LDB: An Efficient Latency Profiling Tool for Multithreaded Applications

Inho Cho, *MIT CSAIL;* Seo Jin Park, *University of Southern California;*
Ahmed Saeed, *Georgia Tech;* Mohammad Alizadeh and Adam Belay, *MIT CSAIL*

https://www.usenix.org/conference/nsdi24/presentation/cho

This paper is included in the
Proceedings of the 21st USENIX Symposium on
Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

LDB: An Efficient Latency Profiling Tool for Multithreaded Applications

Inho Cho[1] Seo Jin Park[2] Ahmed Saeed[3] Mohammad Alizadeh[1] Adam Belay[1]
[1] *MIT CSAIL* [2] *University of Southern California* [3] *Georgia Tech*

Abstract

Maintaining low tail latency is critical for the efficiency and performance of large-scale datacenter systems. Software bugs that cause tail latency problems, however, are notoriously difficult to debug. We present LDB, a new latency profiling tool that aims to overcome this challenge by precisely identifying the specific functions that are responsible for tail latency anomalies. LDB observes the latency of all functions in a running program. It uses a novel, software-only technique called *stack sampling*, where a busy-spinning stack scanner thread polls lightweight metadata recorded in the call stack, shifting tracing costs away from program threads. In addition, LDB uses *event tagging* to record requests, inter-thread synchronization, and context switching. This can be used, for example, to generate per-request timelines and to find the root cause of complex tail latency problems such as lock contention in multi-threaded programs. We evaluate LDB with three datacenter applications, finding latency problems in each. Our results further show that LDB produces actionable insights, has low overhead, and can rapidly analyze recordings, making it feasible to use in production settings.

1 Introduction

Modern datacenter services like search, social networks, and DNN training operate on huge datasets with complex communication patterns and large numbers of servers [17, 28]. Tail latency is a key challenge in this setting because overall performance is often limited by the slowest response [22]. Despite the tremendous effort that goes into optimizing latency-sensitive programs, operators tend to treat high tail latency as inevitable due to the complexity of deployed programs. Therefore, the main method available to operators today is to keep machine utilization low to control for tail latency, wasting both power efficiency and money.

In this paper, our aim is to empower developers to tackle tail latency problems head-on by answering the following question: *Can a debugging tool identify the precise source of tail latency experienced by a request in a server (e.g., the line of code that is responsible)?* This is a significant challenge, as the effort needed to understand tail behaviors is formidable with the tools that exist today. Statistical profilers (e.g., Linux's perf-tool), for example, have only limited utility because their method of periodic sampling captures the average runtime of functions, which may deviate significantly from the tail runtime. Further, they don't account for request semantics, so they cannot differentiate between requests running on the critical path versus the background, making it hard to identify bottlenecks. Instead, developers commonly hand instrument code locations that they *suspect* are problematic, but they can only try a few locations at a time due to instrumentation overhead. Thus, a typical workflow involves multiple iterations of instrumentation location adjustment, deployment, and data collection.

One way to avoid this tedious process would be to use a tool that can instrument all functions simultaneously (e.g., XRay [18]). However, this approach causes significant overhead that can distort an application's behavior. A less invasive option would be to use hardware assistance. For example, Intel recently introduced a hardware extension called Intel Processor Trace (Intel PT) that records every control flow operation (calls, branches, jumps, etc.) to an in-memory log for analysis. NSight recently demonstrated that Intel PT can be used to derive rich tail latency insights, such as a precise timeline of how cycles are spent handling network requests [26].

Unfortunately, Intel PT has drawbacks that make it difficult to use for profiling latency in practice. First, Intel PT is proprietary and requires hardware support, so it is only available on certain platforms. Second, Intel PT generates data at an enormous rate, so it is only feasible to record a few seconds of samples. Finally, Intel PT's compression scheme requires a software decoder that walks a program's object code to reconstruct its control flow. This requires several hours of processing—even for a few seconds of data—prohibiting interactive profiling.

We present LDB, a new latency debugging tool that provides unprecedented visibility into the latency behavior of applications. LDB reports the distribution of the latency of all

```
std::mutex lock;
std::map<int, std::string> db;

void snapshot() {
  std::ofstream out("snapshot.txt");
  std::lock_guard<std::mutex> g(lock);
  for (const auto& kv : db)
    out << kv.first << "," << kv.second << std::endl;
  out.close();
}

void background_thread() {
  while (true) {
    snapshot();
    usleep(10000);
  }
}

void request_handler(int key, std::string& value) {
  std::lock_guard<std::mutex> g(lock);
  db[key] = value;
}

int main() {
  std::thread bg_thread(background_thread);
  for (int i = 0; i < kRounds; i++) {
    int key = std::rand() % dbSize;
    std::string value = generate_random_string();
    request_handler(key, value);
  }
}
```

Figure 1: example.cc: a simple multithreaded program where a foreground thread handles user requests and a background thread snapshots program state every 10 ms.

functions in a process. Furthermore, it allows developers to breakdown the latency faced by a specific request, even when processed by multiple threads, allowing them to zoom in and identify the code responsible for anomalous behavior. LDB provides this information after only seconds of decoding its own traces and without significantly harming the performance of the profiled program, enabling monitoring in production environments. In contrast to Intel PT, LDB is hardware agnostic. In principle, it can be ported to any architecture, and we demonstrate its use on Intel and AMD processors.

The efficiency and portability of LDB stem from a novel, software-only technique, called *stack sampling*. Unlike prior approaches, stack sampling doesn't record per-function timestamps (e.g., the output of the RDTSC instruction [18, 35]). Instead, a separate stack scanner thread polls the call stack of every application thread. During each polling cycle, the stack scanner thread performs a backtrace on each call stack to inspect changes to stack frames. Intuitively, a function's stack frame will be resident on the call stack until the function returns, so the higher its execution time, the longer its stack frame will remain resident. LDB exploits this to capture the runtime of all the functions that contribute meaningfully to latency (i.e., those that last longer than its sub-us polling interval).

While stack sampling is based on a simple premise, we had to overcome several challenges to make it work in practice. First, it is not possible for one core to access another core's stack pointer register, so we had to find an alternative way to locate the deepest stack frame. Second, there is not enough information available in stack frames to discern between repeated invocations of the same function so we had to find a way to detect them. Third, the stack scanning thread could race with application threads causing it to observe corrupted stack frames, so we had to develop a mechanism to detect and discard bad samples. Finally, backtracing can cause false sharing with variables on the call stack, negatively impacting application performance, so we needed a way to limit this overhead without sacrificing resolution. We discuss our solution to each of these problems in §3.2.

In addition to efficiency improvements, LDB provides better visibility into latency problems through *event tagging*, recording several types of events with timestamps and event-specific metadata. Examples include the start and end of requests; cross-thread interactions like locks; and the transfer of request ownership among threads. This allows LDB to track the timeline of each request and correlate this information across multiple threads. For example, LDB can identify a slow function running inside a critical section that is protected by a lock, and then tie it back to a request that is blocked in another thread waiting to acquire the same lock. LDB also uses event tagging to track context switching, allowing it to identify delays from the OS scheduler.

We demonstrate the value of LDB by profiling two latency-sensitive applications (Memcached and Lucene) and a best-effort application called Qperf, which is a benchmark for the QUIC transport protocol. We show that LDB can detect complex interactions between threads and identify which functions are responsible for impacting performance (both latency and throughput). Then, we provide an evaluation of the performance overhead of LDB when used to profile these three applications. In particular, we show that the overhead of LDB is less than the overhead of Coz and Xray; and comparable to Intel PT on recent Intel architectures. LDB maintains its low overhead across Intel and AMD architectures. On the other hand, the overhead of Intel PT is considerably higher when used on older Intel architectures.

