How to diagnose nanosecond network latencies in rich end-host stacks

Roni Haecki, ETH Zurich; Radhika Niranjan Mysore, Lalith Suresh, Gerd Zellweger, Bo Gan, Timothy Merrifield, and Sujata Banerjee, VMware; Timothy Roscoe, ETH Zurich

https://www.usenix.org/conference/nsdi22/presentation/haecki
How to diagnose nanosecond network latencies in rich end-host stacks

Roni Haecki\(^1\), Radhika Niranjan Mysore\(^2\), Lalith Suresh\(^2\), Gerd Zellweger\(^2\), Bo Gan\(^2\), Timothy Merrifield\(^2\), Sujata Banerjee\(^2\), Timothy Roscoe\(^1\)

\(^1\)ETH Zurich, \(^2\)VMware

Abstract

Low-latency network stacks have brought down network latencies within end-hosts to the microsecond-regime. However, end-host profilers have such high overheads that they are useful only to confirm a hypothesis, not to diagnose a problem in the first place. Indeed, every one of twenty low-latency network projects we surveyed rolled their own analysis tools instead of using an existing profiler.

This paper shows how to build a latency diagnosis tool with full-stack coverage and low overhead that can identify, not just confirm, sources of latency in end hosts. The unique measurement methodology reconstructs network-message lifetimes within end hosts with nanosecond precision, by reconciling CPU and NIC hardware profiling traces across multiple time domains (network and CPU). It uncovers unexpected latency sources in both kernel and user-space stacks. We demonstrate these capabilities by using our implementation, NSight, to systematically identify and remove performance overheads in memcached, reducing 99.9th percentile latency by a factor of 40 from 2.2 ms to 41 µs.

1 Introduction

Operating systems and network stacks are routinely blamed for increasing network latencies. Clearly, we need diagnostic tools to identify sources of latency in end-host stacks. Thankfully, there is no paucity of end-host profilers [1,3,4,6,13,16,22,25,29,31,37,38,49–51,73,77]. We examined 21 networking projects whose goal was to achieve low latency [2,5,8,11,20,23,26,27,33–36,41,46,48,55,57,58,60,65,69]. Surprisingly, not one of these projects have used these profilers! Instead, all of them design their own handcrafted latency measurement system. This indicates that in spite of the excellent and vast body of prior work, there is no diagnostic tool for network latencies introduced at the end host, especially in the microsecond regime. In this paper we present NSight to address this important gap.

Our investigations identify three reasons that existing end-host profilers fail at network latency diagnosis. First, existing profilers fail to capture latency deviations added by the NIC, from the point when messages enter (or exit) the NIC to the point that they are received by (or exit) the driver. Many system designs identify these latency deviations to be important [8,24,26,34,35,48,55,58,60,62].

Second, their high overheads severely disturb the latency distribution, overwhelming the root causes being pursued. Figure 1 shows how two widely-used profilers, eBPF and Ftrace, add to memcached request-receive latencies, measured from the point requests are received at the NIC to the point they are received by memcached (socket recv). eBPF adds 18–40% overhead while measuring the latency of a single function, while Ftrace adds 298–841% profiling all functions of the end-host software stack. Also shown is the minimal impact of NSight and in the same range as Intel-PT, which is an example of a hardware CPU profiler that NSight builds on.

Third, existing profilers are too heavyweight to apply to the entire stack. A developer diagnosing network latencies must already have a guess of where to look before such a profiler is useful. Blogs [15,28,32,43,45] tell exactly this story: Users determine the parts of the stack that might add latency deviations and then use profiling tools to measure latencies in these parts. The unfortunate result is that the latency sources from unexamined parts of the stack are not caught. In addition, important interactions between different parts of the stack go unnoticed [46,68]. For example, profiling the NIC separately from the network software stack hides the impact of NIC deviations on scheduling decisions. When there are large latency deviations at the NIC, a CPU waiting for network messages can idle and go into a lower power state. This increases scheduling latency, and in turn, overall network latency (§7.4 has an example).

![Figure 1: CDF of memcached request-receive latencies with and without profiling. eBPF-1 stands for eBPF probing a single function; Ftrace, Intel-PT and NSight, profile all system functions in the end-host stack. eBPF-1 and Ftrace add variable latencies to functions being profiled that are hard to differentiate from true latency deviations.](image-url)
Our contribution. We correct these shortcomings by demonstrating the feasibility of a low-overhead, holistic tool called NSight for diagnosing network latency deviations introduced at end hosts. The guiding principle is that entire lifetimes of network messages within end hosts must be examined to determine the precise causes of latency deviations. These lifetimes are defined by all system activity, not limited to network processing, that impacts messages from the time they enter end hosts to when they exit it. To be useful in the microsecond regime, NSight must reconstruct these lifetimes with nanosecond precision. NSight does so by reconciling timelines of two fine-grained data sources, CPU hardware profiling and NIC hardware timestamps.

This reconstruction is challenging for two reasons. First, the two data sources record time using different hardware clocks. Hardware CPU profiling uses a monotonically increasing clock for capturing precise intra-end-host latencies, while NIC and software CPU clocks are often synchronized using PTP [61] to aid inter-end-host latency measurements. To align the timestamps in these sources correctly, NSight tracks the conversion between the two time domains during profiling.

Second, CPU hardware profiling does not track the passage of network messages in multi-core systems across kernel cores (which process the message) and application cores. To track this path, NSight captures timestamps and core numbers at the boundary where kernel hands off the message to the application. This boundary is also the point where message processing can move across cores.

Once the lifetimes of messages are constructed, they can be compared to identify anomalous processing that led to their latency deviations. Unfortunately, due to the deep nesting of end-host call stacks, latency deviations in functions introduce deviations in their parent functions making them look anomalous too! To reduce ambiguity in attributing root causes to anomalies, NSight traverses the call stacks until it finds functions with latency deviations that cannot be attributed to nested calls.

