Flinn, J., Peek, D., and Wenisch, T. F. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, 2014), USENIX Association, pp. 217–231.

[14] Clements, A. T., Kaashoek, M. F., Zeldovich, N., Morris, R. T., and Kohler, E. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 1–17.

[15] CloudLab. https://www.cloudlab.us.

[16] Curtsinger, C., and Berger, E. D. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 184–197.

[17] D3.JS Javascript Library. https://d3js.org/.

[18] David, F., Thomas, G., Lawall, J., and Muller, G. Continuously measuring critical section pressure with the free-lunch profiler. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (New York, NY, USA, 2014), OOPSLA '14, ACM, pp. 291–307.

[19] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 10–10.

[20] Dtrace. http://dtrace.org/.

[21] Erlingsson, U., Peinado, M., Peter, S., and Budiu, M. Fay: Extensible distributed tracing from kernels to clusters. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 311–326.

[22] Garcia, S., Jeon, D., Louie, C. M., and Taylor, M. B. Kremlin: Rethinking and rebooting gprof for the multicore age. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2011), PLDI '11, ACM, pp. 458–469.

[23] Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 29–43.

[24] Goldin, M. Thread performance: Resource contention concurrency profiling in visual studio 2010. https://msdn.microsoft.com/en-us/magazine/ff714587.aspx.

[25] GNU gprof. https://sourceware.org/binutils/docs/gprof/.

[26] Graham, S. L., Kessler, P. B., and Mckusick, M. K. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction* (New York, NY, USA, 1982), SIGPLAN '82, ACM, pp. 120–126.

[27] Apache HBase (TM) Configuration. http://hbase.apache.org/0.94/book/important_configurations.html.

[28] HBase Administration Cookbook. https://www.safaribooksonline.com/library/view/hbase-administration-cookbook/9781849517140/ch09s03.html.

[29] HDFS. http://hadoop.apache.org/hdfs.

[30] He, Y., Leiserson, C. E., and Leiserson, W. M. The cilkview scalability analyzer. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2010), SPAA '10, ACM, pp. 145–156.

[31] Hill, J. M. D., Jarvis, S. A., Siniolakis, C. J., and Vasilev, V. P. Portable and architecture independent parallel performance tuning using a call-graph profiling tool. In *Parallel and Distributed Processing, 1998. PDP '98. Proceedings of the Sixth Euromicro Workshop on* (Jan 1998), pp. 286–294.

[32] Holt, R. C. Some deadlock properties of computer systems. *ACM Comput. Surv. 4*, 3 (Sept. 1972), 179–196.

[33] Huang, J., Mozafari, B., and Wenisch, T. F. Statistical analysis of latency through semantic profiling. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 64–79.

[34] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2010), USENIX-ATC'10, USENIX Association, pp. 11–11.

[35] Joao, J. A., Suleman, M. A., Mutlu, O., and Patt, Y. N. Bottleneck identification and scheduling in multithreaded applications. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS XVII, ACM, pp. 223–234.

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

[37] Kaldor, J., Mace, J., Bejda, M., Gao, E., Kuropatwa, W., O'Neill, J., Ong, K. W., Schaller, B., Shan, P., Viscomi, B., Venkataraman, V., Veeraraghavan, K., and Song, Y. J. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 34–50.

[38] Kambadur, M., Tang, K., and Kim, M. A. Harmony: Collection and analysis of parallel block vectors. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (Washington, DC, USA, 2012), ISCA '12, IEEE Computer Society, pp. 452–463.

[39] Kambadur, M., Tang, K., and Kim, M. A. Parashares: Finding the important basic blocks in multithreaded programs. In *Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings* (Cham, 2014), F. Silva, I. Dutra, and V. Santos Costa, Eds., Springer International Publishing, pp. 75–86.

[40] Kelley, J. E. Critical-path planning and scheduling: Mathematical basis. *Oper. Res. 9*, 3 (June 1961), 296–320.

[41] Kernel Probe. https://www.kernel.org/doc/Documentation/kprobes.txt.

[42] Kwon, Y., Balazinska, M., Howe, B., and Rolia, J. Skewtune: Mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2012), SIGMOD '12, ACM, pp. 25–36.

[43] Li, J., Chen, Y., Liu, H., Lu, S., Zhang, Y., Gunawi, H. S., Gu, X., Lu, X., and Li, D. Pcatch: Automatically detecting performance cascading bugs in cloud systems. In *Proceedings of the Thirteenth EuroSys Conference* (New York, NY, USA, 2018), EuroSys '18, ACM, pp. 7:1–7:14.

[44] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 190–200.

[45] Mace, J., Roelke, R., and Fonseca, R. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 378–393.

[46] Memaslap benchmark. http://docs.libmemcached.org/bin/memaslap.html.

[47] Memcached. http://memcached.org.

[48] Miller, B. P., Clark, M., Hollingsworth, J., Kierstead, S., Lim, S. S., and Torzewski, T. Ips-2: the second generation of a parallel program measurement system. *IEEE Transactions on Parallel and Distributed Systems 1*, 2 (Apr 1990), 206–217.

[49] Miller, B. P., and Hollingsworth, J. K. *Slack: A New Performance Metric for Parallel Programs*. University of Wisconsin-Madison, Computer Sciences Department, 1994.

[50] Miller, B. P., and Yang, C. IPS: an interactive and automatic performance measurement tool for parallel and distributed programs. In *Proceedings of the 7th International Conference on Distributed Computing Systems, Berlin, Germany, September 1987* (1987), pp. 482–489.

[51] MySQL. http://www.mysql.com.

[52] Mysql bug no. 81376. https://bugs.mysql.com/bug.php?id=81376.