LDB has some limitations. First, it can't yet capture hardware interrupts and some other types of traps into the kernel, which could contribute to tail latency. Second, LDB requires programs or libraries to be recompiled to support stack sampling, so it cannot trace latency inside unmodified binaries or libraries. Finally, if one chooses not to annotate requests in the source code (typically just a few lines), LDB cannot capture information about request latency, but it can still deliver statistics about the latency of each line of code, which is enough for debugging many tail latency problems.

LDB is available as open-source software at https://inhocho89.github.io/ldb/.

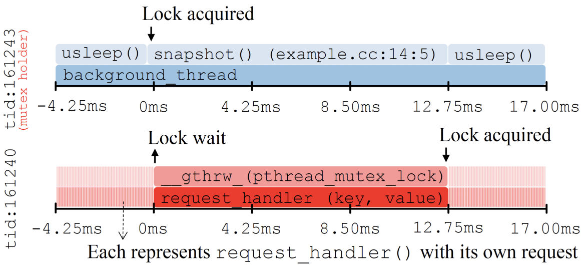


Figure 2: A timeline visualization of the time spent in each function during the longest request. The longest request starts at 0 ms and finishes at 12.75 ms. The thread on the top is the mutex holder, while the thread on the bottom is the request handler, which is blocked waiting to acquire the mutex.

2 Background and Motivation

2.1 Debugging a Tail Latency Problem

Consider the example shown in Figure 1, based on a pattern found in many real programs. A request processing thread and a background thread require synchronized access to the same data. The request processing thread responds by executing `request_handler()`, which inserts items into a `std::map`. Concurrently, the background thread takes a snapshot of the `std::map` every 10 ms. Access to the `std::map` is serialized through a `std::mutex`. As a result, the 99.99th percentile latency of the `request_handler()` function is 10 ms while its median is 244 ns (a 40,000× increase)!

This is a challenging issue to debug because of the rare interaction between the two threads. LDB, however, can easily identify the root cause. It captures everything that happened in the program and can generate a timeline visualization for each request that includes all the involved threads. By plotting the timeline of the longest request (Figure 2), it becomes clear that the snapshot thread (shown on top) delayed `request_handler()` (shown on bottom) by holding the mutex it was trying to acquire. This suggests that tail latency can be improved by optimizing `snapshot()` or reducing the size of its critical section.

However, existing profilers struggle to debug tail latency issues like these. For example, Figure 3 shows the output of perf, one of the widely used performance debugging tools. The majority of time is spent in `generate_random_string()` and other functions under `request_handler()`. `snapshot()` accounts for only 0.6% and was buried under other 13 functions. This result reveals three interesting problems of using perf for tail latency debugging. First, tail behavior is amortized, so it gets buried down under average behaviors. Second, perf is measuring where the CPU cycles go, not how long each function takes, so it is unable to show the time spent on blocking I/O or synchronization. Figure 2 suggests that `snapshot()` runs for over 10 ms and then sleeps for 10 ms, so it should account for at least about 50% time on average. However, much of the time spent on `snapshot()` is spent blocking on I/O, so perf reports only 0.62%. Lastly, perf cannot capture the interplay across threads caused by the mutex.

Function	CPU Time ▽
`generate_random_string`	63.75%
`request_handler`	7.43%
`std::_Rb_tree_increment`	2.82%
...(13 more functions)...	
`snapshot`	0.62%

Figure 3: Perf's output with the example application.

2.2 Intuition and Challenges

We observe that the metadata in x86 stack frames (e.g., the number of stack frames, return instruction pointers, saved based pointers, etc.) remains unchanged as long as a thread is executing a bottlenecked function. LDB takes an approach in which a separate dedicated busy-running thread, called the stack scanning thread, periodically scans these stack frames. It then measures the latency of the function call by examining whether the metadata in the stack frame metadata does not change. If a change is detected in the metadata, it signifies that a function has either returned or that a new function call has been invoked.

The realization of this *stack sampling* idea entails the following challenges:

1. Finding the deepest stack frame. Stack frames form a singly linked list data structure. Starting with the most recent (deepest) stack frame, one can traverse the entire call stack by following the saved base pointers. This traversal is necessary for LDB to ascertain whether the stack frame metadata has been modified or not. The location of the deepest stack frame can be retrieved from the RBP register. However, threads other than the application thread itself cannot access this register, making it challenging for the stack scanning thread to determine where to start traversing the stack frames.

2. Differentiating stack frames for different function calls. When the same function is repeatedly invoked from the same line of code (such as within a loop), the metadata may remain identical across samples of the stack frames. This can lead to an overestimation of the function call latency measurement performed by the stack scanning thread, as it may fail to detect that there were separate invocations to the same function. Therefore, to accurately measure the latency of individual function calls, it is necessary to find a way to differentiate between stack frames from distinct function calls, even if their stack frame metadata appears to be the same.

3. Cache thrashing and false sharing. Stack frames are frequently accessed by the application thread for local variables. If the stack scanning thread accesses stack frames too often, it may lead to performance degradation because of cache thrashing and false sharing. Repeated access to stack frames by the stack scanning thread can cause the data to be continually invalidated from the application thread's cache, harming the performance of the monitored application.

4. Data race for the stack frames. While the stack scanning thread is traversing the stack frame, the stack frame can be concurrently modified by the application thread when a func-

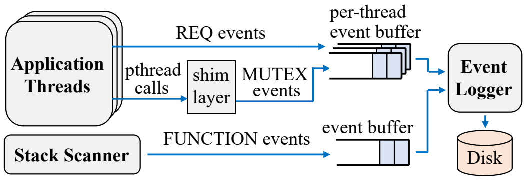


Figure 4: The flow of events that are recorded by LDB.

tion returns or a new function is called. This data race can result in the stack scanning thread collecting incorrect data (e.g., a half updated frame), which leads to inaccurate measurement of function latencies. For precise latency measurement, we need a way to detect and gracefully handle such data races, ensuring the integrity of the data collected by the stack scanning thread.

3 System Design

3.1 Overview

Our objective is to create a lightweight, portable latency profiling tool that can capture fine-grained information about the time spent in each function in a program. Thus, the per-function cost has to be minimal. We achieve this through two key ideas. First, we use a separate busy-polling core to shift away the instrumentation cost that would normally be incurred inside program threads, such as timestamping and recording events to memory. Second, we reduce the trace data generation rate by recording only functions with stack frames that are resident on the call stack for longer than the polling interval. Intuitively, very short functions do not contribute to latency, so it is okay to not spend resources in capturing them.

Building upon these ideas, we propose a new technique, called stack sampling, where a stack scanner thread repeatedly scans the call stacks of application threads. By observing the persistence of stack frames across multiple scans, the stack scanner thread can estimate each function's invocation latency. These invocation latencies can then be integrated with other event sources (e.g., acquiring a mutex, starting to process a request, spawning a thread, etc.) that are tagged with metadata and synchronized timestamps. This enables greater visibility, such as capturing locking interactions across threads.

Event recording. Figure 4 shows how different types of events are tagged and recorded by LDB. LDB has three main components that generate events. First, a *stack scanner*, which runs in a busy-polling thread, scans application threads' call stack and records invocation latencies each time a function returns. Second, a *shim layer* intercepts common threading operations (e.g., `pthread_mutex_lock()`) and records an event before forwarding the operation to its underlying implementation. Finally, application threads can generate events directly when they are annotated by the programmer, such as the start and end of a request. LDB records all events to per-thread shared-memory queues to improve scalability. An *event logger*, running in a separate thread, then gathers the events and stores them to disk for later analysis. Separately, the existing OS performance monitoring subsystem can be used to record scheduling events (not shown in the figure) like context switches and thread migrations [13]. Our design is extensible, and we plan to add additional event sources in the future, such as recording delays caused by interrupts.