NSight demonstrates that these techniques are sufficient to overcome the listed challenges, while incurring overheads comparable to hardware profiling (see Figure 1). Due to its low overhead, NSight can be used to diagnose even sub-microsecond increases in network latency at the end host. We describe our use of NSight to profile both the Linux kernel and Mellanox’s VMA [48], a user-space network stack. On these stacks we describe how we profile unmodified applications memcached and redis. We dive deeply into a detailed case study of memcached, the application of choice in several low-latency end-host stack papers [11,35,36,39,55,58,60,70,71], to prove the diagnosis capability of NSight. In the case study NSight systematically narrows down the bottlenecks of the Linux stack, as we mitigate 99.9th percentile memcached tail latency by over 40x - from 2.2 ms to 51 µs. When one of the mitigations increases latencies below the median by as much as 11 µs, NSight helps identify the reasons for this increase.

2 Background and related work

The vast body of profiling and diagnostic tools fall broadly into two groups: fault diagnosis tools ([10,64,74]) and performance diagnosis tools ([21,75,76]). Some diagnose problems within end-hosts ([1,3,4]). Others help in distributed settings ([19,21,68]). NSight is a performance diagnosis tool for end-hosts. Its focus is on latency diagnosis. We focus on software- and hardware-based tools built for latency diagnosis below.

Software-based end-host tools. Software-based diagnostic tools are built on top of profiling data sources such as probes (Uprobes [17], Kprobes [30]), kernel tracepoints, performance counters, and software tracing. The data source determines the overheads and insights that can be drawn from the tool.

Tools that depend on probes (LTTng [13], eBPF [1]) and kernel tracepoints (dtrace [51]) allow users to instrument functions of interest. Users of these tools must already know where to look, making them less suited for latency diagnosis than tools that depend on software tracing (Ptrace [3]). Tools based on performance counters (perf stat [4]) do not capture outlier latency events (due to aggregation) making them unsuited for tail latency diagnosis in particular.

Unlike NSight, all of these tools miss NIC delays altogether and have much higher overheads than hardware profilers.

Hardware-based end-host tools. CPU profilers ([12,40,53]) record software function call and return times, their core number and process names at processor speeds. The timestamps help measure software latencies with nanosecond precision and low overhead. NSight, like perf and VTune [25], uses CPU profilers to track software function latencies.

Most NICs support hardware timestamping [14] for network packets. The timestamps can be retrieved per-message using standard Linux socket calls, to determine network latencies across and within end-hosts. NSight uses NIC timestamps to track entry and exit of network messages during profiling.

Unlike NSight, these tools cannot identify the sources of network latencies in end-hosts by themselves or when combined with one another. Instead they require the user to make a conjecture that the tools can help confirm (See §2.1 and §4).

Distributed tools. Even though NSight is designed for network latency diagnosis within end-hosts rather than in distributed settings, some similarities and differences are worth noting. Like NSight, many diagnostic tools for distributed systems [7,9,18,19,44,45,54,68,72] reconstruct the path of messages in distributed settings. They do not capture network latency sources at end-hosts but capture other latency sources like packet drops, routing issues and workload spikes.

Inband network telemetry (INT) [56] is a mechanism to probe specific points in network dataplanes. In software, these probes are expensive [67] and do not cover many points in end-host stacks, like the OS stack, that we do with NSight. Therefore they can only confirm causes of latency, not identify them. On the other hand, INT works with all programmable network hardware while NSight is relevant only for end-hosts.
2.1 Diagnosing network delays at end-hosts

We now describe common network latency debugging practices at end-hosts using existing tools (blogs [15, 28, 32, 43, 45]). The process usually begins with tools that summarize system activity, like perf [4], SystemTap [59], unless developers already have deep knowledge of the application of interest [32]. These tools help identify the parts of the stack that are most active and might impact network latency.

For example, we can generate flame graphs like Figure 2 with perf. The flame graph shows two applications, memcached and NGINX both running on the same machine. It is natural to ask whether they interfere to cause network delays [20, 27, 55]. To verify, developers can isolate applications, or turn on software tracing, like Ftrace, and visualize network send/receive paths using tools like KernelShark [66].

Figure 2 also shows memcached LRU cache maintenance and paging activity to be high. Even though unrelated, these can delay network activity if scheduled during network processing. To verify, developers can use tools like eBPF or Ftrace to track periods of such activity and correlate them with periods of high network latencies. To verify delays on network processing paths, these tools can generate latency distribution graphs for the functions of interest [45].

These steps alone might be insufficient [15]. System activity summaries like Figure 2 do not show problems like NIC delays, scheduling bottlenecks, or head-of-line blocking that can slow down network processing. Finding these problems requires tracking specific system/NIC performance counters [4, 52] or handcrafted measurements [35, 39].

3 Using NSight for diagnosis

We now describe a typical debugging experience with NSight. Let us suppose users are running memcached on Linux and see large tail latencies despite light query loads. To use NSight to quickly diagnose the cause of poor performance, the user first turns on NSight profiling for a second. Once NSight analyzes the profiling data, the user inspects the initial result, Figure 3. This balloon plot shows the latencies of all memcached requests profiled and the top causes of the slowest (tail) requests. The user notices that the tail requests are largely slowed down by idling CPU cores, CPU Idle (A), where A stands for application-core on which memcached runs. The slowest request is also slowed down by core idling due to NIC interrupt coalescing delay, NIC delay (R), where R stands for the core on which the kernel receives the request.

From this result, the user can drill down into the presented causes for tail latencies. For instance, the user cross-checks why the cores were idling (X-axis), and the latency deviation due to those causes (Y-axis). Both the application (A) and receive cores (R) idle, for reasons ranging from waiting during a context switch or file access, with NIC interrupt coalescing, NIC delay (R), causing the largest wait.

Figure 3: Balloon plots summarize message latencies (Y-axis) with the top causes for latency deviations (legend) mapped to corresponding messages using similar balloons. The balloons are in Message ID order (X-axis) to identify bunching together of balloons that indicates a single underlying cause. If they are spaced apart, there are likely multiple independent reasons that need to be addressed separately.

Figure 4: A zoomed-in box plot of causes for CPU idling that impacted the slowest message. This presentation is valuable for identifying why the cores were idling (X-axis), and the latency deviation due to those causes (Y-axis). Both the application (A) and receive cores (R) idle, for reasons ranging from waiting during a context switch or file access, with NIC interrupt coalescing, NIC delay (R), causing the largest wait.