[53] Scaling the HDFS NameNode (part 2). https://community.hortonworks.com/articles/43839/scaling-the-hdfs-namenode-part-2.htmll.

[54] Hadoop Tuning Notes. https://anandnalya.com/2011/09/hadoop-tuning-note/.

[55] Nethercote, N., and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2007), PLDI '07, ACM, pp. 89–100.

[56] Network File System. https://en.wikipedia.org/wiki/Network_File_System.

[57] Off-CPU Analysis. http://www.brendangregg.com/offcpuanalysis.html.

[58] Off-CPU Flame Graphs. http://www.brendangregg.com/FlameGraphs/offcpuflamegraphs.html.

[59] OProfile - A System Profiler for Linux. http://oprofile.sourceforge.net.

[60] Oyama, Y., Taura, K., and Yonezawa, A. Online computation of critical paths for multithreaded languages. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing* (London, UK, UK, 2000), IPDPS '00, Springer-Verlag, pp. 301–313.

[61] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org.

[62] perf-map-agent: A Java Agent to Generate Method Mappings to Use with the Linux 'perf' Tool. https://github.com/jvm-profiling-tools/perf-map-agent.

[63] Rong Shi, Y. G., and Wang, Y. Evaluating scalability bottlenecks by workload extrapolation. In *26th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOT '18)* (Milwaukee, WI, 2018), IEEE.

[64] Shvachko, K. HDFS scalability: the limits to growth. http://c59951.r51.cf2.rackcdn.com/5424-1908-shvachko.pdf.

[65] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (Washington, DC, USA, 2010), MSST '10, IEEE Computer Society, pp. 1–10.

[66] Soares, L., and Stumm, M. Flexsc: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 33–46.

[67] Szebenyi, Z., Wolf, F., and Wylie, B. J. N. Space-efficient time-series call-path profiling of parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (Nov 2009), pp. 1–12.

[68] Transaction Processing Performance Council. The TPC-C home page. http://www.tpc.org/tpcc/.

[69] Valiant, L. G. A bridging model for parallel computation. *Commun. ACM 33*, 8 (Aug. 1990), 103–111.

[70] Wang, G., Xie, W., Demers, A. J., and Gehrke, J. Asynchronous large-scale graph processing made easy. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings* (2013).

[71] Xie, C., Su, C., Kapritsos, M., Wang, Y., Yaghmazadeh, N., Alvisi, L., and Mahajan, P. Salt: Combining acid and base in a distributed database. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, 2014), USENIX Association, pp. 495–509.

[72] Xie, C., Su, C., Littley, C., Alvisi, L., Kapritsos, M., and Wang, Y. High-performance acid via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 279–294.

[73] Xie, W., Wang, G., Bindel, D., Demers, A., and Gehrke, J. Fast iterative graph computation with block updates. *Proc. VLDB Endow. 6*, 14 (Sept. 2013), 2014–2025.

[74] Xiong, W., Park, S., Zhang, J., Zhou, Y., and Ma, Z. Ad hoc synchronization considered harmful. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 163–176.

[75] Yourkit Java and .Net Profiler. https://www.yourkit.com/.

[76] Yu, T., and Pradel, M. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (New York, NY, USA, 2016), ISSTA 2016, ACM, pp. 389–400.

[77] Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R., and Stoica, I. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 29–42.

[78] Zhao, X., Rodrigues, K., Luo, Y., Yuan, D., and Stumm, M. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, 2016), USENIX Association, pp. 603–618.

[79] Zookeeper. http://hadoop.apache.org/zookeeper.

A Appendix: Edge weights of wait-for graphs

Table 4 to Table 13 record the edge weights of the wait-for graphs wPerf generated in each experiment.

B Appendix: Reports of COZ

Figures 11 to 14 show the reports COZ generated for grep over NFS, BlockGrace, Memcached, and MySQL. COZ currently does not support Java. Since COZ can test different lines of code and generate many graphs accordingly, we sort these graphs by their maximal speedup and show the top six for each application.

C Appendix: Off-CPU flame graphs of perf

Figures 15-21 show the off-CPU flame graphs perf generated for all the applications. As one can observe, for H-Base, grep over NFS, and BlockGrace, the longest events in the flame graphs are the same as the bottleneck edges reported by wPerf. For ZooKeeper, HDFS NameNode, MySQL, and Memcached, the longest events in the flame graphs are not the same as the bottleneck edges reported by wPerf.

D Appendix: Reports of SyncPerf

Figures 22-24 show SyncPerf's reports on grep over N-FS, BlockGrace, and Memcached. We were not able to run SyncPerf with MySQL and Java applications.

SrcThread	DstThread	Edge Weight
VMThread	Handler	1069.02
RespProc(Log)	NIC	824.77
ConcurrentGC	VMThread	400.01
LogRoller	Handler	173.04
RespProc(MemStore)	Streamer(MemStore)	96.25
IPC Client(HDFS)	NIC	92.62
ParallelGC	VMThread	88.69
Streamer(MemStore)	CacheFlusher	88.07
Handler	RespProc(Log)	87.42
ParallelCMS	ConcurrentGC	78.03
Streamer(Log)	Handler	71.65
NIC	Streamer(Log)	50.36
CacheFlusher	Handler	30.95
VMThread	RespProc(Log)	25.61
VMThread	CacheFlusher	23.04
Streamer(Log)	RespProc(Log)	15.88
VMThread	ParallelGC	15.74
NIC	Streamer(MemStore)	14.68
VMThread	IPC Client(HDFS)	14.08
RespProc(Log)	Handler	13.71
Streamer(MemStore)	NIC	13.69
VMThread	Streamer(Log)	13.27
CacheFlusher	RespProc(MemStore)	12.57
RespProc(Log)	VMThread	9.30
VMThread	NIC	7.24
RespProc(Log)	Streamer(Log)	6.20
RespProc(MemStore)	NIC	5.80
IPC Client(HDFS)	VMThread	4.95
CacheFlusher	Streamer(MemStore)	4.31
Streamer(Log)	LogRoller	3.99
NIC	Handler	3.50
Handler	Streamer(Log)	3.32
RespProc(Log)	IPC Client(HDFS)	2.81
IPC Client(HDFS)	RespProc(Log)	2.66
VMThread	ConcurrentGC	1.25
CacheFlusher	NIC	1.21
IPC Client(HDFS)	Handler	1.09
Handler	VMThread	0.97
VMThread	LogRoller	0.78