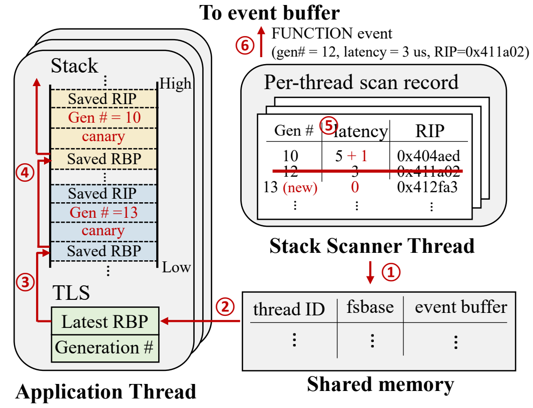


Figure 5: Stack Scanner Thread's Stack Sampling.

3.2 Stack Sampling

Compiler instrumentation. Similar to many profiling tools [18, 21, 30, 33], LDB relies on compiler instrumentation that inserts small, low-overhead changes to the function calling conventions, which enables lightweight latency profiling even without application modification. First, the compiler emits a frame pointer for each stack frame. Normally, most compilers optimize away frame pointers, but they are needed by the stack scanner to backtrace the call stack. While functions can be identified using the return address saved on the stack frame, this value doesn't allow us to differentiate between multiple invocations of the same function. This difference is critical for latency debugging, as we care about the per-invocation latency on each function, not aggregate measures like CPU time. To resolve this problem, LDB uses generation numbers to differentiate different function invocations. If a generation number is different in an otherwise identical stack frame, LDB knows that it was a separate invocation. The compiler appends a generation number to each stack frame. The generation number is a monotonically increasing number, derived by incrementing a word stored in thread-local storage (TLS). Finally, the compiler records the frame pointer of the deepest stack frame (i.e., RBP register value), also placing it in TLS. The use of TLS avoids cache contention between application threads, allowing LDB to scale well across cores.

Sampling the stack. Figure 5 illustrates how the *stack scanner* samples the call stacks of application threads. The stack scanner runs as a separate thread in the same process as the application, allowing it to share its address space. The stack scanner maintains a table of the application threads that are currently running (①). For each application thread, it fetches the frame pointer of the deepest stack frame by reading from its TLS region, (②) and starts scanning the call stack (③).

It traverses all the stack frames up to the main function by following the stack frame pointers (④). While traversing the call stack, it collects each generation number, which is located at a fixed offset from the current frame pointer, along with the return address of the stack frame.

Latency calculations. When the stack scanner collects information from the call stack, it updates its *scan records*, which are a table of metadata for each stack frame (⑤). If the scanner detects a new stack frame, it creates a new scan record and records the current timestamp and generation number. If an existing scan record's stack frame is not found during the new scan, LDB concludes that the function has returned and generates the FUNCTION event with the generation number, latency, and return address (⑥). It then removes the scan record. We now discuss the various enhancements we made to this basic procedure to address validating stack frames, avoiding race conditions, and minimizing probing effects.

Validating stack frames. Another challenge is in identifying valid stack frames. For example, even a program compiled using LDB's compiler may still be linked against a library that does not contain LDB instrumentation. Thus, some stack frames may not have valid generation numbers. To detect this, LDB reserves an eight bytes space in the stack frame, called a canary. The canary contains a known magic value that the stack scanner looks for before parsing the generation number. If it is missing, latency is not reported for that function, but any parent functions that have the canary will still be reported. To avoid pursuing invalid stack frames, LDB stops traversing when the canary in the current stack frame is invalid, or the next base pointer is invalid. Thus, LDB is guaranteed to terminate its stack traversal. Further, it avoids segmentation faults by validating whether a certain memory address is between the start of the call stack (base pointer of the very first stack frame recorded at thread start) and the end of the call stack (latest RBP value in the TLS) before reading it.

Preventing data races. The application thread could race with stack scanning if it calls or returns from a function while the stack scanner thread is traversing the call stack. To avoid collecting invalid samples, the stack scanner uses TLS data as a sequential lock, a form of optimistic concurrency control [6]. Because the frame pointer changes with each function call or return, and the generation number changes with each function call, TLS data can be used to verify that the collected generation numbers are valid. The stack scanner compares TLS data before and after each stack scan, and if they don't match, it discards the collected sample and tries again.

Reducing probing effects. Another potential concern is that reading the stack could impact an application's performance. For example, if an application thread frequently modifies a variable stored on its stack frame, and it lands in the same cache line as a stack frame, this could result in false sharing between the stack scanner and the application.

To prevent this, LDB uses TLS data to detect function calls and returns, and initiates stack sampling only when they occur. This avoids all false sharing during function execution. The stack scanner supports this by polling the most recent generation number stored in each TLS region, and waiting for it to increase before sampling the thread's call stack. The stack scanner then proceeds with the scan, retrying if there was a race condition (which is rare). Once it gets a valid sample, it stops scanning until the next time the generation number increases. The generation number is placed in a dedicated cache line, allowing it to remain in the shared cache state. Therefore, no coherence traffic is generated while it is polled (until it is modified). LDB also supports pausing the stack scanner between probes (e.g., delaying for 1 µs). However, we found that the above technique allows LDB to poll in a tight loop with negligible probing effects and better resolution.

Security. As the stack scanner shares the address space with the application thread, the application may be at threat if the stack scanner is compromised. In this paper, we assume that the LDB compiler and library are not compromised and will undergo a thorough security review.

3.3 Tracing Cross-thread Request Handling

LDB analyzes cross-thread interaction with three types of events: request events, synchronization events, and scheduling events. Each event is timestamped and included in the trace. The time duration between two events (e.g., waiting for and acquiring a lock) along with other functions that happen between the two events, help construct a rich timeline. To minimize the extra latency required for event logging, each event is recorded to a per-thread circular event buffer. Then, the events in the event buffers are polled by the event logger which persists the events to disk.

Request events. For multi-threaded applications, it is hard to figure out which threads are responsible for high tail latency. To enable per-request tracing with a multi-thread environment, LDB provides an API for developers to annotate when a thread starts and finishes handling a request. Using request annotations, LDB constructs the timeline for a specific request showing the interaction between the threads handling the same request and revealing which threads contribute to a long request processing time. In particular, all function invocations in a thread that happen between a REQ_START and a REQ_END are counted towards the timeline of the processing of that request. Request tagging is optional, but the more the application developers tag events including when a request is temporarily placed in a queue (i.e., REQ_BLOCK), the more accurate timeline LDB can construct.

Synchronization events. Contention for shared resources can be a major source of latency. Visibility into synchronization events (i.e., mutex wait, mutex acquire, and mutex release) can play a key role in identifying performance bottlenecks in the presence of cross-thread interactions. Mutex events reveal not only which mutex is contended and how long it delays a request but also which function the mutex holder thread is executing while holding the mutex. For mutex events recording, LDB interposes synchronization library functions (e.g., pthread mutex) and generates the corresponding mutex events. We discovered recording every mutex event can introduce extra overhead, especially for mutex-intensive applications. To minimize this overhead, LDB decides whether it should record mutex events outside of the critical section after releasing the mutex. If either mutex wait time or lock time exceeds MUTEX_EVENT_THRESH (1 µs by default), it records the mutex event in the event buffer.

Scheduling events. The operating system scheduler can contribute to request latency through context switching between applications or threads. Revealing the delay caused by context switches can guide the developers to look at operating system configurations, not the application, to improve latency. Unlike other types of events, LDB collects the scheduling events from an external source. In particular, we collect scheduling information with `perf-sched` for Linux. LDB timestamps the events with the same clock source as the external tool to generate a unified timeline.

