# Chronicle: Capture and Analysis of NFS Workloads at Line Rate

Ardalan Kangarlou, Sandip Shete, and John D. Strunk, *NetApp, Inc.*

https://www.usenix.org/conference/fast15/technical-sessions/presentation/kangarlou

This paper is included in the Proceedings of the
13th USENIX Conference on
File and Storage Technologies (FAST '15).
February 16–19, 2015 • Santa Clara, CA, USA
ISBN 978-1-931971-201

Open access to the Proceedings of the
13th USENIX Conference on
File and Storage Technologies
is sponsored by USENIX

# Chronicle: Capture and Analysis of NFS Workloads at Line Rate

Ardalan Kangarlou, Sandip Shete, and John D. Strunk
*NetApp, Inc.*

## Abstract

Insights from workloads have been instrumental in hardware and software design, problem diagnosis, and performance optimization. The recent emergence of software-defined data centers and application-centric computing has further increased the interest in studying workloads. Despite the ever-increasing interest, the lack of general frameworks for trace capture and workload analysis at *line rate* has impeded characterizing many storage workloads and systems. This is in part due to complexities associated with engineering a solution that is tailored enough to use computational resources efficiently yet is general enough to handle different types of analyses or workloads.

This paper presents *Chronicle*, a high-throughput framework for capturing and analyzing Network File System (NFS) workloads at line rate. More specifically, we designed Chronicle to characterize NFS network traffic at rates above 10Gb/s for days to weeks. By leveraging the actor programming model and a pluggable, pipelined architecture, Chronicle facilitates a highly portable and scalable framework that imposes little burden on application programmers. In this paper, we demonstrate that Chronicle can reconstruct, process, and record storage-level semantics at the rate of 14Gb/s using general-purpose CPUs, disks, and NICs.

## 1 Introduction

The storage industry is in a state of flux. As new workloads emerge and the characteristics and economics of the storage media change, it is vital to reevaluate the design of storage systems. Many of yesterday's caching, prefetching, and data tiering techniques have limited applicability to today's workloads and hardware. The recent emergence of software-defined data centers and data-driven management that support a more application-centric view of storage has further increased the interest in workloads. The latest trends aside, characterizing traditional workloads and storage systems is also critical. Designing benchmarks for legacy workloads and troubleshooting deployed systems, product and service selection, capacity planning, and billing all hinge on some understanding of workloads.

Despite the ever-increasing interest, the lack of high-quality workload traces and general workload analysis frameworks have been major stumbling blocks in characterizing storage workloads and systems. Ideally, workload traces should be thorough and fine-grained enough to accurately capture the dynamics of the workloads. An I/O-by-I/O view of workloads undoubtedly provides richer insights compared to views based on aggregate statistics or sampling. Additionally, the trace collection and analysis procedure should cause the least amount of interference with the systems under study.

This paper presents *Chronicle* [2], a high-throughput framework for capturing and analyzing workloads at line rate for an extended period of time. Specifically, we designed Chronicle to characterize Network File System (NFSv3) traffic at rates above 10Gb/s for days to weeks. Chronicle runs as a Linux-based middlebox that *passively* monitors network traffic via network taps or port mirroring. The most important aspect of Chronicle is that, through deep packet inspection (DPI), it can reconstruct storage protocol semantics at line rate. Chronicle has the flexibility to capture long-term traces, perform real-time analytics on the in-flight network traffic, or do some combination of both.

We favored a middlebox approach over instrumenting NFS servers or clients because it is independent of the systems under study; and more importantly, it has no impact on system performance. We also opted for a solution based on commodity hardware. Although there are very efficient solutions based on specialized hardware, such as FPGA packet capture cards, monitoring in the networking hardware (e.g., [5, 35]), or GPUs (e.g., MIDeA [33] and PacketShader [21]), these systems tend to be limited in scope. Capture cards are generally limited to timestamping and DMAing packets; networking hardware is generally optimized to perform a few operations like lookup, filtering, and counting; and GPUs excel for applications that conform to the single instruction, multiple threads (SIMT) mode of programming.

Earlier efforts in fields such as software routing, network security, and software-defined networking (SDN) have shown the applicability of general-purpose hardware for high-speed packet processing. For instance, RouteBricks [16] achieves a throughput of 10Gb/s, 6.4Gb/s, and 1.4Gb/s for forwarding, routing, and IPsec encryption of 64B packets respectively. In this paper we demonstrate that general-purpose hardware can also handle more complex operations like TCP reassembly, pattern matching, data checksumming, compression, real-time analysis, and trace storage at rates higher than 10Gb/s.

Parallelizing packet processing using multicore architectures has been the focus of many efforts in the past [12, 16, 19, 27, 29, 34]. Partitioning and pipelining work across threads, judicious placement and scheduling of threads for better cache hit rates, and minimizing synchronization overhead (e.g., by using lock-free data structures for thread communication) are a few examples of the techniques discussed in the literature. However, constructing such carefully engineered systems imposes great burdens on programmers, because it requires intimate knowledge of hardware platforms and careful management of shared state and resources among threads. These designs often result in systems that are heavily tailored to specific hardware platforms and require manual tuning.