Table 4: Edge weights of HBase with 10 handlers

SrcThread	DstThread	Edge Weight
VMThread	Handler	996.74
RespProc(Log)	NIC	596.38
ConcurrentGC	VMThread	359.80
LogRoller	Handler	230.73
IPC Client(HDFS)	NIC	225.56
CacheFlusher	Handler	206.35
VMThread	ParallelGC	203.68
Streamer(Log)	Handler	110.11
RespProc(MemStore)	Streamer(MemStore)	93.53
ParallelGC	VMThread	86.10
ParallelCMS	ConcurrentGC	83.11
Streamer(MemStore)	CacheFlusher	82.97
IPC Client(HDFS)	VMThread	79.97
RespProc(Log)	Handler	75.29
RespProc(Log)	VMThread	56.60
LogRoller	RespProc(Log)	55.29
Streamer(Log)	NIC	46.02
Streamer(Log)	RespProc(Log)	45.08
NIC	Streamer(Log)	36.94
RespProc(Log)	IPC Client(HDFS)	28.67
VMThread	CacheFlusher	26.96
RespProc(Log)	Streamer(Log)	25.82
Streamer(MemStore)	NIC	23.36
IPC Client(HDFS)	RespProc(Log)	22.39
IPC Client(HDFS)	Handler	21.13
CacheFlusher	RespProc(MemStore)	18.94
VMThread	LogRoller	17.61
Handler	RespProc(Log)	16.54
RespProc(MemStore)	NIC	14.09
CacheFlusher	NIC	13.24
CacheFlusher	LogRoller	10.90
NIC	Streamer(MemStore)	10.70
VMThread	NIC	8.30
CacheFlusher	Streamer(MemStore)	8.16
VMThread	IPC Client(HDFS)	6.55
Handler	Streamer(Log)	5.70
VMThread	RespProc(Log)	5.03
Handler	IPC Client(HDFS)	4.84
Streamer(Log)	LogRoller	3.45
VMThread	ConcurrentGC	3.14
RespProc(Log)	RespProc(MemStore)	3.02
Handler	VMThread	2.73
NIC	Handler	2.40
VMThread	Streamer(Log)	2.19

Table 5: Edge weights of HBase with 60 handlers.

SrcThread	DstThread	Edge Weight
ParallelGC	ParallelGC	2463.49
VMThread	SyncThread	416.98
ProcessThread	NIOServerCxn	291.52
VMThread	NIOServerCxn	202.48
SyncThread	ProcessThread	199.81
VMThread	ProcessThread	177.22
NIOServerCxn	NIC	110.86
ParallelGC	VMThread	88.92
NIOServerCxn	SyncThread	18.72
CompilerThread	NIOServerCxn	15.03
CompilerThread	SyncThread	14.98
NIC	SyncThread	4.30
VMThread	ParallelGC	3.31
NIC	NIOServerCxn	3.04
NIOServerCxn	VMThread	2.01
SyncThread	VMThread	1.07
SyncThread	NIOServerCxn	1.05
ProcessThread	VMThread	0.87
NIOServerCxn	ProcessThread	0.26
ProcessThread	SyncThread	0.02

Table 6: Edge weights of ZooKeeper with readonly workload.

SrcThread	DstThread	Edge Weight
ParallelGC	ParallelGC	2548.91
JBDThread	Disk	767.39
SyncThread	JBDThread	721.92
NIOServerCxn	SyncThread	561.17
VMThread	SyncThread	369.92
SyncThread	Disk	272.87
ProcessThread	NIOServerCxn	262.45
VMThread	NIOServerCxn	241.97
VMThread	ProcessThread	199.60
ParallelGC	VMThread	91.70
Disk	JBDThread	60.97
NIC	SyncThread	44.56
JBDThread	SyncThread	43.09
Disk	SyncThread	21.68
CompilerThread	SyncThread	20.26
CompilerThread	NIOServerCxn	13.48
CompilerThread	ProcessThread	8.31
VMThread	ParallelGC	3.13
SyncThread	VMThread	2.79
NIOServerCxn	VMThread	1.36
ProcessThread	VMThread	0.66
NIOServerCxn	ProcessThread	0.34
SyncThread	NIOServerCxn	0.26
SyncThread	ProcessThread	0.05
VMPeriodicThread	Disk	0.03
ProcessThread	SyncThread	0.02

Table 7: Edge weights of ZooKeeper with 0.1% write workload.

SrcThread	DstThread	Edge Weight
Handler	Handler	4516.08
ParallelGC	ParallelGC	2507.06
VMThread	Handler	206.31
SocketReader	Handler	133.39
BlockReportProc	Handler	108.25
Handler	Disk	100.64
NIC	Handler	89.58
ParallelGC	VMThread	89.40
Disk	Handler	81.78
JBDThread	Handler	78.97
VMThread	SocketReader	49.32
VMThread	VMThread1	11.17
JBDThread	Disk	9.38
VMThread	BlockReportProc	7.07
JBDThread	BlockReportProc	4.94
VMThread	ParallelGC	2.43
Handler	BlockReportProc	0.89
Disk	JBDThread	0.76
Handler	VMThread	0.25
SocketReader	VMThread	0.24
BlockReportProc	SocketReader	0.11
BlockReportProc	VMThread	0.11
Kworker	Disk	0.07
Handler	SocketReader	0.02
VMThread1	VMThread	0.01
Disk	Kworker	0.01

Table 8: Edge weights of HDFS NameNode.