3.4 Analysis Script

LDB provides an analysis script to generate per-function statistics for collected latency samples. Further, it provides another analysis script that constructs a timeline of specific requests with function names and line numbers if request tags are given. It stitches together the events generated by application threads (i.e., request and synchronization events), the stack scanner, and the OS scheduler (i.e., scheduling events).

Constructing the timeline for a specific request, identified by its request ID, requires stitching together all events that occurred during the processing of that request. Such a timeline can have multiple components, requiring the script to make multiple passes over the data generated by the profiler. First, the script looks through the event log until it observes the REQ_START event with the request ID, indicating the arrival of that request. The script tracks all FUNCTION events generated by the thread processing that request thereafter. Upon reaching a MUTEX_WAIT event, if the thread experiences non-negligible wait time (e.g., longer than 1 µs between the MUTEX_WAIT and MUTEX_LOCK events), the script scans the event log backward to identify the mutex holder thread by searching for a MUTEX_LOCK event with the same mutex. Once the thread holding the mutex is identified, the script logs all FUNCTION events produced by that thread until it releases the mutex. Then, the script continues logging FUNCTION events by the original thread processing the request until it finds a REQ_BLOCK, REQ_END, or REQ_END_ALL event. The output of the script is a log of all events impacting the processing time of the request, each event identified by (event info, thread ID, start time, end time) tuple. Such information can be easily visualized as shown in Figure 6.

4 Implementation

We implemented a prototype of LDB for the x86 architecture and the Linux environment. Our implementation has three components: 1) an extension to LLVM [32], called LLVM-LDB, to instrument stack frames, 2) a stack scanner library to poll the generation numbers and calculate latency values, and 3) an API and bindings to `pthread` library to capture request and synchronization events automatically. Our implementation integrates with the Linux performance monitoring subsystem (`perf-sched`) to track context switches [13]. We also developed scripts to parse and analyze the data recorded by our tool. The core LDB tool is ≈900 lines of C code, the scripts are ≈1,200 lines of Python code, and the changes made to LLVM are ≈250 lines of C++ code. In this section, we describe more implementation-specific details for LDB.

4.1 LLVM-LDB

Reserving TLS. We reserve two 8 B TLS variables (generation number and the latest RBP register value) at a fixed offset from the TLS base address (FS base) with LLVM `ModulePass`. We make sure that LDB TLS variables are inserted into the TBSS section after all the in-application TLS variables are inserted so that LDB TLS variables are located at a fixed offset from the FS base.

Stack frame instrumentation. We modified the sequence of the function prologue and epilogue in the LLVM x86 backend. In the function prologue, we reserve the 16 B space in a stack frame by decrementing the RSP before the RBP is pushed into the stack frame. After the RBP is updated to the current RSP, we fill up the reserved stack space and update the TLS variables. First, we increment the generation number in TLS and copy it into the reserved space. Then, we set up the canary. Finally, now that the stack frame is ready to be scanned, we update the latest RBP in TLS to the current RBP register value so that the stack scanner can start scanning from a newly created stack frame.

In the function epilogue, we revert the instrumented operations in the prologue. First, before tearing down the stack frame in the function epilogue, the compiler first updates the latest RBP in TLS to avoid the current stack frame being scanned while it is being destroyed by copying the saved RBP in the current stack frame—which holds the RBP of the parent stack frame—into the TLS region. After the saved RBP is popped from the stack frame with a standard epilogue sequence, RSP is incremented by 16 to destroy the reserved space for LDB. In total, we add 9 instrumentation instructions (7 in the prologue and 2 in the epilogue) which add less than 1 ns to each function call.

Thread instrumentation. We instrument the main function to initialize LDB using LLVM `ModulePass`. The initialization allocates the shared memory and per-thread event buffer before registering the main thread into the shared memory with its thread ID, FS base address, and event buffer address. Then, LDB launches the stack scanner thread and the logger thread. To initialize a newly launched thread and clean up the state before it exits, we interpose on `pthread_create()`. Before a newly created thread executes its original start routine, LDB allocates the per-thread event buffer and registers the thread into the shared memory. After the original thread start routine returns, it frees the event buffer and deregisters from the shared memory so that the exited thread is no longer scanned by the stack scanner.

4.2 The LDB API and Parameters

Request tagging API. LDB provides a way to annotate the threads with the following C APIs:

```
void ldb_req_start(uint64_t req_id, void *queue=NULL);
void ldb_req_block(uint64_t req_id, void *queue);
void ldb_req_end(uint64_t req_id);
void ldb_req_end_all();
```

When a thread starts to handle a request, the thread can be annotated with the request ID using `ldb_req_start()`. Optionally, if a request is dequeued from a software queue, the queue address can be specified. Multiple threads can be annotated with the same request ID with parallel processing, and a single thread can be annotated with multiple request IDs for batch processing. When a thread needs to enqueue a partially executed request into the queue the thread can hand off the responsibility of the request with `ldb_req_block()`. It indicates that the current thread is not responsible for the request anymore, but the current thread or another thread will resume processing the request later. If a thread finishes processing a request, it can clear the annotated request ID with `ldb_req_end()`. Alternatively, when a thread needs to clear all the annotated request IDs to the current thread, it can use `ldb_req_end_all()`. We decided to allow the programmer to specify the request ID, so that it can be correlated at the RPC level in coordination with other tools.

5 LDB Use Cases

To demonstrate the broad utility of LDB, we illustrates four use scenarios: visualizing a timeline of a specific request, debugging tail latency, debugging throughput, and studying the latency of specific functions. We evaluate these use scenarios with two latency-sensitive applications and one throughput-oriented application:

1. Memcached is a multithreaded, latency-sensitive, in-memory key-value store. We debug two different workloads: SET and GET. The SET workload exposes mutex and memory-intensive code paths. Each SET request can access a global lock, `slabs_lock`, multiple times to allocate and free the memory and a hash table bucket lock, `items_lock`, to update the hash table. Additionally, when a Memcached memory is saturated, it may need to acquire `lru_lock` to evict stale items. On the other hand, a GET request only needs to acquire the `items_lock` before fetching a value from the hash table.

   We allocated 10GB of memory for Memcached and used 100 million keys, evenly distributing them across requests. The value lengths are uniformly distributed between 4B and 1024B. We use the default hash power, which automatically grows based on the number of key-value pairs inserted into the hash table.

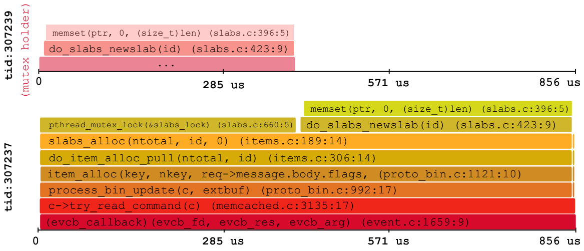


Figure 6: Timeline of the request of the longest request processing time in Memcached SET workload constructed by LDB.

2. Lucene is a multithreaded, latency-sensitive in-memory search engine library [3]. Lucene's processing time is much longer than Memcached, helping us demonstrate the value of LDB under a variety of conditions. We used a dataset of 403,619 COVID-related tweets. Each client generates single-term search queries based on the word distribution in the dataset. For each search request, Lucene first retrieves the list of document IDs from the `Segments` data structure that maps a word to a list of document IDs. Once the list of all relevant document IDs is retrieved, it fetches the pre-computed score (relevance between the document and the search query) for each document and returns the top 100 documents with high scores. All shared data structures are protected by a mutex, so multiple mutexes are acquired to serve each request.
3. Qperf [5] is a performance measurement tool for Quicly, Fastly's implementation of the QUIC protocol [31]. Unlike the other two applications, it measures the highest possible throughput between a server and a client. To achieve the highest throughput, we modified the original Qperf implementation to busy-poll for incoming packets. This application helps showcase the value of LDB when measuring the average per-packet latency for each function, by identifying functions that harm the average throughput of the application. We use Reno as the congestion control algorithm and enable generic segment offload (GSO).