To address these challenges, we developed a user-space programming library, called *Libtask*, which hides such complexities from application programmers. Libtask is based on the actor model paradigm [22] and enables a lock-free, pluggable, pipelined architecture for applications that use it. There are several advantages to this architecture: (1) applications built on top of Libtask are completely decoupled from the underlying hardware, resulting in highly portable and scalable software; (2) interactions among threads are well-defined, thus reducing the possibility of concurrency bugs; and (3) supporting different types of input sources, output formats, analyses, and protocols, beyond what we demonstrate with NFS, simply involves plugging in the right module in the pipelined architecture.

To the best of our knowledge, Chronicle is the first system of its kind to show the applicability of the actor programming model to workload capture and analysis. Another novel aspect of Chronicle is that we extend zero-copy packet parsing to what is considered the application layer in the OSI reference model [36]. Therefore, there is no need to copy packet payloads to a contiguous buffer to reconstruct storage-level semantics. We demonstrate the versatility of the Chronicle framework by describing the implementation of two pipelines, one for trace capture and one for characterizing NFS workloads. These pipelines can operate at the rate of 14Gb/s using only 8 cores, a testament to the framework's efficiency. We have successfully deployed Chronicle in a number of production environments. Our intent is to create a comprehensive trace library that represents different classes of workloads across the industries that constitute our customer base.

The rest of this paper is organized as follows: Section 2 describes some of the main differences between Chronicle and earlier work. Section 3 presents a high-level overview of the Chronicle architecture. Sections 4 and 5 describe the implementation of Libtask and Chronicle pipelines respectively. Section 6 highlights unique aspects of Chronicle relative to other frameworks. Section 7 presents a comprehensive evaluation of Libtask and Chronicle, and Section 8 briefly discusses of some of the insights gained through implementing and using Chronicle.

## 2 Related Work

Capture and analysis of network storage workloads (e.g., NFS and CIFS) have been the focus of a few efforts in the past [8, 18, 26]. Of these efforts, Driverdump [8], a system based on modifying the network driver to directly store packets in the pcap format, is the most powerful software-only solution that can operate at the rate of 1.4Gb/s. It is unfair to directly compare the performance of Chronicle to these systems because of the hardware advances. Instead, we would like to highlight the unique features of Chronicle that have advanced the state of the art in capture and analysis of network-attached storage (NAS) workloads. These features can be summarized as (1) TCP reassembly; (2) inline parsing; and (3) efficient trace storage. As a result, Chronicle can characterize workloads at higher rates, for a longer time, and with better coverage of I/O operations compared to all the previous efforts.

The use of multiple cores for efficient packet processing is an active area of research in packet forwarding and software routing [10, 12, 16, 17, 21, 27, 32]. These efforts differ from Chronicle in that they typically do not perform any DPI and are limited to parsing the network header. This simple difference, however, has great implications for Chronicle with respect to programmability and functionality.

Kernel frameworks, such as the elegant and modular kernel-mode Click [25], require expert knowledge to extend them in a performance-optimized way. Extending such frameworks with arbitrary types of processing (e.g., custom or preexisting libraries for parsing, compression, etc.) can be especially daunting for nonexperts. The recent port of netmap [31] to user-mode Click along with techniques such as batching packet processing and recycling allocated memory, have improved the forwarding throughput by 10x, close to the throughput of Click's kernel-mode implementation [32]. In Section 6, we extensively compare the implementation of Chronicle with a few well-known packet-processing frameworks and demonstrate that our actor model framework facilitates *implicit* concurrency, serialization, and batching to achieve high throughput.

Dobrescu, Argyraki et al. [15] proposed a framework to eliminate the "tedious manual tuning" that underlay RouteBricks [16]. They devised a formula to identify the optimal parallelization strategy when packet-processing elements can be cloned or pipelined across cores. This type of framework tends to be effective in scenarios where the exact processing cost of each packet is known. In Chronicle's application scenario, per-packet processing cost can be quite variable, because factors such as packet reordering and the type of the NFS operation embedded in a packet affect the processing overhead.

Many papers and projects (e.g., [1, 13, 29, 34]) have addressed efficient use of multicore architectures for DPI or network monitoring. In addition to supporting lower rates, many of these systems have a much narrower scope than Chronicle because (1) their implementations are tied to specific multicore architectures; (2) many do not do TCP reassembly; and (3) DPI is not performed on the whole packet payload. De Sensi [14] addresses some of these limitations by leveraging structured parallel programming on top of FastFlow [7].

DPI on FastFlow is similar to Chronicle, in that both define higher-level abstractions for users to represent a workflow. The main difference lies in the programming model. For example, actors cannot share state in the actor model paradigm (Section 4). Another difference is zero-copy parsing beyond the network layer by Chronicle. Unfortunately, a direct comparison of the throughput of the two frameworks is not possible because DPI on FastFlow was evaluated using HTTP network traces, as opposed to *live* NFS traffic, and presumably with less CPU-intensive processing compared to our evaluation scenario; however, this framework could operate at 11Gb/s.