SrcThread	DstThread	Edge Weight
KworkerHP(Recv)	NIC	587.24
Kworker(Recv)	KworkerHP(Recv)	385.96
Grep	Kworker(Recv)	261.49
Grep	KworkerHP(Recv)	112.74
Kworker1(TTY)	Grep	90.61
Kworker(Send)	Grep	89.07
Kworker2(TTY)	Grep	87.77
NIC	Kworker(Send)	61.27
NIC	Grep	22.77
Kworker3(TTY)	Grep	19.11
NIC	Kworker(Recv)	5.13
NIC	KworkerHP(Send)	0.30
KworkerHP(Send)	Grep	0.10
NIC	KworkerHP(Recv)	0.02
KworkerHP(Send)	Kworker(Send)	0.02

Table 9: Edge weights of NFS client.

SrcThread	DstThread	Edge Weight
Worker	Main	64.99
Main	Worker	301.99

Table 10: Edge weights of BlockGrace with the SSSP workload.

SrcThread	DstThread	Edge Weight
Worker	Worker	3.85
NIC	Worker	88.25

Table 11: Edge weights of Memcached.

SrcThread	DstThread	Edge Weight
KworkerHP	KworkerHP	1033.50
JBDThread	Disk	719.38
PurgeCoord	Worker	330.61
Worker	Worker	279.16
NIC	Worker	89.90
Worker	IOThread	78.83
CleanerCoord	JBDThread	68.66
Worker	JBDThread	64.45
Disk	KworkerHP	62.80
KworkerHP	Disk	41.65
MasterThread	IOThread	31.84
Disk	JBDThread	16.19
MasterThread	JBDThread	14.75
IOThread	CleanerCoord	13.62
JBDThread	Worker	9.51
CleanerCoord	Disk	8.46
JBDThread	MasterThread	7.41
IOThread	IOThread	7.02
MasterThread	Worker	6.34
JBDThread	CleanerCoord	5.52
IOThread	Worker	4.49
Worker	NIC	3.69
IOThread	MasterThread	2.98
CleanerCoord	Worker	2.33
Worker	MasterThread	1.73
Kworker	Disk	1.14
Worker	Disk	0.67
MasterThread	CleanerCoord	0.53
CleanerCoord	MasterThread	0.47
MasterThread	Disk	0.44
Disk	CleanerCoord	0.19
Disk	Worker	0.09
JBDThread	Kworker	0.89
Worker	CleanerCoord	0.08
Disk	Kworker	0.03
Disk	MasterThread	0.01

Table 12: Edge weights of MySQL on disk.

SrcThread	DstThread	Edge Weight
Worker	Worker	1378.99
PurgerCoord	Worker	228.30
IOThread	IOThread	106.43
SrvWorker	PurgerCoord	83.96
Cleaner	Worker	79.87
NIC	Worker	75.07
IOThread	Cleaner	67.85
IOThread	Worker	53.80
DictThread	Worker	35.01
PurgerCoord	IOThread	17.37
Worker	IOThread	16.70
Worker	NIC	14.49
SrvWorker	IOThread	11.59
LockWait	Worker	11.39
SrvWorker	SrvWorker	11.21
PurgerCoord	SrvWorker	6.73
Cleaner	IOThread	5.41
SrvWorker	Worker	5.09
IOThread	PurgerCoord	0.92
Worker	SrvWorker	0.92
SrvMaster	Worker	0.86
Worker	Cleaner	0.71
PurgerCoord	Cleaner	0.66
Cleaner	SrvWorker	0.42
Worker	PurgerCoord	0.36
Cleaner	PurgerCoord	0.34
SrvWorker	Cleaner	0.08
Cleaner	SrvMaster	0.07
SrvMaster	Cleaner	0.05
DictThread	PurgerCoord	0.03
PurgerCoord	DictThread	0.02
Worker	DictThread	0.01
Cleaner	DictThread	0.01
DictThread	SrvWorker	0.01
Worker	SrvMaster	0.01
IOThread	DictThread	0.01
SrvWorker	DictThread	0.00

Table 13: Edge weights of MySQL in memory (128MB buffer).