To use LDB on the applications listed above, we compiled the applications with `LLVM-LDB`. In addition, we inserted tagging annotations at each code location where a thread starts to handle a new request (Memcached and Lucene) or a new packet (Qperf). These points were easy to identify, and only required 2–4 LOC changes.

5.1 Reconstructing the Timeline of the Request

When the application tags each request with a unique request ID, LDB can construct a timeline of any tagged request including interactions with other threads, which has been expensive with existing tools. Figure 6 shows an example timeline of a request in the Memcached SET workload. We picked the request with the longest request processing time we observed during the initial slab allocation phase because it provides a simple yet strong example of the value of LDB.

The detailed request timeline of LDB immediately shows what slowed down the processing of the request, including

Function	p50	p99	p9999 ▽
slabs_alloc() ↳ pthread_mutex_lock(&slabs_lock)	< 1	13.57	22.00
do_item_unlink_nolock() ↳ STATS_LOCK()	< 1	6.93	20.51
lru_pull_tail() ↳ pthread_mutex_lock(&lru_locks[])	2.14	17.53	19.62
do_item_link() ↳ STATS_LOCK()	< 1	14.64	19.50
item_unlink_q() ↳ pthread_mutex_lock(&lru_locks[])	2.03	10.56	18.88

Figure 7: Latency statistics of top 5 functions (and its caller) ranked by 99.99th percentile latency in Memcached SET workload. All numbers are in µs.

Function	p50	p99	p9999 ▽
resp_finish() ↳ THR_STATS_LOCK()	< 1	9.04	18.55
transmit() ↳ THR_STATS_LOCK()	< 1	9.66	18.26
do_item_get() ↳ assoc_find()	< 1	10.10	18.25
item_lock() ↳ mutex_lock(&item_locks[])	< 1	10.12	17.19
resp_start() ↳ memset()	1.00	10.28	17.00

Figure 8: Latency statistics of top 5 functions (and its caller) ranked by 99.99th percentile latency in Memcached GET workload. All numbers are in µs.

Function	p50	p99	p9999 ▽
IndexSearcher::search() ↳ Scorer::score()	72.18	2,005	6,232
Norm::bytes() ↳ IndexInput::readBytes()	657.7	1,690	4,899
boost::make_shared() ↳ new()	30.59	61.60	78.61
SegmentReader::docFreq() ↳ TermInfosReader::get()	< 1	57.05	61.59
TopDocsCollector::topDocs() ↳ populateResults()	< 1	20.27	25.09

Figure 9: Latency statistics of top 5 functions (and its caller) ranked by 99.99th percentile latency in Lucene workload. All numbers are in µs.

interactions with another request. When the request processing thread starts to handle the request, it waits for the slabs_lock mutex. LDB does not only tell the waiting time for the slabs_lock but also helps identify the thread holding the mutex and the function it's executing. In this example, the thread holding the mutex executes memset() while holding slabs_lock. After the thread processing the request acquires the lock, the dominant request processing time is spent executing memset() that took 645.7 µs. Such fine-grained tracing helps identify the main culprit which is performing memset() while holding the slabs_lock. This appears naturally in an LDB trace but is nearly impossible to identify using any existing tool without considerable manual work.

5.2 Tail Latency Debugging

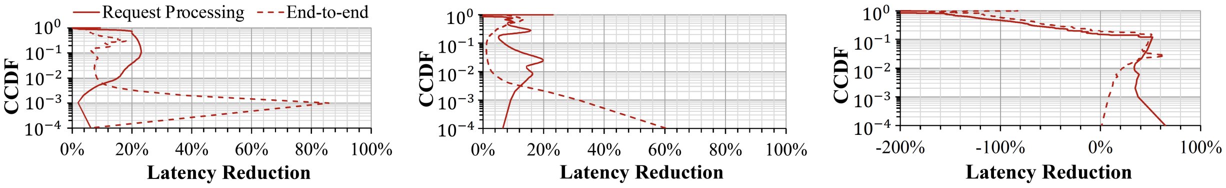
With Memcached and Lucene, we demonstrate that LDB can list functions that contribute to high tail latency, giving an insight as to how to improve their tail latencies. Note that LDB can generate latency statistics per line of code (Figure 7-9, 11) without request annotation.

For Memcached SET workload, Figure 7 lists the top five tail-contributing functions of Memcached ranked by 99.99th percentile latency. All five functions perform locking. Three out of the five functions contend for global locks related to memory management (slabs_lock) and statistics collection (stats_lock). To fix the tail latency from the slabs_lock, one could consider reducing contention by using a per-thread cache [19] and by zeroing memory without holding the lock. The stats_lock, on the other hand, could be fixed by either not using a lock, which would reduce accuracy, or by maintaining per-thread stats. Finally, the other two functions use the per-slab class lock, which is required for updating the LRU timestamp and evicting stale key-value pairs. To reduce the latency, one could fine-tune the chunk size growth factor (-f) based on the value length distribution.

For Memcached GET workload, Figure 8 shows that two of the top five functions are from per-worker thread locks (THR_STATS_LOCK). In Memcached, each network connection is assigned to one of the worker threads, but the requests can be processed by any worker thread. While the worker thread processes the request, it needs to acquire the lock of the worker thread that owns the network connection to update the statistics counters. When there is a small number of connections compared to the number of worker threads, or when the load is skewed to a subset of network connections, the per-worker thread statistics lock can be congested. The solutions mentioned above for stats_lock apply here too. Another two of the top five functions are for the hash table data structure (assoc_find() and per-bucket lock, item_locks). When a hash collision happens in the hash table, assoc_find() iterates over the bucket to find the item with the same key while holding the item_lock. One should consider initializing the Memcached with higher hashpower.

The last one is for memory operation to clear the allocated memory for a response. Considering that a response buffer will be overwritten with response data, one could consider removing the memset() operation, but care must be taken to avoid sending uninitialized data.

For Lucene workload, Figure 9 reports that the top two functions dominate the tail request processing time. Once Lucene receives a search query, it first fetches a list of document IDs by binary searching Segments after reading Segments in IndexInput::readBytes(). Once it has the list of document IDs, it looks up the score (the relevance between the query and the document) for each document and enqueues the document ID with its score into the max heap tree in Scorer::score(). In this case, a tail latency problem arises because the most popular term in the dataset appears in 88,558 documents. Thus, Scorer::score() needs to iterate 88,558 times to look up the score and enqueue it into the max heap tree, which can take 6.2 ms. To reduce this latency, one could consider utilizing an increased level of parallelism [27]. That is, if the length of fetched document ID is too long, the search



(a) Memcached SET

(b) Memcached GET

(c) Lucene

Figure 10: Latency reduction in request processing and end-to-end after applying patches that fix latency problems identified by LDB.

Function	Avg. Latency ▽
`send_pending()` ↳ `send_dgrams()`	29.63 µs
`allocate_ack_eliciting_frame()` ↳ `do_allocate_frame()`	2.79 µs
`encrypt_packet()` ↳ `ptls_aead__do_encrypt()`	2.41 µs

Figure 11: Top 3 functions (and its caller) ranked by the highest average latency in Qperf workload. The average processing time for 32 packets is 38.11 µs.

application could use multiple threads where each thread fetches the score of a subset of document IDs.

The other three functions are less significant. The memory allocation for reading the `Segments` with `new()` takes up to 79 µs, fetching the score of a document with `get()` takes up to 62 µs, and popping the top 100 documents from the max heap tree in `populateResults()` takes up to 25 µs.