## 3 Chronicle Overview

This section outlines at a high level the design and architecture of Chronicle. It also describes and justifies some of the design decisions we made to address many challenges of workload characterization at line rate. We especially highlight three important challenges: (1) DPI to construct application layer semantics; (2) trace storage at line rate; and (3) efficient use of CPU cores.

Although it is not unique to Chronicle, it is important to point out that performing DPI to construct application layer semantics is more involved than simply examining a few bytes of packets beyond the network header. This complexity is due to the fact that the TCP/IP layer is completely oblivious to the nature of the application layer data it transports. For instance, to characterize NFS traffic over TCP/IP, Chronicle needs to handle situations where an RPC protocol data unit (PDU) starts in the middle of a packet or crosses multiple packets. Therefore, unlike high-speed packet forwarding and routing, this type of DPI requires reassembly of TCP segments and *stateful* parsing *across* packets. Additionally, TCP reassembly should cope with packet loss and packet retransmissions.

Another important challenge is trace storage at rates higher than 10Gb/s. At such high rates, storage bandwidth can easily become a cause for concern. We could use a high-end array of disks or SSDs, but that would conflict with our goal of using affordable off-the-shelf hardware. Additionally, workload capture for an extended period of time at these rates requires a considerable amount of storage. For example, capturing network traces using a standard tool like tcpdump at 10Gb/s for a week requires more than 750TB of storage.

We use three techniques to address these data storage challenges. The first technique is to prune the raw data that we capture off the wire. One by-product of performing DPI inline is that we can identify fields of interest in the stream of bytes we capture. For example, in the context of our NFS workload capture implementation, Chronicle records several fields of interest in the network header and almost all of the RPC and NFS fields. The second technique is inline checksumming of the NFS read and write data, which results in substantial savings over storing the raw data for data deduplication analysis [28]. The third technique is to perform inline compression prior to writing traces. By directly writing traces in the DataSeries [9] format, we can leverage DataSeries' inline compression, nonblocking I/O, and delta encoding functions, which reduce both the bandwidth and capacity requirements of trace storage. These techniques collectively reduce the amount of data recorded to a rate that a single standard disk can handle.

Although performing DPI along with inline compression and checksumming help to alleviate the storage bottleneck issues, these techniques come at the expense of increased CPU utilization. As illustrated by other high-throughput systems such as PacketShader [21] and RouteBricks [16], excessive processing at high rates can easily make CPU the bottleneck and hurt the overall throughput of the system. For instance, RouteBricks achieves a throughput of 10Gb/s for forwarding 64B packets, but performing more complex operations like routing or IPsec dropped the throughput to 6.4Gb/s and 1.4Gb/s respectively. Similarly, with netmap [31] packets can be received at 14.88Mpps (at 10Gb/s), but Open vSwitch [30] packet forwarding on top of netmap drops the throughput by more than 75% [32]. Considering the more complex nature of the operations performed by Chronicle, efficient use of CPU cores is critical.

The rest of this section describes a host of techniques that prevent CPU from becoming the bottleneck. The first technique, and arguably the most important one, is the use of the *Libtask* library. Section 4 describes the implementation of Libtask in detail. Here we briefly discuss our main objectives in designing Libtask and its place in the Chronicle architecture. Libtask's main purpose is to provide seamless scalability to many cores. It enables a pluggable, pipelined architecture, in which each module performs a different task in parallel. Applications written on top of Libtask can then use all the cores in the system without any knowledge of underlying hardware, such as the number of cores or their topology. The seamless scalability to many cores results in great portability for the Chronicle software. Additionally, the pluggable, pipelined architecture results in a very flexible framework in which, by chaining the right set of modules, Chronicle can capture traces, do statistical analysis, or perform some combination of both. Section 5.1 covers the Chronicle pipelines extensively.

The second technique is zero-copy packet parsing at both the application and network layers. As packets pass through different modules in our pipelined architecture, each module parses a specific layer (e.g., the network, RPC, or NFS layer) or performs some kind of computation based on information from previous pipeline modules (e.g., checksumming of the read/write data or compression). Therefore, to keep the overhead of the pipelined architecture low, it is imperative to avoid any sort of copying between different modules. Section 5.2 elaborates on our zero-copy packet parsing method.

The third technique is the use of custom network drivers, which allows a user-space application to bypass the kernel when reading packets. Techniques such as DPDK [3] and netmap [31] are proposed to eliminate most of the overhead associated with packet processing in standard operating systems, like the overhead of copying packets, memory allocation for packet descriptors (e.g., sk_buff structures in Linux), and interrupt processing. Our Chronicle implementation uses netmap to read packets from the NICs.

# 4 Libtask Library

We developed Libtask as a general actor model (AM) [22] library that facilitates seamless scalability to many cores. Central to the AM paradigm are the concepts of *actor*, *task*, and communication among actors. An actor refers to a computational agent that processes tasks. Each task is addressed to a target actor and includes some *message*, which is the information to be shared with the target [6]. As actors process the messages in tasks, the computation in an AM system advances. Processing a task by an actor can lead to sending a message (either to itself or to some other actor), creation of new actors, or actor replacement.