Figure 5: A detailed timeline of all system activity that impacted the slowest message. Eyeballing the presentation, the user confirms that there is a gap in system activity before the message is processed but long after the message is received at the NIC, showing that interrupt coalescing delays slowed down the tail request by as much as 180μs.
4 Challenges and key ideas

Why is latency diagnosis hard? To derive the precise causes of message slowdowns within end-hosts, we must profile network-message lifetimes, like Figure 5. These lifetimes capture all system activity (G1) that slowdown message processing, whether it is network stack activity or something unrelated, from the point a network message enters (or exits) the end-host to the point it is received (or sent) by the application. To work with low-latency stacks, we must capture these lifetimes automatically (G2), with nanosecond-precision (G3) and low overhead (G4). We must then analyze these message lifetimes to identify system activity that slows messages down and their precise impact, as shown in Figure 3.

4.1 Profiling network-message lifetimes

Unfortunately, the task of capturing network-message lifetimes is difficult, because there is not one system component that processes network messages. Rather, in most modern OSes [42, 47, 55], network messages are processed across the NIC and one or multiple CPU cores. Capturing message lifetimes that span these devices is challenging for two reasons.

[Challenge 1 (C1):] NIC, CPU profiling, and software timestamps come from independently changing clocks. Figure 6 illustrates this problem. To construct message lifetimes NIC, CPU profiling, and software clocks must align at all times, but they do not. NIC and CPU profiling clocks drift independently of each other. This is not a problem for CPU profiling, because it uses the hardware clock to report nanosecond-granularity function latencies within an end-host.

NIC timestamps are used to measure both inter-host and intra-host message latencies. To aid latency measurement across multiple time-domains, socket libraries convert NIC timestamps to software timestamps during profiling, with the help of conversion parameters calculated by software synchronization mechanisms like phc2sys on Linux.

Software clocks across end-hosts are usually synchronized using protocols like ptp. These synchronization mechanisms constantly readjust software clocks and change their relationship with the CPU profiling clock, under the hood. These changes pose a problem for constructing message lifetimes.

Key idea 1, Time reconciliation: The ideal solution to this problem would be to modify time synchronization protocols to expose clock changes to profilers like perf. Profilers could then present all measurements based on the software clock to simplify system-wide latency measurements.

We do not want to refactor the entire software stack, so we use a simpler but more expensive workaround. After time synchronization protocols change the software clock, the kernel recalculates the conversion between CPU profiling clock and software clock. By exposing these conversion parameters to user-space using virtual dynamic shared object (vDSO) mechanisms, we can poll them from a user-thread for the duration of profiling. The goal is to capture every change to the software clock to have the most accurate mapping between the clocks at all points in time. A log of the conversion parameters helps reconcile CPU profile timestamps to software-clock domain post profiling. This method burns an entire core for the user-thread, but it adds no latency overhead during profiling because the user thread is isolated from the rest of the system.

[Challenge 2 (C2):] There is no support to track network-message lifetimes within host software stacks. Software profiling tools can only track network messages within the network stack, missing other system activity that slows down message processing. This is because CPU architecture and operating systems treat network messages like regular data structures, as objects in memory. Consequently, network messages have no relevance outside of network stack functions.

On the other hand, CPU profiling tracks all system activity but not the message lifetimes within which such system activity occurs. Figure 7 illustrates this problem. The two rectangular boxes labeled Core X and Core Y show the timeline of system activity captured by CPU profiling at these two
cores. We only see the process contexts (kernel, application) and individual functions (not shown), but not messages.  

**Key idea 2. Message profiling:** To construct message lifetimes, we augment CPU profiling with per-message timestamps and core numbers. Figure 7 shows the information collected by NSight profiling to construct two message lifetimes, $M^1$ and $M^2$. For each message (for example, $M^2$), we seek to get three timestamps: a NIC timestamp ($M^2(n)$), a core hand-off timestamp ($M^2(h)$), and an application timestamp ($M^2(a)$). The NIC and application timestamps allow us to capture the start and end of a message’s lifetime on the end-host; for instance, $M^2$’s lifetime is from $M^2(n)$ to $M^2(a)$. The core hand-off timestamp ($M^2(h)$) along with core information (Core $X$→$Y$), helps identify the per-core system activity that a message encounters in its lifetime; for example, $M^2$ is processed on Core $X$ between $M^2(n)$ to $M^2(h)$, and then on Core $Y$ between $M^2(h)$ to $M^2(a)$. Crucially, these timestamps are sufficient to produce a single timeline of all system activity related to the processing of a message.

We now explain how these timestamps can be obtained and how they suffice to reconstruct detailed message lifetimes. The per-message NIC hardware-timestamps and application software-timestamps can be taken at the application send/receive operations. They help order messages, so we can assign message IDs, $M^1$ and $M^2$. Messages sent from end-hosts are ordered by their application timestamps. Messages received at end-hosts, like $M^1$ and $M^2$, are ordered by NIC timestamps, $M^1(n)$ and $M^2(n)$. The per-message hand-off timestamps and core information can be collected at points where messages cross software-processing and core boundaries. In kernel network stacks, there is one boundary where messages are handed-off between kernel and application, like `sock_def_readable` in Linux. User-space network stacks can have similar boundaries [47,55] while others do not [26,48,58].

In our example in Figure 7, there is an overlap between lifetimes of $M^1$ and $M^2$ on core $Y$ from $M^2(h)$ to $M^1(a)$. $M^2$ has to wait for the application on core $Y$ to complete processing $M^1$ before it can be processed. This overlap shows inter-message interference or head-of-line blocking. We can similarly detect unrelated system activity or application interference that appear in message lifetimes.

Figure 8 describes NSight’s presentation of $M^2$. Between timestamps $M^2(n)$ and $M^2(h), M^2$ is processed by core $X$ (Receive core). After $M^2(h), M^2$ is processed by core $Y$ (App core) where $M^2$ waits for the application, memcached, to process $M^1$ (App Recv 1 to App Send), after which $M^2$ is received (App Recv 2).