Actionable Insights. To demonstrate that LDB provides actionable insights that developers can use to improve the latency behavior of real applications, we patch Memcached and Lucene using the output of LDB. We show both the request processing time, revealing the improvement to just the part of the application that LDB can profile, and the end-to-end processing time, which includes other sources of tail latency like the kernel network stack and the network fabric.

We patched Memcached to (1) preallocate the slabs to avoid memory allocation while serving the request, (2) fine-tune the object size of each slab to avoid contention in slab classes by specifying minimum object size and adjusting chunk size growth factor, and (3) convert global and per-connection stats into per-thread stats. Figure 10 (a) and (b) show the improvement in the request processing time and end-to-end latency after applying the patch. Because multiple responses can be batched before written to the wire, the improvement of end-to-end latencies is larger than the request processing times at some tail percentiles. The patch reduces the 99th percentile request processing time by 15% and 99th percentile end-to-end latency by 8% for SET workload; 99th percentile request processing time by 16% and 99th percentile end-to-end latency by 3% for GET workload.

For Lucene, we patched it to add inter-request parallelism, using four concurrent threads to serve each request. Figure 10 (c) shows the improvement in request processing time and end-to-end latency with the patch applied. The increased parallelism hurts performance for short requests due to the straggler effect, synchronization, and scheduling overhead, but achieves our overall goal of reducing request latency in the tail. This problem has been studied extensively in prior work, which suggests a more sophisticated approach would be to dynamically adjust parallelism based on the number of instantaneous requests in the system and the execution time [27]. The patch reduces 99th percentile request processing time by 34% and 99th percentile end-to-end latency by 13%.

Function	CPU Time ▽
`__libc_recvfrom`	7.61%
`send_pending`	4.11%
`quicly_send`	2.56%
... (39 more functions) ...	
`do_allocate_frame`	0.28%
... (10 more functions) ...	
`ptls_aead__do_encrypt`	0.20%
... (152 more functions) ...	
`send_dgrams`	0.04%

Figure 12: Top 5 functions ranked by the highest CPU time in Qperf workload reported by Linux perf.

5.3 Debugging Throughput of Qperf

We use LDB to debug the average performance of Qperf, demonstrating its value beyond tail-latency debugging. In particular, we profile the egress path on a Qperf server, focusing on the average per-packet latency, allowing us to determine an upper bound on achievable throughput. We find that each batch of 32 1500-byte packets takes 38.11 µs on average, putting a cap on throughput at around 9.8 Gbps. Note that actual throughput has to be lower because not all batches nor packets are maximum sized. Further, the server performs other functions beyond continuously transmitting data packets (e.g., process and transmit acknowledgments).

We use LDB to identify which functions take the most time on average for transmission handling, revealing throughput bottlenecks. Figure 11 shows the top three functions with the highest average latency. The biggest bottleneck, responsible for 77.7% of the processing time of a batch, is `send_dgrams()` which transmits packets through the kernel's `sendmsg()`, showing that the biggest performance bottleneck lies in the kernel. Other bottlenecks include memory allocation in (`do_allocate_frame()`) and encryption (`encrypt_packet()`). The remaining processing time for a batch of packets can be attributed to a collection of lower-latency functions. Thus, to improve the throughput, one should optimize the network stack (e.g., by using kernel-bypass), memory operations, and cryptographic operations.

To highlight the value of the profile produced by LDB,

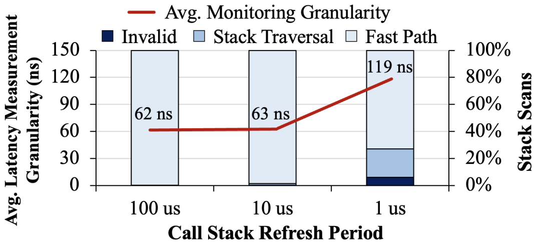


Figure 13: Average latency measurement granularity and the breakdown of stack scanning iterations with different call stack refresh periods in the synthetic application.

we compare its output to the profile produced by Linux's `perf`. Figure 12 reports the list of function names ranked by highest CPU time by `perf`. It shows that `perf` cannot pinpoint any of `send_dgrams()`, `ptls_adad__do_encrypt()`, or `do_allocate` functions that are responsible for 91% of the packet processing time, reporting that they consume 0.04%, 0.28%, and 0.2% of the CPU time, respectively. In particular, `perf`'s focus on average CPU time provides very coarse grain results, focusing on top-level functions like `quicly_send` which encapsulate all egress path functionality. Furthermore, it doesn't differentiate between functions on the critical path of egress traffic, and those happening periodically off the critical path, and it can't tie kernel delays to functions. Thus, we conclude that LDB can provide superior insights even when average performance is the focus of the debugging process.

6 Performance Evaluation

Our evaluations answer the following key questions:

1. What is LDB's latency measurement granularity?
2. Is LDB more portable than hardware-assisted latency debugging systems?
3. Can LDB limit the overhead it places on applications?
4. Can the trace data from LDB be decoded quickly?
5. How much does each component contribute to overhead?

Testbed. We use two machines with eighteen-core Intel Xeon Gold 6534 3.0GHz CPU (Ice Lake), 64GB RAM, and Mellanox ConnectX-6 200GbE NIC. For the portability experiment(§ 6.2), we compare its performance with Intel Broadwell machines (Intel Xeon E5 2640 v4 2.4GHz CPU, 64GB RAM, and Mellanox ConnectX-4 25GbE NIC) and AMD Zen3 Milan machines (AMD 7543 2.8GHz CPU, 256GB RAM, and Mellanox ConnectX-5 25GbE NIC). The median network RTT between two machines measured with ICMP packets is 30 µs. We use one machine as a server and the other as a client. Memcached and Lucene clients generate the requests following an open-loop Poisson arrival process, and Qperf clients generate a stream of requests for a data packet to measure the network bandwidth with TCP Reno as transport.

Applications. We use a synthetic application described in §6.1 for microbenchmark. To evaluate the performance of LDB, we reuse the workloads used in §5; Memcached SET/GET and Lucene are latency-sensitive workloads, and Qperf is a throughput-oriented workload.

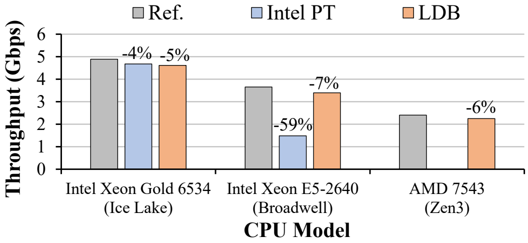


Figure 14: Average throughput of reference (without any profiling), Intel PT, and LDB with Qperf workload with different CPU architectures.

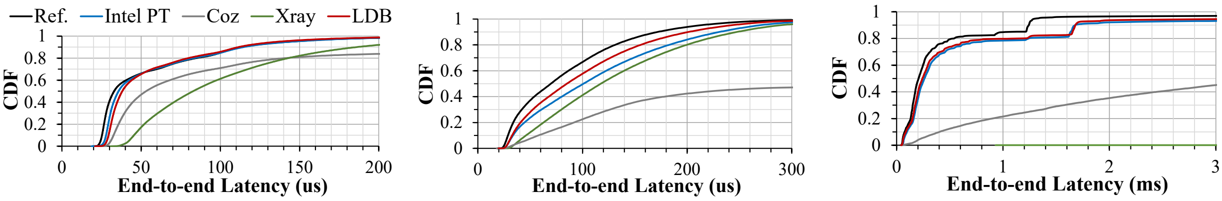
Baseline. We compare LDB to Intel Processor Trace (Intel PT) which backs state-of-the-art latency profilers [2, 4, 26], Coz that profiles the causal relationship between the function speedup and program speedup, and Xray that profiles the application's latency behavior with static timestamping. For Intel PT, we use `perf-intel-pt` provided by Linux to record and decode the Intel PT packets. For a fair comparison, we use a coarse-grained timing packet with `tsc` and decode only function call and return events with command line argument `--call-ret-trace`. We disable return compression (`noretcomp`) for more reliable decoding.