Two aspects of the AM programming that make it highly attractive to high-throughput computing are no sharing of state and asynchronous communication among actors. In this paradigm, the only way in which actors can affect each other is through sending messages (as opposed to sharing variables). The outcome is a very modular design in which bugs caused by concurrent execution can be easily avoided. Asynchronous communication among actors is necessary for an actor to send a message to itself and is desirable for our purposes because actors do not block to receive acknowledgements from targets. These properties enable a highly scalable and programmer-friendly framework in which many actors can be created and pipelined to carry out tasks in parallel.

Many languages such as D, Erlang, and Scala (with Akka toolkit) have borrowed concepts from the AM framework. Additionally, there are languages like Go that are based on the somewhat similar paradigm of Communicating Sequential Processes (CSP) [23]. Instead of relying on existing AM frameworks, we decided to implement our own standalone AM library in C++ for better performance and a finer level of control. Libtask is quite lightweight and small (fewer than 2,000 lines of code).

The rest of this section describes a few Libtask constructs such as Process, Scheduler, and Message. A Libtask Process is equivalent to an actor, and its implementation can be thought of as an event-driven state machine that performs a certain task. A Process has complete ownership of the data it processes. Therefore, there is no sharing of state among different Processes, as specified by the actor model. Each Process has a queue for receiving incoming Messages and is runnable as long as there is a pending Message in its queue.

A Scheduler's job is to schedule and run Processes. Each Scheduler has a queue of runnable Processes. On the occasion that the queue becomes empty, the Scheduler may steal a Process from other Schedulers. The Scheduler's run queue can become empty either as a re-

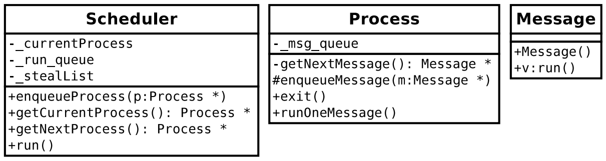

Scheduler
-_currentProcess
-_run_queue
-_stealList
+enqueueProcess(p:Process *)
+getCurrentProcess(): Process *
+getNextProcess(): Process *
+run()
Process
-_msg_queue
-getNextMessage(): Message *
#enqueueMessage(m:Message *)
+exit()
+runOneMessage()
Message
+Message()
+v:run()

Figure 1: Simplified UML diagram for Libtask.

sult of exhausting its list of runnable processes or due to Process-stealing. Upon getting scheduled, a Process runs a bounded number of Messages, where each Message is run to completion before the next is processed. The Scheduler is implemented as a POSIX thread, and there are typically as many Schedulers as there are logical cores (i.e., hardware threads) in the system.

We have developed two versions of Libtask for balancing load across cores. In one version, Schedulers are pinned to distinct logical cores and prefer to steal Processes from Schedulers running in the same NUMA node (i.e., the same CPU), thus preserving warm caches. In the other version, Schedulers are not tied to specific cores and perform NUMA-agnostic Process-stealing. Section 7.1 compares the performance of these two versions relative to implementations in Erlang and Go.

Processes use Messages as the communication mechanism between themselves. The main purpose of a Message is to specify the action to be performed by the next Process in the pipeline. A Message can contain the data to be passed or some reference to it. In either case, a Message exchange signals the transfer of data ownership between the sender and the target. Figure 1 illustrates the simplified UML diagram for the Libtask classes described earlier. One point to note is that Process::enqueueMessage is implemented as a protected method. This is to ensure that the sending and receiving Processes agree on the exchanged Message type or types. All subclasses of Process have a public wrapper method for Process::enqueueMessage (one per Message type) to force type checking at compile time.

# 5 Chronicle Implementation

We have implemented Chronicle as multiple pipelines of Libtask Processes. Each Process corresponds to a *module* in the Chronicle architecture, and performs functions such as parsing, computation, and trace storage. All modules and all messages exchanged between them are implemented as subclasses of Process and Message respectively. Section 5.1 describes Chronicle pipelines and Section 5.2 describes the zero-copy parsing method used by different pipeline modules.

## 5.1 Chronicle Modules and Pipelines

Figure 2(a) depicts the high-level view of the Chronicle architecture. This figure shows a few Packet Reader and Network Parser modules and a few Chronicle pipelines. Each pipeline itself is made up of more modules, as illustrated in Figure 2(b). The rest of this section describes the functions of each module and its role in the overall architecture. Discussion of the Messages passed between modules is postponed to Section 5.2.