4.2 Diagnosing high message latencies

Profiling even for brief periods of time, will leave us with hundreds of thousands of message lifetimes. To diagnose why some message lifetimes are longer, like we did in Figure 3, we must identify anomalous system activity that slow down those
messages relative to others, by comparing their lifetimes. Two typical types of anomalies in message lifetimes are functions that take longer and functions that occur more frequently in some lifetimes compared to others; for example, App Recv occurs twice in $M_2$ in Figure 8. Deeply nested end-host stacks can result in nesting of anomalies of different types, leading to a third challenge.

**Challenge 3 (C3):** Due to nesting, the same latency deviation can be explained by multiple anomalies. Figure 9 explains this problem. When compared to $M_1$, most functions in $M_2$ appear anomalous! Some take longer, like the first invocation of event_handler; others appear more often, like __libc_recvmsg; and the rest are unexpected, like __libc_send, a send in the middle of receiving $M_2$.

**Key idea 3, Anomaly disambiguation:** To reduce ambiguities from nesting, we only report anomalous functions that cannot already be explained by their nested functions. To determine whether an anomalous function is explained by nested anomalies, we use a heuristic. If the nested anomalies together account for more than 80% of the latency deviation of the parent function (see §5.2 for why), we conclude that the nested anomalies explain the parent anomaly and omit the parent anomaly as a reason for deviation.

Figure 9 describes how the latency deviation in call_function_single_interrupt is accounted for by a nested anomalous ip_rcv call. Therefore, ip_rcv is listed as a root cause but not call_function_single_interrupt.

5 Design and implementation

In this section, we describe how the three key ideas from §4 realize the goal of network latency diagnosis in microsecond-regimes. NSight is composed of two subsystems. The first is a profiling subsystem, that tracks broad CPU activity and passage of messages through end-host stack (§4.1, Message profiling). It also tracks the relationship between NIC, CPU profiling and software time-domains so that the observations from different devices can be aligned into a single timeline (§4.1, Time reconciliation). Put together, it is responsible for capturing all system activity (G1) during the profiling period with nanosecond-precision (G3) and low-overhead (G4). The second is an analysis subsystem that reconstructs network-message lifetimes and diagnoses network latency deviations within them. To do so, it analyses anomalous system activity in network-message lifetimes, identifies root causes, and attributes precise deviations to these root causes (§4.2, Anomaly disambiguation). The profiling and analysis subsystems together automate (G2) network latency diagnosis.

5.1 NSight profiler

Figure 10 explains how we get broad information from CPU hardware profiling (G1) and combine it with per-message profiling information from a shim layer to establish a causal link between system behavior and message lifetimes (§4.1, Message profiling). Typical CPU profiles collect system activity at function granularity and list all function call and return times, their execution context and core numbers with nanosecond granularity (G3). Independent of the CPU profiler, the shim layer intercepts messages, irrespective of the application, to collect timestamps at three points in the stack, shown
in Figure 10, as metadata for each message. Core numbers are collected at the kernel–userspace boundary. For socket-based stacks, the shim layer also collects socket file descriptor numbers to detect head-of-line blocking that arises when application threads are multiplexed across multiple sockets. We quantify overheads from profiling in Section 6 (G4).

To reconcile independent observations from the CPU profiler and the shim layer into a single timeline, §4.1, *Time reconciliation* polls the conversion between the clocks for the duration of profiling from a user-thread. To construct message lifetimes accurately, the user-thread must detect all conversion changes and align the observations at all points in time. We quantify the accuracy of this scheme in §6 (G3).

**Profiler implementation and deployment.** The shim layer is implemented using standard Linux and socket APIs, with 1055 lines of C code. It can be dynamically linked by unmodified applications using LD_PRELOAD; statically linked applications might need to be modified, however.

We extend the Linux NIC timestamp framework to obtain timestamps at the kernel–userspace message hand-off boundary by patching 40 lines of the 5.4 Linux kernel (§4.1, C2). VMA stack has no such boundary and needs no modification.

NSight is built on top of the Intel-PT CPU profiler, whose traces are available through perf. perf exports Intel-PT timestamps from TSC to sched_clock timedomain. We modify 443 lines in perf to use the time reconciliation parameters and export CPU profiler timestamps directly in software time domain (CLOCK_REALTIME). The thread that polls these parameters is implemented in 363 lines of C code (§4.1, C1).

§7 shows how NSight is effective even when users turn it on for only a few seconds at a time. To capture random events across time, users must turn on NSight repeatedly. We have ambitions of using NSight for continuous profiling, but the current buffering implementation in Intel-PT limits such use.

### 5.2 NSight analysis

The analysis subsystem consists of three parts. The first part reconstructs message lifetimes from profiling data (§4.1, *Message profiling*). The second part detects anomalous system activity during these lifetimes. The third part sifts through the anomalies to identify root causes for latency deviations within message lifetimes (§4.2, *Anomaly disambiguation*, G2).

**Identifying anomalous system activity.** In §4.2 we mentioned three types of anomalous system activity that contribute to latency deviations: functions that take longer, functions that are called more frequently, and unexpected functions that show up in some message lifetimes compared to others.

There are three other classes of anomalies that slow down message lifetimes. The first class consists of *entire program contexts* that are unexpected but show up in message lifetimes as a result of scheduling decisions or interrupts. The second class is defined by the *absence of system activity* that is seen when the CPU is idling. This anomaly occurs when the CPU waits for an event, like a memory read due to a cache miss, and is often indicative of resource bottlenecks. The third class is *cross-message interference* in the network stack.

The algorithm that classifies system activity within message lifetimes as anomalous or normal runs in four phases. The *first phase* compares message lifetimes belonging to the same application and identifies unexpected *program contexts* that occur in a minority of message lifetimes. OS and application interference is usually identified in this phase.

The *second phase* identifies gaps in system activity that are not associated with any function or program context, like Figure 5. It also identifies the cause for each gap from the last function call preceding the gap. Figure 4 is an example result.