Evaluation Metrics. We report end-to-end latency (for latency-sensitive applications), average throughput (for best-effort application), raw trace size, and decoding time. End-to-end latencies and the average throughput are measured at the clients, and raw trace size and decoding time are measured at the server after the experiment finishes. Raw trace size measures the output size of each system, and decoding time measures the time required to parse the raw output to function-level latencies and to calculate the statistics of the function latencies. Because Intel PT takes too much time to decode, we measure the latency for decoding 1 ms long Intel PT trace. For LDB, we run the experiments for 4 seconds for Memcached and Qperf, and 1 minute for Lucene.

6.1 Microbenchmark

We delve into a detailed analysis of LDB's latency measurement granularity using a synthetic application which repeatedly destroys and reconstructs 20 stack frames through recursion, with a predefined refresh period. We experiment with varying the call stack refresh period from 100 μs to 1 μs and measure the average latency measurement granularity, defined as the average time elapsed between two successive valid stack scans. We further categorize the stack scanning iterations into three groups: invalid scans resulting from sequential lock fails with data races, stack traversals, and fast path iterations where no modification is detected in either the most recent RBP or the generation number in the TLS region.

Figure 13 presents the results. As the call stack refresh period decreases, the application thread interacts with the stack frames more frequently to destroy existing stack frames with function returns and to build new ones with new function calls. This increased frequency leads to cache thrashing and



(a) Memcached SET (b) Memcached GET (c) Lucene

Figure 15: End-to-end latency distribution of reference (without any profiling), Intel PT, and LDB with Memcached SET, GET, and Lucene workload at 20% load.

Workload		Trace Size / s (trace errors / s)	Decoding Time / s
Memcached SET	Intel PT	696.84 MB (2k trace errors)	48.4 min
	LDB	149.38 MB (-79%)	1.7 s
Memcached GET	Intel PT	796.72 MB (5k trace errors)	1.8 hr
	LDB	237.46 MB (-70%)	2.7 s
Lucene	Intel PT	1,066.29 MB (6k trace errors)	3.1 min
	LDB	2.12 MB (-99%)	0.7 s
Qperf	Intel PT	944.03 MB (559k trace errors)	3.7 hr
	LDB	25.4 MB (-97%)	0.8 s

Figure 16: Trace size and decoding time of Intel PT and LDB for four workloads. Trace size and decoding time are normalized by execution time.

data races between the application thread and the stack scanning thread more often, increasing the average granularity in latency measurement with more invalid stack scans. In addition, with more frequent modifications in the stack frames, the stack scanner requires more iterations of full stack frame traversals, which further increases the average latency measurement granularity. In an experiment with a function depth of 20 and 1 μs call stack refresh period, the latency can be measured with the granularity of 119 ns with 6% invalid stack scans, 21% of full scans, and 73% of fast path scans.

When dealing with multiple application threads, the average granularity increases in proportion to the number of application threads being profiled. For finer granularity in latency measurement, multiple stack scanner threads can be used, each profiling a subset of the application threads.

6.2 Portability of LDB

LDB is not designed for a specific platform. In principle, its design can be used on most architectures such as x86, ARM, and RISC-V. However, Intel-PT-based tools are tied to Intel's specific architectures and cannot be ported to other platforms. Our LDB prototype is implemented for x86 architectures and works well on any x86 architectures while Intel PT only works with some Intel processors (later than Broadwell).

To illustrate the portability of LDB, we run the Qperf workload with different x86 CPU models and compare it against the reference (i.e., no latency profiling) and Intel PT. Figure 14 shows the average throughput measured by Qperf on three different CPU architectures. It shows that Intel PT's performance highly depends on the CPU architectures. Even though Intel PT has only a 4% of throughput drop on the recent Ice Lake Intel CPU, it experiences 59% of the throughput drop on an Intel Broadwell CPU, and it cannot be used for AMD processors. On the other hand, LDB has a more consistent overhead of up to 7% thanks to its software-based approach.

6.3 Overheads of LDB

Application performance degradation. To get more confidence in LDB's low overhead, we measure the application performance impact on three latency-sensitive workloads (Memcached SET/GET and Lucene) and compare it to other profiling mechanisms. For the benefit of Intel PT, benchmarks ran on our testbed with Intel Xeon Gold 6534.

Figure 15 shows the end-to-end latency distribution measured at the client when the load is 20% of the system's capacity for Memcached and Lucene. We compare the performance of the applications when no profiling is done (i.e., Ref.) to when LDB, Intel PT, Coz, or XRay is used. For all workloads, Coz has the largest overhead at tail because it intentionally delays all the other threads than the thread being sampled, which makes it impractical to use over live traffic. The overhead of XRay is proportional to the number of function invocations as it statically instruments every function entry / exit to measure the latency. Due to its high overhead, the load exceeds the capacity, leading to extremely high latency with high queueing delay. Intel PT and LDB have comparable overhead across the workloads. LDB increases median(99th percentile) latency by 16%(1%), 22%(10%), and 18%(43%) for Memcached SET, GET, and Lucene workloads while Intel PT increases 9%(2%), 45%(23%), and 27%(64%) in the same setting.

Trace size and decoding time. Figure 16 reports the trace size and decoding time of Intel PT and LDB for the three applications. Intel PT requires high memory / PCIe bandwidth and disk space, especially for applications with more branches and jump instructions. For example, in Qperf, Intel PT outputs 944 MB/s of trace data. In addition, because of the limited memory bandwidth, it drops the event records and results in up to 559 thousand trace errors per second, which makes its visibility limited. To make matters worse, Intel PT takes up to 3.7 hours to decode 1 second of trace data, converting raw branch and jump information into function-level latencies. In contrast, the size of LDB trace is up to 99% smaller than Intel PT, typically requiring less than 250 MB/s, and it only takes a few seconds to decode 1 second of trace data.

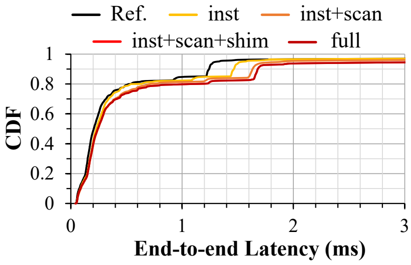


Figure 17: Performance Breakdown of LDB with Lucene workload at 20% load.

6.4 Breakdown of LDB's overhead

We analyzed how much each component contributes to the overhead for Lucene workload with the highest latency distortion under LDB whose median(99th percentile) latency is increased by 35%(69%). We gradually activated four components of LDB: application instrumentation (inst), the stack scanning/logging thread (scan), the shim layer (shim), and Linux scheduling event recording.

Figure 17 shows that instrumentation, the stack scanning, the shim layer, and Linux scheduling event recording are responsible for 11% (23%), 7% (17%), 3% (14%), and 1% (0%) of the median (99th percentile) latency increase, respectively.

7 Related Work

Sampling-based tools. Existing statistical sampling-based tools, such as perf [13] and Coz [21], are unable to analyze tail latency because they sample too slowly. Coz is designed to identify performance bottlenecks by estimating an application's virtual speedups through dynamic experiments that measure thread interactions. By artificially delaying all but one thread, Coz simulates the hypothetical performance improvement of specific functions. Although Coz addresses some of the limitations of perf, it is unsuitable for tail latency debugging because its reliance on statistical sampling causes it to miss tail behaviors. Moreover, when it identifies problematic code lines, it doesn't reveal the underlying reasons for the bottleneck, like LDB can through its request timelines.