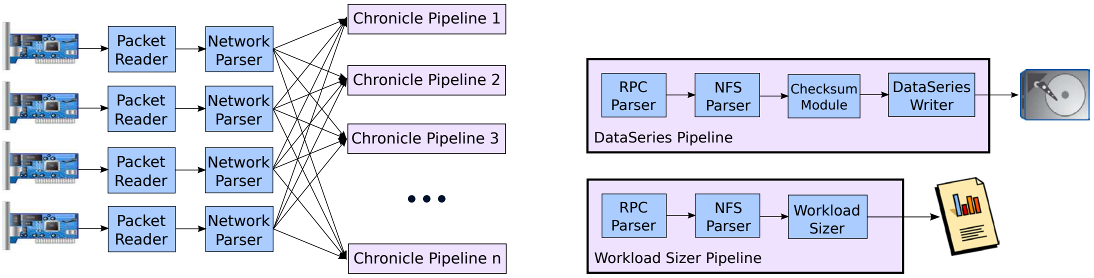
A Packet Reader (Reader) module reads Ethernet frames from a NIC using the netmap [31] drivers mentioned in Section 3. We made small changes to netmap to support jumbo frames and larger buffer sizes. Our implementation dedicates one Reader per NIC. However, for modern NICs that have multiple queues, it is possible to dedicate one Reader per queue for faster processing of the packets. Each Reader polls the corresponding NIC, timestamps all the available packets, and copies them to an internal packet buffer pool.

The main functions of a Network Parser module are parsing the network header portion of a packet and multiplexing further processing across different Chronicle pipelines. A Network Parser parses L2, L3, and L4 headers in an Ethernet frame and retrieves information such as source and destination IP addresses and port numbers, TCP sequence number, and TCP payload offset. It then uses the 5-tuple of source IP address, destination IP address, source port number, destination port number, and transport protocol to delegate further processing to one of the Chronicle pipelines. To avoid cross-pipeline communication or locking, Network Parser designates the same pipeline to process the packets belonging to either direction of a connection.

Figure 2(b) illustrates two examples of the pipelines that Chronicle currently supports. The *DataSeries Pipeline* is the pipeline of choice for trace capture at high rates due to the reasons mentioned in Section 3. We use the *Workload Sizer Pipeline* as an example of a pipeline whose purpose is to perform real-time analytics on the NFS traffic. The rest of this section describes these pipelines and their constituent modules.

### 5.1.1 Trace Capture Pipeline

The DataSeries pipeline receives a stream of packets on one end and generates traces in the DataSeries format [9] on the other end. The DataSeries format is characterized by efficient storage of structured serial data. Each DataSeries trace file is composed of a series of records, where each record is in turn composed of a series of fields. The records of the same type are organized into groups of extents, which are similar to tables in databases. For example, in our application scenario, we have one extent type for storing network-level infor-



(a) Chronicle modules and pipelines

(b) Different Chronicle pipelines

Figure 2: Overview of the Chronicle architecture.

mation, another extent type for storing RPC header information, and additional extent types for different NFS operation types. The records in these extents correspond to a packet, to an RPC PDU, and to an NFS operation respectively. We use a common field called *record_ID* to link related records in the network, RPC, and NFS extents. The DataSeries Writer module at the end of the DataSeries pipeline is responsible for storing traces in the DataSeries format. This module makes extensive use of a few features of DataSeries such as inline lzf compression, relative packing (delta encoding), and unique value packing.

For DataSeries Writer to store all the fields of interest in the DataSeries format, it has to rely on information provided by the preceding modules in the pipeline, starting with the RPC Parser module. This module performs the following main functions: (1) filtering of TCP and RPC traffic; (2) reassembly of TCP segments; (3) detection and parsing of RPC headers; (4) construction of RPC PDUs; and (5) matching RPC replies with the corresponding calls.

The RPC Parser module is the key module that facilitates DPI on NFS traffic. To perform DPI, this module needs to have some kind of receive-side TCP functionality to handle in-order, out-of-order, and retransmitted packets. Given that Chronicle passively captures network traffic via network taps or port mirroring, it is possible for an RPC Parser to see an acknowledgement for a TCP segment that will be seen in the future or that will never be seen. Under these circumstances, we could not rely on a standard TCP implementation and had to develop a custom TCP reassembly facility.

Packet losses and out-of-order packets directly impact the performance of the RPC Parser module and the overall throughput of Chronicle. In the absence of any losses and out-of-order packets, the identification of RPC header in the byte stream is very straightforward, because the length of a PDU is part of the header. Advancing from the current RPC header by the length of a PDU results in finding the next RPC header in the byte stream. This technique works well not only for cases where an RPC header starts immediately after the TCP header but also for commonly occurring cases where a PDU starts in the middle of a packet, when a packet contains multiple PDUs, or when a PDU spans multiple packets. When an RPC Parser uses this technique to find RPC headers, we deem that it is operating in the *fast mode*. Unfortunately, this technique falters in the event of packet loss or out-of-order delivery of packets and causes the module to enter the *slow mode*. In the slow mode, the RPC Parser module has to scan the byte stream and perform pattern matching to find an RPC header based on its signature. Once a header is found, RPC Parser can return to the fast mode if a complete PDU is present.

The NFS Parser module is responsible for parsing the NFS fields in an RPC PDU. The values for these fields are provided to the DataSeries Writer module to supply the records for the NFS operation-specific extents in a DataSeries trace file. The Checksum module operates on NFS read and write PDUs and computes 64-bit checksums for 512B read/write data blocks at 512B-aligned offsets. These checksums are also passed to the DataSeries Writer module to be stored in a data checksum extent. The checksums computed by this module can be used for online or offline data deduplication analysis [28].