The *third phase* compares message lifetimes belonging to the same application and identifies individual *functions* that are slower, more frequently called, or unexpected. Figure 8 presented several anomalous functions identified in M2 lifetime. Error handling code paths, like TCP retransmission, and application bottlenecks are usually identified in this phase.

The *final phase* identifies overlaps between messages, similar to how we detect overlap between M1 and M2 in §4.1. *Message profiling*. This phase detects cross-application network interference when the overlap is between messages belonging to different applications, and head-of-line blocking when the overlap is between messages of the same application.

**Attributing root causes to anomalies.** §4.2, C3 discussed how nesting ambiguates latency-deviation attribution. Often, only one of the anomalies in a deeply nested stack is sufficient to explain the latency deviation due to the stack and identify corrective action. In such cases, listing the nested anomalies as root causes only ambiguates diagnosis.

We now revisit this challenge in the context of an example in Figure 11 that has an anomalous program context (ksoftirqd) and cross-application message interference (ip_rcv). It is unnecessary to flag nested anomalies within anomalous program contexts or cross-application message interference because the corrective action to address the deviation is clear: the scheduling policy must be revisited to avoid these anomalies. Therefore, the algorithm will not flag queue_work_on or tcp_v4_rcv even if they are anomalous.
§4.2. Anomaly disambiguation described how we do not report anomalous parent functions if 80% percent of their latency deviation is already explained by the nested anomalies. When we report nested anomalies instead of their parent anomaly, we will not be able to explain some portion of the latency deviation. This is due to the fact that nested function latencies seldom make up 100% of the parent function latencies. We define the fraction of the latency deviation explained by NSight diagnosis as its coverage. For example, if a parent function takes 1000 ns longer than normal, and nested function takes 900 ns longer than normal, reporting only the nested function as root cause will result in 90% coverage. We evaluate NSight’s coverage in §6.

Analysis implementation. The algorithms in the analysis subsystem are implemented with 2189 lines of R code. The visualisations comprise 1768 lines of R.

The precision of Intel-PT is limited because it profiles in batches, a few CPU cycles at a time. Latencies of functions whose call and return times are within the same batch can be under-reported to take no time at all! On the other hand, small latencies, corresponding to the batch granularity, can be added between batches, giving the appearance of an idling CPU. Such under-reporting and bogus gaps in system activity obfuscates anomaly detection. NSight analysis algorithms therefore ignore differences in latency smaller than the batch granularity (up to 322 ns in our system).

Instead of directly reporting anomalous functions such as ksoftirqd or ip_recv that are hard to interpret as root causes, we categorize them by functionality and report a single root cause, like OS threads or Receive processing. Each category is suggestive of a class of corrective actions that apply to all functions in that category. Table 1 shows 4 examples of categories used in this paper.

The process of categorization is partly automated. For example, process contexts are automatically derived from perf. We categorize 1350 functions in Linux by hand, using the contextual information from their names in a user configurable CSV file. We also introduce head-of-line blocking as a category. To do so, we automate summarization of all functions that are executed when processing messages with minimum latency. When these functions occur more frequently or their latencies deviate (for example, NIC interrupt processing takes longer) in message lifetimes of the same application, NSight shows head-of-line blocking as one of the root causes. We verify the latency deviations attributed to head-of-line blocking by cross-referencing message lifetimes to identify overlaps.

6 Evaluation

Our work on NSight is motivated by the paucity of full-stack, lightweight and high-fidelity network latency diagnosis tools that can be used to diagnose latencies in the microsecond regime. In this section, we examine to what extent the ideas in this paper address this gap.

<table>
<thead>
<tr>
<th>Category</th>
<th>Anomalies</th>
</tr>
</thead>
<tbody>
<tr>
<td>NGINX</td>
<td>NGINX process context</td>
</tr>
<tr>
<td>head-of-line blocking</td>
<td>Functions that were involved in processing the fastest network messages.</td>
</tr>
<tr>
<td>OS Threads</td>
<td>ksoftirqd, kthreadd, kworker, swapper</td>
</tr>
<tr>
<td>Receive Processing</td>
<td>Kernel/Driver packet processing (e.g. ip rcv, net_rx_action, mlx5e_poll_rx_cc)</td>
</tr>
</tbody>
</table>

Table 1: Category examples and corresponding anomalies

To provide full-stack visibility, §4.1. Time reconciliation aligns observations from the CPU profiler and shim layer that use independent clocks. §6.1 examines the accuracy of time reconciliation which is crucial for tracking causality. To be lightweight, §4.1. Message profiling relies on hardware profiling while introducing software profiling latency overheads at a few points in the end-host stack. §6.2 examines the overheads of this profiling scheme for two reasons; first, to determine if it can produce reliable diagnosis despite the overheads and second, if NSight can be used in production. Finally to be high-fidelity yet unambiguous, §4.2. Anomaly disambiguation only reports a subset of anomalies that can explain a majority of latency deviations in messages as root causes. §6.3 examines the extent to which this technique is successful especially in the microsecond regime.

6.1 Time reconciliation correctness

Software clock changes make reconciling system activity, captured by CPU profilers, and message lifetimes, captured by shim layer, hard. To align system activity and message lifetimes correctly, §4.1. Time reconciliation must capture all software-clock change events and use the correct conversion between CPU profiling and software clocks at all points in time. If old or incorrect conversion is used, the system activity and message lifetimes will be incorrectly lined up, introducing spurious or missing system events in message lifetimes.

To test the accuracy of §4.1. Time reconciliation, we design a benchmark that continuously captures CPU profiling (TSC) and software timestamps (CLOCK_REALTIME) one after the other. When we use the conversion parameters captured by NSight’s user-space thread to convert all the CPU profiling timestamps to software timestamps, we expect to construct a linear timeline from the interspersed converted and measured software timestamps; that is, consecutive timestamps must always advance in time. If the old or incorrect conversion parameters are used in any time window, we detect a deviation from the linear timeline, showing that events observed across CPU profiling and the shim layer will be reordered.

Across multiple runs on different machines, over 10,000,000 conversions for each run, the consecutive converted and measured software timestamps always advance in time, never deviating. In each run, we observe that the software clock changes 4000+ times. This shows that the proposed time reconciliation maintains the ordering of events in message lifetimes even with software clock changes. We note
Table 2: Overhead of profiling tools on median and tail measurements. eBPF-n is eBPF used to profile n functions.