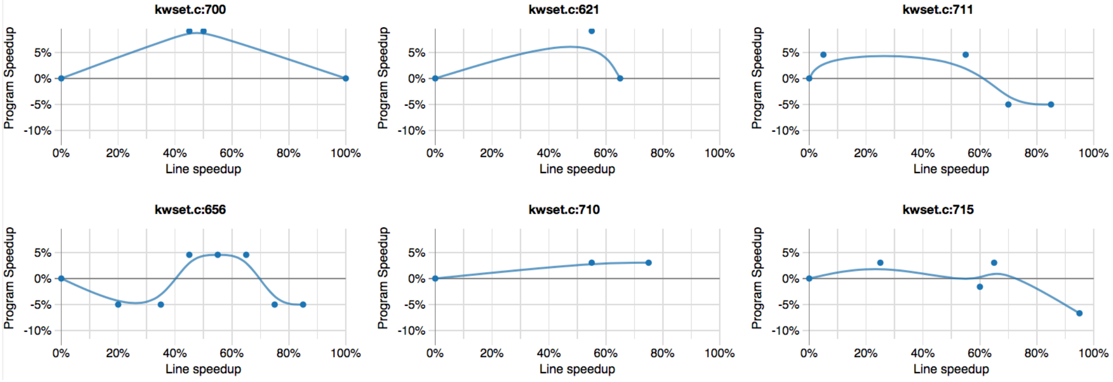


Figure 11: COZ output for grep over NFS

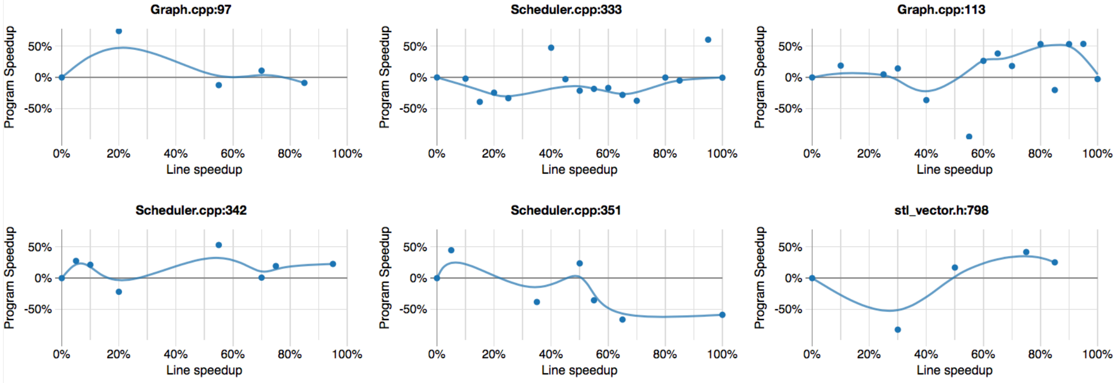


Figure 12: COZ output for BlockGrace

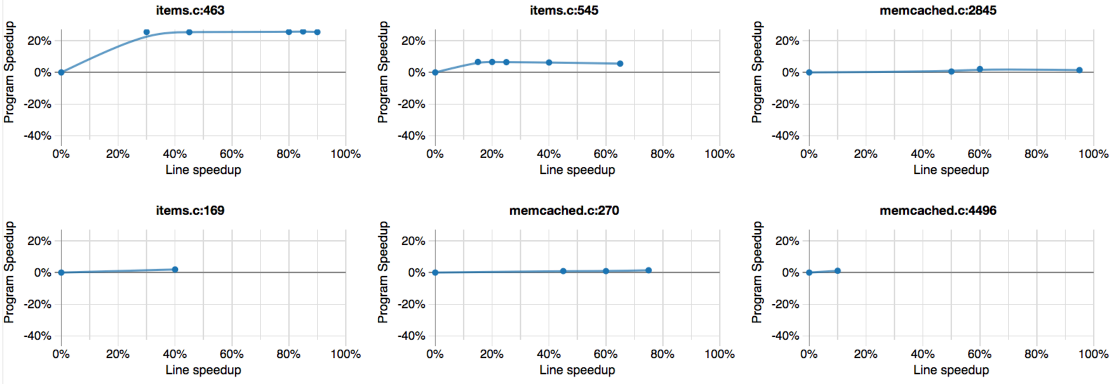


Figure 13: COZ output for memcached

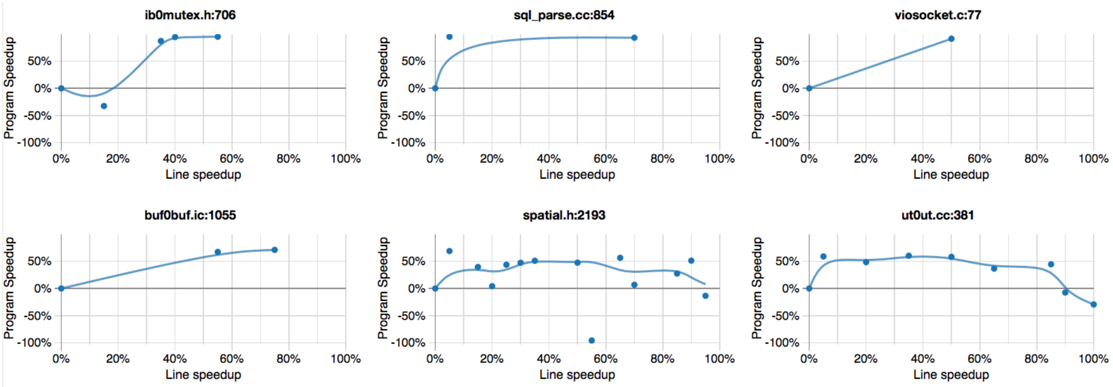