In addition to supporting NICs with netmap drivers for input, Chronicle supports input from NICs or files through the standard pcap interface. It also supports writing traces in the pcap format. We will not elaborate these capabilities much further for the following reasons: (1) pcap NIC interfaces are quite lossy at high data rates; (2) the focus of this paper is to characterize *live* NFS network traffic; and (3) trace storage in the pcap format is quite bulky and requires further parsing of the data. However, these capabilities demonstrate the flexible nature of our pluggable, pipelined architecture where supporting new input sources, output formats, or protocols merely involves plugging the right set of modules in the right place in the pipeline.

### 5.1.2 Workload Sizer Pipeline

Another example of our flexible, pipelined architecture is a pipeline for sizing storage workloads. Workload sizing is a pre-sales practice in the storage industry to identify the right platform for a given workload. A sizer typically takes as input workload-specific information such as the rate of I/Os, the random read working set size, and the ratio of random reads and writes, and generates as output the number of heads and spindles as well as the estimated CPU and disk utilization levels for different storage platforms.

The main function of the Workload Sizer module is to generate a workload profile that will be used as input to an off-box sizer. This module processes each I/O request and leverages synopsis data structures [20] due to their speed in absorbing updates and their small memory footprint. This module also performs top-k analysis [11] and quantile calculation. Other examples of real-time analysis that Chronicle can support are pipelines to determine the data deduplication rate, the hottest files by the number of bytes or requests, and the most active clients. The insights from these pipelines are helpful in dynamically tuning a storage system.

## 5.2 Zero-Copy Packet Parsing

Zero-copy parsing at the network level is a standard practice and has been used extensively in operating systems and packet processing frameworks to avoid the cost of data copy between different modules. Our contribution is that we extend zero-copy parsing to the application layer. Our approach is novel in that it does not require copying packet payloads to a contiguous buffer to reconstruct application layer semantics.

The key idea behind our parsing technique is to maintain ancillary data structures on top of the packet buffer pool. Each entry in the buffer pool has a corresponding, fixed *packet descriptor* structure that serves as a handle to a particular buffer pool entry and holds all network-level information about a packet (either the data itself or its offset). Packet descriptors are allocated once at the beginning of a Chronicle run as opposed to upon every packet arrival.

Upon receipt of a packet, the Packet Reader module passes the corresponding descriptor to the Network Parser module. The Network Parser module populates most entries in the descriptor and passes it along to the appropriate Chronicle pipeline. The RPC Parser module then chains packet descriptors belonging to the same flow based on their TCP sequence number values. Prior to chaining, the RPC Parser may adjust packet descriptors to ensure that no two descriptors overlap in the TCP sequence number space.[1] When an RPC Parser finds an RPC PDU, it creates another ancillary data structure called a *PDU descriptor*. Each PDU descriptor holds RPC- and NFS-level information and points to the chain of packet descriptors that constitute the PDU. The RPC Parser then passes PDU descriptors to the next module in the pipeline. Packet or PDU descriptors passed between modules are the embodiment of the actor model messages described in Section 4.

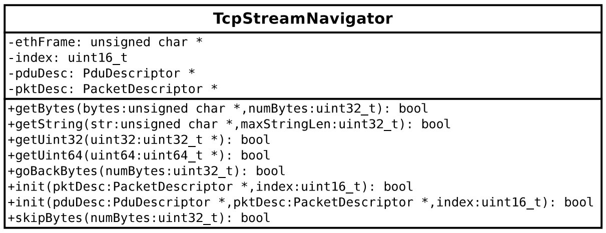

TcpStreamNavigator
-ethFrame: unsigned char *
-index: uint16_t
-pduDesc: PduDescriptor *
-pktDesc: PacketDescriptor *
+getBytes(bytes:unsigned char *,numBytes:uint32_t): bool
+getString(str:unsigned char *,maxStringLen:uint32_t): bool
+getUint32(uint32:uint32_t *): bool
+getUint64(uint64:uint64_t *): bool
+goBackBytes(numBytes:uint32_t): bool
+init(pktDesc:PacketDescriptor *,index:uint16_t): bool
+init(pduDesc:PduDescriptor *,pktDesc:PacketDescriptor *,index:uint16_t): bool
+skipBytes(numBytes:uint32_t): bool

Figure 3: Simplified UML diagram for the facility that navigates TCP byte streams.

The main enabler for the application layer zero-copy parsing is the implementation of a facility for traversing packet payloads. Figure 3 presents a simplified UML diagram for this facility. This facility maintains a point of reference, which consists of a packet descriptor and a byte offset in the payload, and uses the TCP information to retrieve certain bytes in the byte stream. We implemented an XDR parser on top of this facility for parsing the RPC header and NFS fields. One important aspect of this facility is that it enables parsing data (e.g., a single field or a group of fields as in the RPC header) that cross multiple packets. This capability is unique to Chronicle and does not exist in previous NFS tracing efforts (e.g., [8, 18]) and in standard tools like Wireshark. Another advantage is that it enables skipping all nonrelevant bytes for the DPI task at hand without any data copy.