<table>
<thead>
<tr>
<th>Tool</th>
<th>All numbers in μs, h = high load, l = low load</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>median (h) 99.9th (h) median (l) 99.9th (l)</td>
</tr>
<tr>
<td>Baseline</td>
<td>30.3  112.6  10  14.4</td>
</tr>
<tr>
<td>Intel-PT</td>
<td>30.8 (2%) 120.4 (7%) 10.6 (5%) 15.3 (6%)</td>
</tr>
<tr>
<td>NSight</td>
<td>31.1 (3%) 132.8 (18%) 11 (10%) 16.2 (12%)</td>
</tr>
<tr>
<td>eBPF-1</td>
<td>38.6 (27%) 157.6 (40%) 11.8 (18%) 17.3 (20%)</td>
</tr>
<tr>
<td>eBPF-2</td>
<td>41.8 (38%) 165 (46%) 13.2 (31%) 18.6 (29%)</td>
</tr>
<tr>
<td>eBPF-4</td>
<td>51.9 (71%) 556 (39%) 14.1 (41%) 19.4 (35%)</td>
</tr>
<tr>
<td>eBPF-8</td>
<td>59.1 (95%) 565 (402%) 15.5 (54%) 21 (45%)</td>
</tr>
<tr>
<td>Ftrace</td>
<td>201.8 (565%) 1060 (841%) 40.1 (298%) 66.4 (359%)</td>
</tr>
</tbody>
</table>

that a stronger test for time reconciliation, in which we take two timestamps at the same time and test their equivalence is impossible. Capturing timestamps takes time, and is the reason for a majority of NSight’s profiling overheads.

6.2 Message profiling overheads

Latency overheads of profiling tools can perturb message lifetimes and obfuscate latency diagnosis. This is why these tools are only used for confirming hypotheses rather than diagnosis [15, 28, 32, 43, 45]. Even though NSight perturbs message lifetimes, it does not share this challenge because NSight is built on top of CPU profiling which profiles NSight itself. When NSight profiling slows down message lifetimes, it will show up in system activity captured by §4.1. Message profiling and §4.2. Anomaly disambiguation will detect it as the root cause. Perf Idle (A) in Figure 4 is an example where the CPU idles during NSight profiling introducing latencies.

Latency overheads of profiling tools also inhibit their use for latency measurement in production environments. To be useful in the microsecond-regime, we expect that this overhead should not be beyond a few microseconds.

To evaluate the latency overhead of NSight, we run a benchmark using memcached and record the receive latency of requests, measured from the time they arrive at the NIC to when they are received by memcached; this is the baseline measurement. A peak 4 core load on our system is 500k requests per second (rps); for this experiment we measured overheads across two loads on 4 cores, a low load (40k rps, 8% of peak), and a high load (300k rps, 60% of peak).

Table 2 shows the result of the experiment. Intel PT adds 2-7% over all measurements; Gathering software timestamps and core information with NSight adds another 1-11% totaling 3-18% overhead. This shows that NSight can be turned on in production for brief periods of time without perturbing median latencies by more than a few microseconds.

In contrast, as Figure 1 showed, profiling even a single function call, like __sys_recvmsg, using eBPF (eBPF-1) adds more overhead than NSight. As we increase the number of functions calls profiled with eBPF, its overhead increases (See eBPF-2, eBPF-4 and eBPF-8). Over all experiments, eBPF-1 adds 18-40% latency overhead. Ftrace, that profiles the full stack like NSight, has the largest impact on performance (up to 841% overhead). The high latency overheads of these tools make them impractical for use in production environments.

The current design for §4.1, Time reconciliation burns an entire core for the user-thread. This overhead can be prohibitive in production environments. Thankfully, the overhead is avoidable because software clocks change relatively slowly, once every 4 ms in Linux. If the software clock changes are tracked directly from the time synchronization protocol, there is no need for an extra core during profiling.

6.3 Coverage after anomaly disambiguation

For high-fidelity, latency diagnosis must be able to report root causes for all latency deviations seen in message lifetimes, even if the deviations are in the order of microseconds. We measure fidelity in terms of coverage, defined as the latency deviation explained by diagnosis as a fraction of overall latency deviation. However, §4.2. Anomaly disambiguation leads to less than 100% coverage. Trading off some coverage for getting rid of false negatives is still reasonable because NSight can be used iteratively to improve coverage; in each iteration the most conspicuous root causes remaining can be discovered with NSight and corrected for (See next section).

By not reporting parent anomalies, when 80% or more of their deviation is explained by their nested anomalies, we get a minimum coverage of 69% and a median coverage of 96% across all experiments and all observed message-latencies. Only 10 messages across our experiments have a coverage less than 87%. This shows that highlighting only anomalies that explain 80% or more of their parent’s latency deviation as root causes reduces false negatives and yet allows for a majority of root causes to be discovered in the first iteration.

7 Latency diagnosis with NSight

We now describe the iterative process by which NSight can quickly diagnose the causes of poor performance. Let us start with an initial system configuration that runs memcached in which large memcached tails are observed. Profiling the system with NSight identifies the prominent causes of tails. Users can mitigate or eliminate the causes found and profile the reconfigured system with NSight to identify the remaining causes of tails. Three such iterations help reduce memcached 99.999th percentile latency from 15.3 ms to 182 μs in our setup. Unfortunately, some of the mitigations we use increase median latencies by a few microseconds. NSight helps identify which mitigations cause this increase by comparing profiling data collected across iterations.

We now describe these iterations with NSight and the notable causes of network latency in our system. We present diagnosis results only for memcached server receive latencies because the majority of end-host delays are known to show up on the receive path [35, 39, 78].
7.1 First iteration

Initial configuration. Our setup consists of two Linux machines (2x14 Intel Xeon Gold 5120, 192 GB RAM), running kernel version 4.18 (Ubuntu 18.04) connected by 100Gbps interfaces to a 3.2Tbps Ethernet switch. memcached, a latency sensitive workload (similar to [20, 33, 55]) runs alongside NGINX, the interfering workload. Both applications use the default configuration and share 4 cores. The workload consists of memcached requests arriving at 160-190k requests per second (60-80% of expected memcached throughput across 2 cores in our system), and NGINX requests arriving at 50-60k requests per second (60-80% of expected NGINX throughput on a single core). ntp and ptp were disabled in our setup when we collected the experimental results in this section.