Figure 14: COZ output for MySQL

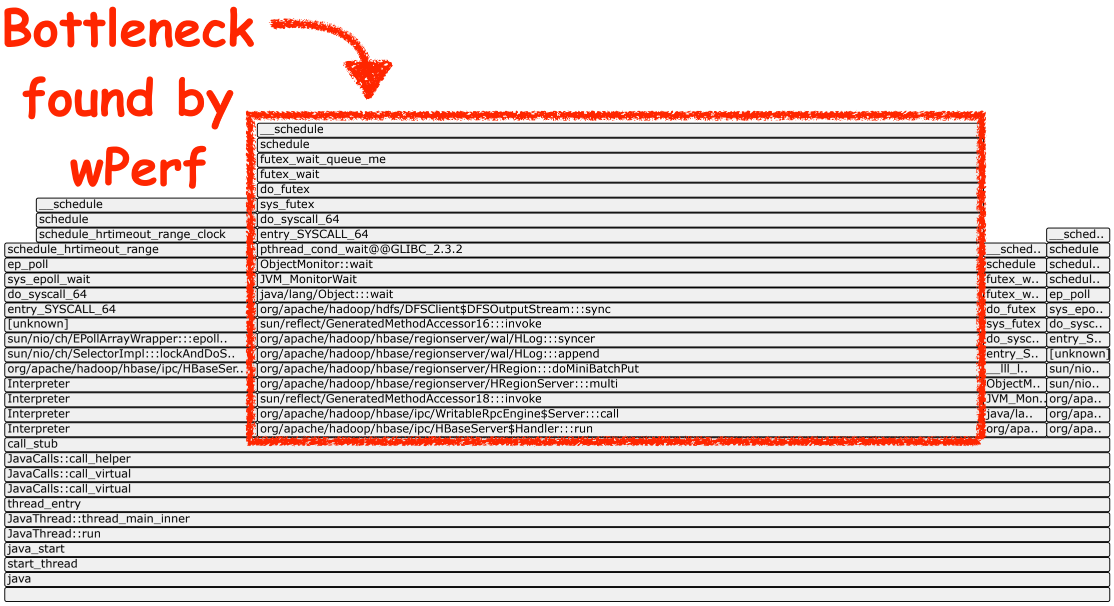


Figure 15: Off-CPU flame graph of HBase (10 handlers).

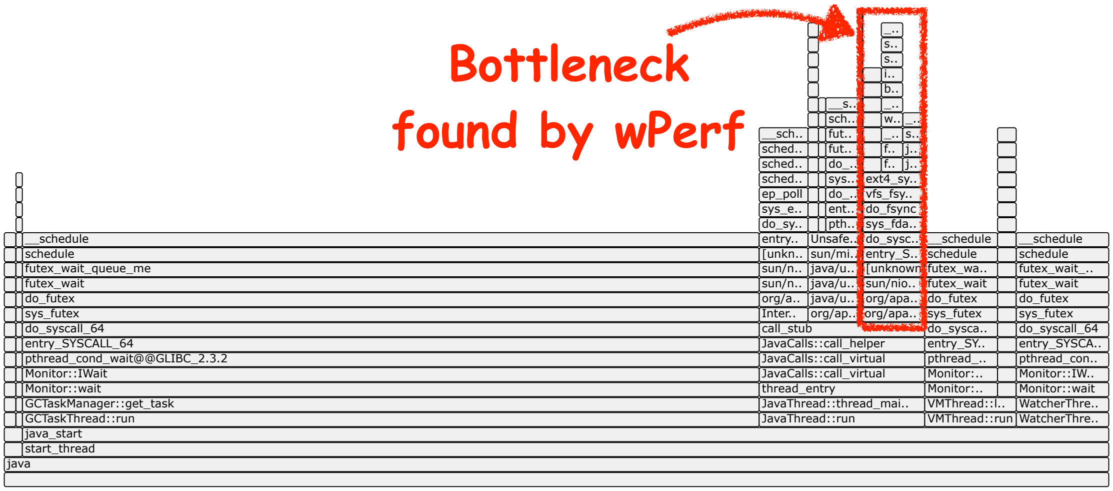

Bottleneck found by wPerf

Figure 16: Off-CPU flame graph of ZooKeeper (0.1% write).