# 6 Comparison with Other Frameworks

This section compares and contrasts Chronicle with the implementation of a few high-throughput packet-processing frameworks. On the surface, there are many similarities between Chronicle and frameworks like Click [25]. For instance, a Click router consists of a number of modules called elements. These elements can get pipelined, packets can get multiplexed across pipelines of elements, and there is zero-copy packet parsing across elements. Additionally, elements can run in the context of multiple scheduler threads [12]. However, there are some subtle differences, particularly with re-

[1] This condition may occur as a result of TCP retransmissions.

spect to the application scenario and programmability, which are highlighted in the rest of this section.

**Latency vs. Throughput:** For a software router like Click, low latency in the processing path can be as crucial as high throughput. The processing path in Click typically consists of a sequence of push and pull elements, where each element either pushes a packet to a downstream element or pulls a packet from an upstream element. *Queue* elements are typically used only when there are transitions between pull or push paths or when multiple paths converge to temporarily store packets.

Because only the source of a push path and the sink of a pull path are schedulable elements, other elements in the path must run in the context of the same thread that schedules the source or sink element [12]. This implementation minimizes thread communication, reduces scheduling overhead and cache conflicts, and imposes minimal queuing delay, which together reduce processing latency. However, to improve throughput, recent efforts [24, 32] have suggested better use of I/O and computation batching so that an element can process multiple packets at a time. In our application scenario, achieving high throughput is the primary objective and Chronicle's actor model architecture, with the implicit queues at independently schedulable processing elements, facilitates *seamless* I/O and computation batching.

**Explicit vs. Implicit Parallelism:** Despite some similarities, parallelism in Click and Chronicle is different in a number of ways. First, every module in Chronicle is schedulable and can run on any core. Second, with the exception of Packet Readers, when a Chronicle module gets scheduled, it is guaranteed that it has some useful work to do. This is a side effect of our implementation, where a Libtask Process gets placed on a Scheduler's queue only when it has a pending Message, and the fact that Processes only "push" Messages to each other. Third, for some frameworks, certain layouts of modules require use of thread-safe queues or modules. A positive aspect of our actor model implementation is that such complexities are not exposed to the users of Chronicle because the framework itself provides *implicit* parallelization and serialization.

**Network vs. Application Layer:** Differences between packet processing at the network and application layers explain some of the design decisions behind Chronicle. For instance, parsing a network header is generally not CPU-intensive enough to justify the use of multiple cores per packet. Therefore, spatial assignment techniques (e.g., NetSlices [27] and TNAPI [19]) that impose fixed mappings between packets and cores are very efficient for parsing network headers. On the other hand, these techniques may result in load imbalances and CPU underutilization when processing is expensive or variable. In fact, as we discovered during the evaluation of Chronicle (Section 7), in some scenarios, pinning threads to cores may have adverse effects on throughput. Another issue is that parsing at the application layer requires a framework to be general enough to support different application layer constructs beyond just packets (e.g., RPC PDUs). Our general actor model framework again seamlessly facilitates efficient use of cores as well arbitrary types of application layer constructs.

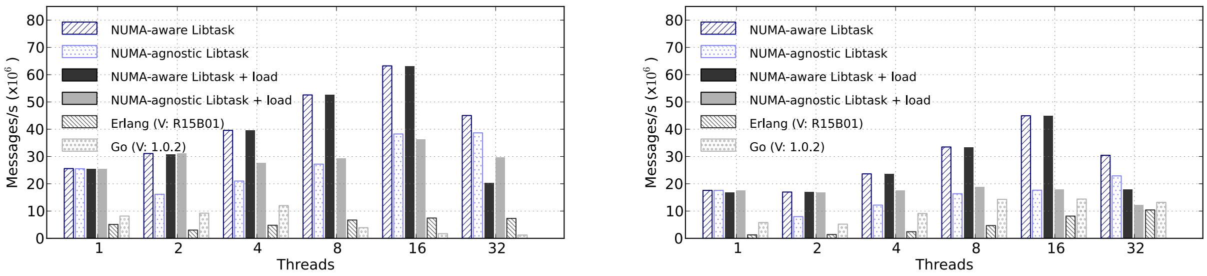
## 7 Evaluation

This section presents a comprehensive evaluation of Libtask and Chronicle. For these experiments, Libtask and Chronicle run on a server with two Intel Xeon E5-2690 2.90GHz CPUs. Each CPU has 8 cores (16 logical cores or hardware threads). The server is configured with 128GB of 1600MHz DDR3 DRAM memory (64GB per CPU). Additionally, it has two dual-port Intel® 82599EB 10GbE NICs, which allows capture from two tapped links or four mirrored links. The storage configuration consists of ten 3TB SATA disks. The total cost of our setup amounted to about $10,000. Section 7.2 illustrates that Chronicle can support network rates higher than 10Gb/s with a much less powerful hardware configuration. The server runs on a 3.2.32 Linux kernel with a patched ixgbe driver to support netmap. The NFS server was a NetApp® FAS6280 with two 10GbE NICs.