Initial diagnosis. To use NSight, users can collect a profiling sample of a few seconds and look at the initial result. In our setup, this allows us to profile 100k memcached request responses. The initial result, Figure 12, a balloon plot similar to Figure 3, shows the distribution of tail messages and their causes. The largest tails are caused by NGINX interference on the application core, NGINX (A). The remaining tails are due to three causes, mainly head-of-line blocking of application threads, HOL blocking (A), in combination with interference due to OS threads (A) and Receive processing (A) at the application core. These causes are defined in Table 1.

A second presentation, the box plot in Figure 13, summarizes the causes (X-axis) sorted by the median total deviation added to each tail message. The number of messages impacted by each cause, shown on the boxes, represents how pervasive the cause is. We notice that the most conspicuous causes impacting the most messages are to the right of Receive processing (R), starting with NGINX receive (A) that impacts 596 tail messages. Going after these can increase the latency reduction achieved with this iteration.

Of the causes in Figure 13, we find that HOL blocking (A) impacts the most tail messages (950), indicating that there is a lack of I/O parallelism in memcached. When we examine the tail message lifetimes slowed down by HOL blocking (A) (the top outliers in the box plot), we see memcached sends delaying memcached request-receives by as much as 1 ms. To understand why, we look at memcached code and find that memcached sends process up to 20 requests per socket, sending responses for each, before processing the next request. This is interesting because papers [20, 55] have conjectured that it is sufficient to use microsecond granularity core-scheduling to improve memcached latency, but in fact memcached itself foils that plan. The way to improve memcached latencies is to first modify memcached to introduce additional I/O parallelism as NSight identifies here.

7.2 Second Iteration

Second configuration. The next step is to mitigate or eliminate the causes identified in the previous diagnostic step. To eliminate NGINX (A) interference, we pin memcached to two cores and NGINX to a third core; similarly, to eliminate interference due to Receive processing (A) and NGINX receive (A), we pin kernel receive activity to a fourth core and configure RSS to send all receive traffic to that core (Similar to IOKernel [55]). To mitigate HOL blocking (A), we limit memcached-requests per socket to 2 (down from 20) and limit memcached server threads to one per core.

Second diagnosis. Having applied the second configuration we rerun NSight. Figure 14 and Figure 15 are the analogous presentations to Figure 12 and Figure 13. The largest latency deviations in Figure 15 come from Paging/Paging Idle (A). The message lifetimes impacted by these causes show repeated calls to change protection and change_prot_numa each taking up to 400 μs. The documentation for change_prot_numa() says that this code is a mechanism to identify beneficial page migrations; interestingly, it creates
shows that all the
7.3
(350 µs vs. 1
shows the CDF of the receive latencies with
are worst
derivations, and adaptive interrupt coalescing/P aging Idle (A)
Figure 7.3 Third iteration
the connection handler
causes tail latencies during startup.
message is received (on the
receive core
for connection setup using libevent. When a new connection
is dispatched to the handler on another thread that must be
scheduled on the
application core. The delay in scheduling
the connection handler causes tail latencies during startup.

7.3 Third iteration
Third configuration. We now mitigate the causes found in
Figure 15. We disable autonuma feature to confirm the Paging/Paging Idle (A) deviations, and adaptive interrupt coalescing to confirm NIC Delay (R) deviations, even though doing so might lead to worse performance. To speed up connection setup, we increase priority of new connection events by changing one line in memcached. Another cause we identified after the second iteration but before the third, was removed by disabling the LRU replacement feature; for compactness, we condense this additional iteration into the third iteration.

Third diagnosis. Having applied the third configuration we rerun NSight. The balloon plot Figure 16 shows that all the significant outliers are in the start up phase. The tail latencies of the initial messages are much lower than those seen with the second configuration in Figure 14 (350 µs vs. 1.4 ms). They are not caused by CPU Idle (A), as they were with the second configuration, but by HOL blocking (A). Overall, the tails are an order of magnitude smaller compared to the 1.1 ms-16 ms tails originally seen with the initial configuration in Figure 12.

7.4 Analysis of diagnosis and configurations
We now confirm tail latency improvements due to the diagnosis by running experiments with all three configurations, initial §7.1, second §7.2, and third configuration §7.3, for a longer duration without NSight profiling. We measure the latency of 10 million requests, using each configuration 10 times. Figure 17 shows the CDF of the receive latencies with these configurations; the left graph shows all the latencies and the right shows the tail latencies. With the third configuration, we see 99.9th percentile latency improve by $43 \times (2.2 \text{ ms to } 51 \text{ µs})$. The 99.999th percentile latency is 67 µs and 99.9999th percentile latency is 182 µs (down from 15.3 ms). Both, the second and third configurations improve the tail latencies compared to the initial configuration but at the cost of increasing latencies in the first 60% of messages.

Diagnosing latency increases in lower percentiles. NSight analyis latency anywhere on a latency curve, not just the tail. Since the deviations in Figure 17 are worst around the 25th percentile (13.3 µs with initial but 24.7 µs with the third configuration), users can configure NSight to focus on the 25th percentile. NSight will analyse a slice of
messages between the 20th and 30th percentiles. Since the latency increases occur across iterations, users can compare message lifetimes across iterations with NSight to identify the causes. We configure NSight to compare the 25th percentile messages in the third §7.3 and first iterations §7.1 to diagnose latency increases; the results in the rest of the section use this configuration. In this setting, latency deviation is defined as latencies that get worse around the 25th percentile in the third iteration compared to those in the first iteration.