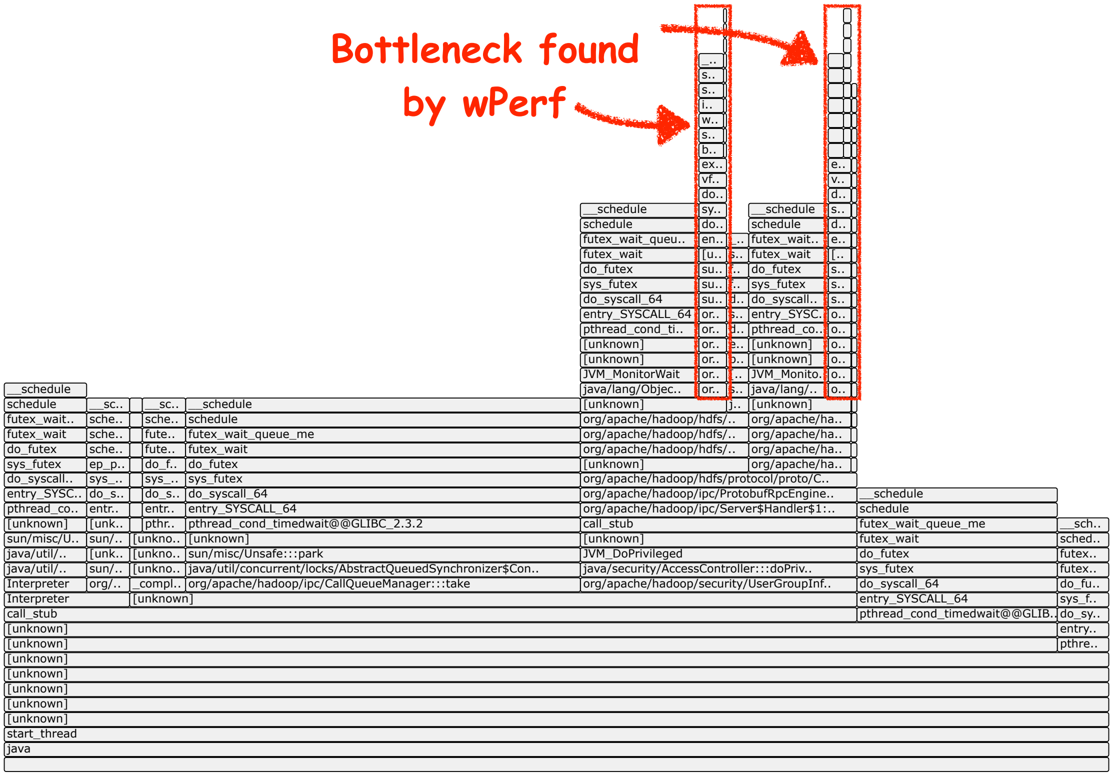
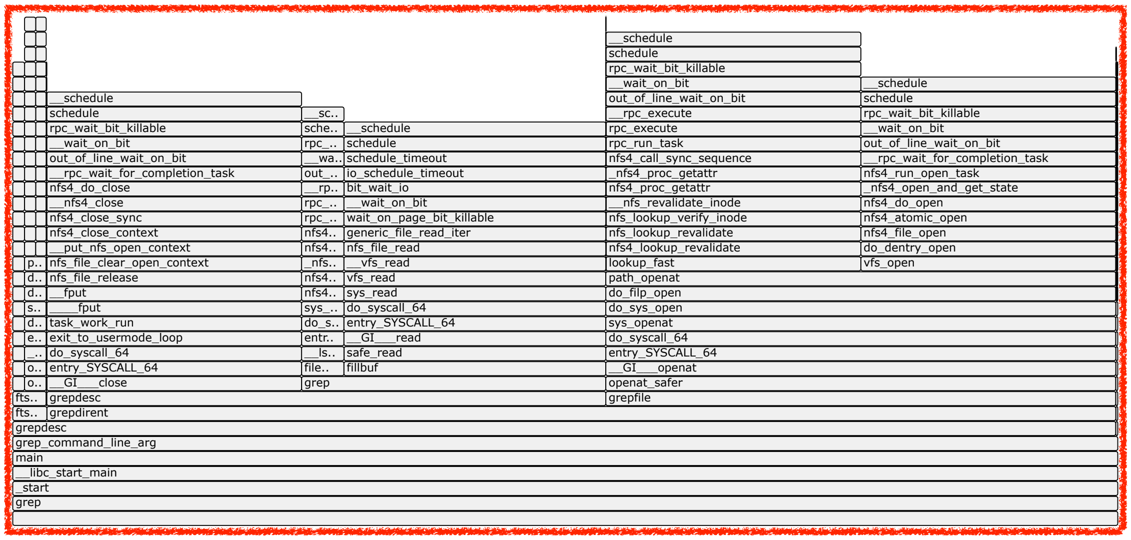
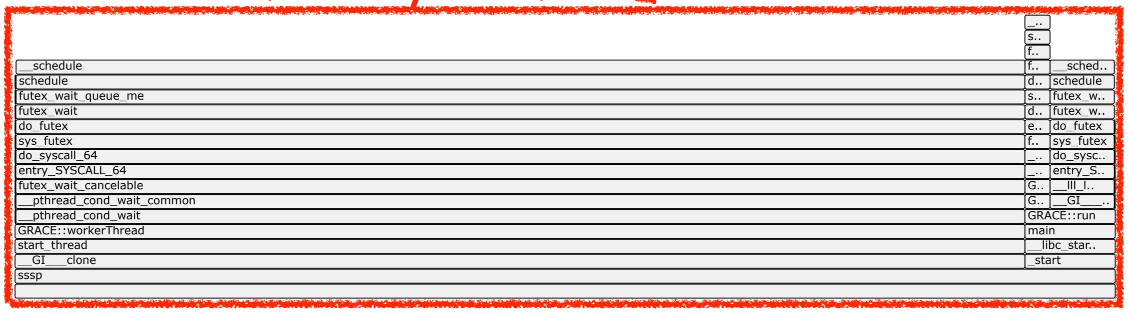

Bottleneck found by wPerf

Figure 17: Off-CPU flame graph of HDFS NameNode (10 handlers).



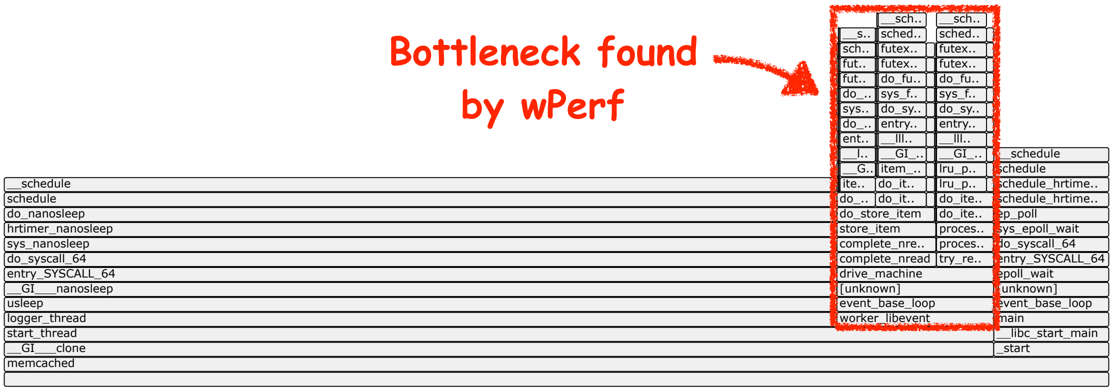
Bottleneck found by wPerf

Figure 18: Off-CPU flame graph of grep over NFS.



Bottleneck found by wPerf

Figure 19: Off-CPU flame graph of BlockGrace.



Bottleneck found by wPerf

Figure 20: Off-CPU flame graph of Memcached.