Trace-based latency profiling tools. Trace-based tools, like XRay [18] and Intel PT, can capture the precise execution times of every function call but come with their own challenges. XRay, through compile-time instrumentation, records the execution times of individual functions. However, this results in high instrumentation overhead, which limits its use in microsecond-scale RPC applications. Intel PT, a hardware-assisted approach, captures control flow information at every branch. While systems like NSight [26] and MagicTrace [4] leverage Intel PT to debug latency problems, Intel PT's massive data rates, up to 1GB/s, necessitate substantial storage and lengthy decoding times to convert control flow information into latency information, which makes it impractical for real-time debugging. Additionally, Intel PT's overhead varies across generations of Intel CPUs, as shown in §6.2.

Continous profiling. SHIM rapidly collects hardware performance counters and software tags through busy polling [36]. It shares LDB's basic strategy of sampling with a busy-spinning core, but it lacks the ability to measure invocation latency without additional mechanisms, such as our proposed stack sampling techniques. Moreover, one busy-spinning SHIM profiler thread is needed for each hyperthread pair, resulting in high overheads due to competition over shared functional units. LDB, by contrast, can avoid this overhead through a different design that enables one monitoring thread to profile multiple threads running across multiple cores.

Limiting tracing to specific functions. As seen with the case of Xray, timestamp instrumentation at each function's entry and exit entails significant overhead. Thus, some tools limit tracing to a few specific functions at a time. There are various techniques and tools to enable dynamically enabling/disabling timestamp instrumentation: notably, dynamic instruction patching [18, 20, 25], dynamic instrumentation via eBPF [1, 10, 24], and instrumentation via JIT compiler [33]. However, because the scope of functions being profiled is limited, they require multiple iterations with the developer's intervention for latency debugging, and they sacrifice the ability to capture complete timelines. There are efforts to streamline these iterations [7, 29, 30]. AMD offers a suite of profiling tools (e.g., Omnitrace [16] and uProf [15]). Both solutions rely on sampling. Further, Omnitrace offers Coz-like functionality as well as specific function instrumentation. Omnitrace's instrumentation adds 1024 instructions per function compared to LDB's 9 instructions per function.

Distributed latency tracing. Envoy [11], Zipkin [14], Jaeger [12], AWS X-Ray [9], and Apache SkyWalking [8] provide tools to trace a request in a distributed computing environment at an RPC or microservice granularity. These tools may find a service causing high end-to-end latency, but they don't have visibility inside the service. Distributed tracing systems and LDB are complimentary. Problematic services can be found with distributed tracing, while problematic functions in a specific service can be found with LDB.

Mutex bindings. Dynamic data race detectors, like Eraser [34], often use similar mechanisms to interpose on locking functions, but their goal is to instead verify if the application follows a consistent locking protocol.

8 Conclusion

In this paper, we presented LDB, an efficient latency profiling tool with low overhead, high visibility, fast decoding, and portability. It utilizes a key technique, stack sampling, where each function's invocation latency is measured by sampling a unique generation number assigned in the stack frame. With optional request tagging by the developer, LDB can construct the detailed timeline of a request, including cross-thread interactions caused by synchronization, the time spent in functions, and the contribution of the OS scheduler. Our evaluation showed that LDB could profile the latency behavior of three applications and reveal their main performance bottlenecks effectively on multiple platforms with low overhead.

Acknowledgments

We thank our shepherd Yongle Zhang, as well as to the anonymous reviewers for their invaluable feedback. We also thank Kostis Kaffes, John Ousterhout, and David Culler for their constructive feedback; and Cloudlab [23] for providing us with machines of different architectures for portability experiments. This work was funded in part by NSF grants CNS-2104398, CNS-2212098, and CNS-2212099; DARPA FastNICs (HR0011-20-C-0089); VMware; and a Google Research Award.

References

[1] eBPF. https://ebpf.io/.

[2] Fix performance bottlenecks with intel vtune profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

[3] Lucene++: c++ port of lucene library. https://github.com/luceneplusplus/LucenePlusPlus.

[4] magic-trace: Diagnosing tricky performance issues easily with intel processor trace. https://blog.janestreet.com/magic-trace/.

[5] qperf: performance measurement tool for QUIC. https://github.com/rbruenig/qperf/.

[6] Sequence counters and sequential locks. https://docs.kernel.org/locking/seqlock.html.

[7] wachy: A new approach to performance debugging. https://rubrikinc.github.io/wachy/.

[8] Apache SkyWalking, 2022. https://skywalking.apache.org/.

[9] AWS X-Ray, 2022. https://aws.amazon.com/xray/.

[10] bpftrace: High-level tracing language for Linux systems, 2022. https://bpftrace.org/.

[11] Envoy Proxy, 2022. https://www.envoyproxy.io/.

[12] Jaeger: open source, end-to-end distributed tracing, 2022. https://www.jaegertracing.io/.

[13] perf: Linux profiling with performance counters, 2022. https://perf.wiki.kernel.org/index.php/Main_Page.

[14] Zipkin, 2022. https://zipkin.io/.

[15] AMD uProf, 2023. https://www.amd.com/en/developer/uprof.html.

[16] Omnitrace: Application profiling, tracing, and analysis, 2023. https://github.com/AMDResearch/omnitrace/.

[17] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[18] D. M. Berris, A. Veitch, N. Heintze, E. Anderson, and N. Wang. Xray: A function call tracing system. Technical report, 2016.

[19] J. Bonwick and J. Adams. Magazines and vmem: Extending the slab allocator to many cpus and arbitrary resources. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference, June 25-30, 2001, Boston, Massachusetts, USA*, pages 15–33. USENIX, 2001.

[20] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *ATC*, 2004.

[21] C. Curtsinger and E. D. Berger. Coz: Finding code that counts with causal profiling. In *SOSP*, 2015.

[22] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 2013.

[23] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, et al. The design and operation of cloudlab. In *ATC*, 2019.

[24] B. Gregg. Linux bpf superpowers. http://www.brendangregg.com/blog/2016-03-05/linux-bpf-superpowers.html, 2016.

[25] B. Gregg and J. Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*. Prentice Hall Professional, 2011.

[26] R. Haecki, R. N. Mysore, L. Suresh, G. Zellweger, B. Gan, T. Merrifield, S. Banerjee, and T. Roscoe. How to diagnose nanosecond network latencies in rich end-host stacks. In *NSDI*, 2022.

[27] M. E. Haque, Y. H. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *ASPLOS*, 2015.

[28] K. M. Hazelwood, S. Bird, D. M. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*, pages 620–629. IEEE Computer Society, 2018.

[29] J. Huang, B. Mozafari, and T. F. Wenisch. Statistical analysis of latency through semantic profiling. In *EuroSys*, 2017.

[30] M. Jovic, A. Adamoli, and M. Hauswirth. Catch me if you can: performance bug detection in the wild. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 155–170. ACM, 2011.

[31] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. In *SIGCOMM*, 2017.

[32] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[33] Y. Luo, K. Rodrigues, C. Li, F. Zhang, L. Jiang, B. Xia, D. Lion, and D. Yuan. Hubble: Performance debugging with in-production, just-in-time method tracing on android. In *OSDI*, 2022.

[34] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *TOCS*, 1997.

[35] S. Yang, S. J. Park, and J. Ousterhout. NanoLog: A nanosecond scale logging system. In *ATC*, 2018.

[36] X. Yang, S. M. Blackburn, and K. S. McKinley. Computer performance microscopy with shim. *ISCA*, 2015.