Since all the messages around the 25th percentile are similarly impacted, we look for the most pervasive causes. Therefore, we turn to a global box plot that presents the causes sorted by the number of messages impacted, Figure 18. The most pervasive cause of latency, impacting all but one of the thousand messages around the 25th percentile in the third iteration relative to the first iteration, is HOL blocking (A). The messages also experience NIC delay (R) relative to messages in the first iteration. This is surprising because we disabled adaptive interrupt coalescing in the third iteration §7.3.

To identify why the 25th percentile messages of the third iteration experience head-of-line blocking relative to the first iteration, we consult NSight’s queuing breakdown described in Figure 19. It shows that a majority of the 25th percentile messages have at least one more message ahead of them in the third iteration relative to the first iteration (top half) and this introduces latency deviations (bottom half). App Send (A) shows more latency deviation than other categories. Kernel (A) and Kernel (R) measure send/receive queuing in the kernel. Because the second configuration (§7.2) isolated kernel receive activity from the application core, Kernel (A) shows latency deviations only due to sending responses. Receive queuing shown in Kernel (R) impacts fewer messages.

Together, deviations in App Send (A) and Kernel (A) show that sending responses takes longer in the third iteration even though fewer responses (reduced to 2 per socket in §7.2) are sent back per socket relative to the first iteration. This shows that the application core cannot keep up in the third iteration.

To find why, we consult NSight’s core context summaries for the iterations, shown in Figure 20. It shows the percentage of time the message lifetimes spend in each processing context. We see that memcached uses three cores to process the 25th percentile messages in the first iteration, Figure 20a, compared to two in third iteration, Figure 20b. This confirms that the
Figure 20: NSight core-context summary showing the process contexts (legend) active in the 25th percentile `memcached` message lifetimes on each core (X-axis), and the percentage of time they are active (Y-axis). `memcached` has three cores available in the first iteration vs. two cores in the third iteration, exacerbating head-of-line blocking in the third iteration.

Threads are backlogged in the third iteration because of the lack of available CPU; the mitigation of pinning `memcached` threads to cores (§7.1) increased head-of-line blocking for the 25th percentile messages in the third iteration.

Figure 20b also shows that in the third iteration, the core used for kernel receive activity is mostly idle/sleeping (`swapper` is the default idle task) waiting for a NIC interrupt, whereas in the first iteration the cores are always running, processing `memcached` requests or receiving messages in the kernel. This explains why we see deviations due to NIC Delay ($R$) in Figure 18 despite disabling adaptive interrupt coalescing; waking the core takes time and delays the NIC interrupt. This is an example of complex NIC–CPU scheduling interactions that NSight captures (§1). Waiting for the NIC interrupt at low loads puts the receive core to sleep since it has nothing else to do, and waking it up delays the NIC interrupt! Thus, the mitigation of pinning the kernel receive activity to a core in §7.2 adds latencies at lower loads to the common case.

8 Diagnosing VMA network stack

NSight can diagnose problems in different applications such as `redis` and different network stacks such as VMA network stack. We now describe the key causes of network tail latencies we found with unmodified `memcached` and `redis` on top of the VMA user-space network stack.

System configuration. For experiments with `memcached`, we pin `memcached` servers to 4 cores and increase the load (400k request/s on 4 cores). For experiments with `redis`, a single-threaded server, we profile a single core `redis` instance using the standard `redis-benchmark` (110k request/s on 1 core). Since these applications are already pinned to cores, studying application interference with NGINX is irrelevant and we do not include it in contrast to the Linux study.

Diagnosis. We now describe the key sources of network latency in VMA found using the same diagnostic strategy we used in the Linux study, guided by NSight’s graphs. As expected, the overall tail latency distribution improves with VMA in comparison to Linux. But surprisingly, the outliers for `memcached` are more severe. While the median `memcached` server request receive latency is 8.38 µs and the 99.9th percentile latency is 45.3 µs, the worst message latency is 34.5 ms! The median, 99.9th and worst `redis` request receive latencies are 1.63 µs, 145.1 µs, and 1.2 ms. Following are the most prominent causes of tails in VMA.

VMA epoll mechanism. NSight diagnostic graphs show that some `memcached` messages are delayed for up to 25 ms in the NIC! In this case, even though the message is received by the NIC, the epoll implementation of VMA does not pick it up. The exact reason for this behavior is unclear, but we posit that either the NIC hardware delays reporting the arrival or the stack waits for a message for a specific socket.

OS interference. We find that the default Linux scheduling policies frequently puts polling based stacks to sleep for short amounts of time (4 µs). In the case of some `memcached` messages that are severely delayed, we detect a kernel stack overflow, that puts all but one of the application threads to sleep for up to 12 ms with the lone thread handling the panic. This is unexpected behavior that seems to suggest a bug.

Buffer management. VMA ring buffer management causes frequent shorter latency deviations. It fills up the RX buffer queue with unused buffers and removes used send-buffers from the TX queue in bursts, adding deviations of up to 2.8 ms (though more frequently in the range of 60-150 µs).

9 Limitations and future work

We have already noted three limitations of NSight. First, NSight cannot be used for continuous profiling due to Intel-PT buffering implementation (§5.1). Second, CPU profilers capture system activity in batches of CPU cycles; NSight cannot capture latency deviations smaller than a batch (§5.2). Finally, consuming a core for §4.1, Time reconciliation is unnecessary if software clock changes are tracked by ptp. Another limitation of NSight is that it produces 600MB-1GB of compressed raw profiling data per second. Decompressing the profiling data to a usable format increases the size by 10-20x. Reducing the amount of data produced and speeding up analysis will reduce the time between NSight iterations.

We are expanding NSight’s scope to RDMA-based stacks, and more general purpose performance diagnosis to analyze Linux’s core operations [63]. We are also using NSight to characterize the design space for low-latency network stacks. We plan to make the tool available as open source.

Acknowledgements

We thank our shepherd, Robert Ricci, anonymous reviewers, Jon Howell, Amin Vahdat, Vyas Sekar, Marcos Aguilera, Ben Pfaff, Ming Liu, Adriana Szekeres, Naama Ben David, Nadav Amit, Amy Tai, Irina Calciu, Jayneel Gandhi, Ana Klimovic, Lukas Humbel and Michael Wei for their insightful feedback.
References