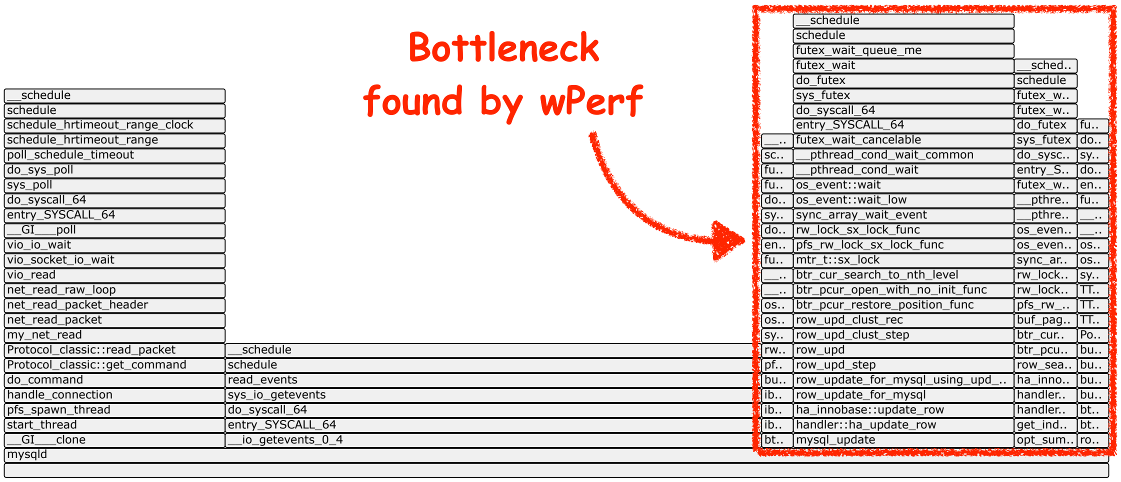


Figure 21: Off-CPU flame graph of MySQL.

```
==============================
HIGH CONFLICT , HIGH FREQUENCY
==============================
Total found : 0


==============================
HIGH CONFLICT , LOW FREQUENCY
==============================
Total found : 0


==============================
LOW CONFLICT , HIGH FREQUENCY
==============================
Total found : 0
```

Figure 22: SyncPerf output for grep over NFS.

```
==============================
HIGH CONFLICT , HIGH FREQUENCY
==============================
Total found : 0


==============================
HIGH CONFLICT , LOW FREQUENCY
==============================
Total found : 5
No.1
-------
Conflict Rate: 19.6286
Acquisition Frequency: 0.127034
Line Numbers: 3
Driver.cpp:86
TaskBase.cpp:46 Scheduler.h:114 Driver.cpp:472 main.cpp:171
Driver.cpp:494 main.cpp:171
No.2
-------
Conflict Rate: 11.4058
Acquisition Frequency: 0.127034
Line Numbers: 3
Driver.cpp:86
TaskBase.cpp:46 Scheduler.h:114 Driver.cpp:472 main.cpp:171
Driver.cpp:494 main.cpp:171
No.3
-------
Conflict Rate: 8.22281
Acquisition Frequency: 0.127034
Line Numbers: 3
Driver.cpp:86
TaskBase.cpp:46 Scheduler.h:114 Driver.cpp:472 main.cpp:171
Driver.cpp:494 main.cpp:171
No.4
-------
Conflict Rate: 8.22281
Acquisition Frequency: 0.127034
Line Numbers: 3
Driver.cpp:86
TaskBase.cpp:46 Scheduler.h:114 Driver.cpp:472 main.cpp:171
Driver.cpp:494 main.cpp:171
No.5
-------
Conflict Rate: 7.42706
Acquisition Frequency: 0.127034
Line Numbers: 3
Driver.cpp:86
TaskBase.cpp:46 Scheduler.h:114 Driver.cpp:472 main.cpp:171
Driver.cpp:494 main.cpp:171


==============================
LOW CONFLICT , HIGH FREQUENCY
==============================
Total found : 0


======================
Asymmetric Locks found : 3
========================
Driver.cpp:494 main.cpp:171
Driver.cpp:86
TaskBase.cpp:46 Scheduler.h:114 Driver.cpp:472 main.cpp:171
```

Figure 23: SyncPerf output for BlockGrace.

```
==============================
HIGH CONFLICT , HIGH FREQUENCY
==============================
Total found : 0


==============================
HIGH CONFLICT , LOW FREQUENCY
==============================
Total found : 1
No.1
-------
Conflict Rate: 40
Acquisition Frequency: 3.17436e-05
Line Numbers: 2
thread.c:111,memcached.c:6567,
thread.c:119,thread.c:351,??:0,


==============================
LOW CONFLICT , HIGH FREQUENCY
==============================
Total found : 119
No.1
-------
Conflict Rate: 4.61315
Acquisition Frequency: 685.809
Line Numbers: 4
items.c:995,items.c:212,items.c:285,memcached.c:3499,memcached.c:3993,
items.c:392,items.c:448,memcached.c:2724,thread.c:107,memcached.c:1116,
items.c:428,items.c:490,items.c:546,memcached.c:2722,thread.c:107,
No.2
-------
Conflict Rate: 0.427492
Acquisition Frequency: 457.228
Line Numbers: 8
logger.c:544,string3.h:90,
assoc.c:73,memcached.c:399,
logger.c:450,??:0,
memcached.c:522,memcached.c:5266,memcached.c:5286,memcached.c:6624,
memcached.c:602,memcached.c:5266,memcached.c:5286,memcached.c:6624,
items.c:438,memcached.c:2724,thread.c:107,memcached.c:1116,memcached.c:4844,
items.c:458,items.c:546,memcached.c:2722,thread.c:107,memcached.c:1116,
memcached.c:706,memcached.c:5032,
No.3
-------
Conflict Rate: 0.893449
Acquisition Frequency: 457.204
Line Numbers: 4
slabs.c:528 items.c:189 items.c:285 memcached.c:3499 memcached.c:3993,
slabs.c:535 memcached.c:2733 thread.c:107 memcached.c:1116 memcached.c:4844,
```

Figure 24: SyncPerf output for Memcached. The report contains many events and we only show the top 3.