### 7.1 Libtask Evaluation

We used two microbenchmarks to measure the performance of Libtask against similar frameworks in Erlang (version R15B01) and Go (version 1.0.2). These evaluations also compare the performance of the NUMA-aware and NUMA-agnostic versions of Libtask. In the *Message Ring* benchmark, 1,000 Processes form a ring and pass approximately 100 million Messages around the ring, so that there are 100 outstanding Messages within the ring at any given time. In the *All-to-All* benchmark, 100 Processes send approximately 100 million Messages to each other in a random way.

Figure 4 presents the number of Messages exchanged per second for different implementations as the number of Scheduler threads varies. The results are for averages of 10 runs. For all configurations, the NUMA-aware Libtask performs the best, and both Libtask implementations outperform implementations in Erlang and Go, because Libtask is a much leaner messaging framework with none of the overhead associated with copying message data, running inside of a virtual machine, or activities like garbage collection. The drop between the 16- and 32-thread configurations for the NUMA-aware Libtask is a result of cross-socket communication. Although



(a) Message Ring benchmark

(b) All-to-All benchmark

Figure 4: Libtask evaluation.

these benchmarks do not reflect CPU-intensive tasks performed by Chronicle, they are indicative of the rate at which Libtask can distribute tasks across the cores.

To test Libtask under a more realistic setup where competing load is present, we ran one CPU-intensive thread in the background for the "+*load*" configurations of Figure 4. Because this thread is not pinned to any core, it only degrades the 32-thread setup for the NUMA-aware configurations. One interesting finding for the 32-thread setup is that NUMA-awareness degraded throughput for Message Ring. This is because for the NUMA-aware setup, at any given time one Scheduler was pinned to the same core as the competing thread. The interference resulted in a *convoy effect* for the Message Ring benchmark that hurt the overall throughput.

Another interesting finding is that for the 2-, 4-, and 8-thread NUMA-agnostic configurations, adding the extra load improved the throughput for both benchmarks. This effect is a direct consequence of the competing thread pushing a larger number of Schedulers to run on the same CPU, resulting in better cache locality for them.

The impact of the extra load suggests possible improvements to the NUMA-aware version of Libtask: (1) pinning a Scheduler to a CPU, not to a core, to alleviate the convoy effect; and (2) taking into account the communication patterns of Processes to reduce the cross-socket communication. A comprehensive analysis of these enhancements is left as future work.

## 7.2 Chronicle Evaluation

We chose to evaluate Chronicle using the DataSeries pipeline because it was more CPU- and disk-intensive than the analysis pipelines. Figure 7 shows the experiment setup. In this setup, a client machine was directly connected to an NFS server via two 10Gb/s links. The server running Chronicle received network traffic on both directions of the client-server links using two fiber taps. For all the experiments described in this section, we used two Chronicle pipelines (one pipeline per client-server link). Hereafter, for brevity, we use the term *core* when referring to logical cores.

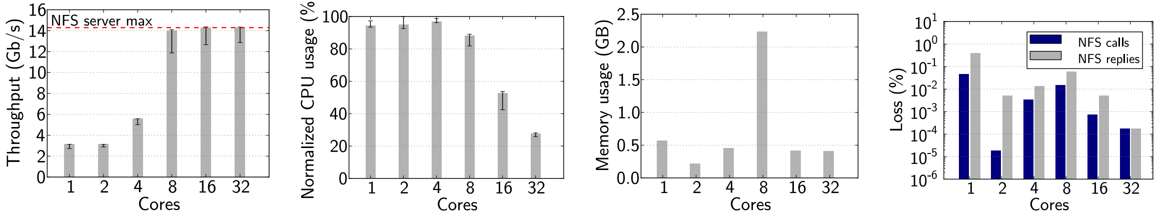
We experimented with many configurations to stress Chronicle. We observed that a mix of NFS read and write workloads resulted in the highest rates for both throughput and I/Os per second (IOPS) on the NFS server. The results presented in this section were all generated using 30-minute, constant-rate fio [4] workload runs. Interestingly, we obtained better results with NUMA-agnostic Libtask due to the convoy effect described in Section 7.1. The competing activities in the trace capture scenario were threads used by DataSeries for compression and I/O as well as applications like Apache that ran in the background. Therefore, we present results for this configuration only. This section measures the performance of Chronicle for a number of metrics, including multicore scalability, CPU and memory usage, and the success rate in capturing and parsing NFS operations.

### 7.2.1 Maximum Throughput

Figure 5(a) shows the maximum *sustained* throughput rates as we varied the number of cores used by Chronicle.[2] The sustained throughput rates are characterized by constant utilization of the buffer pool (Section 5.2). Therefore, Chronicle should handle these workload rates for an infinitely long duration. This also means that Chronicle can support higher data rates at the expense of higher buffer pool utilization, albeit for a bounded amount of time.

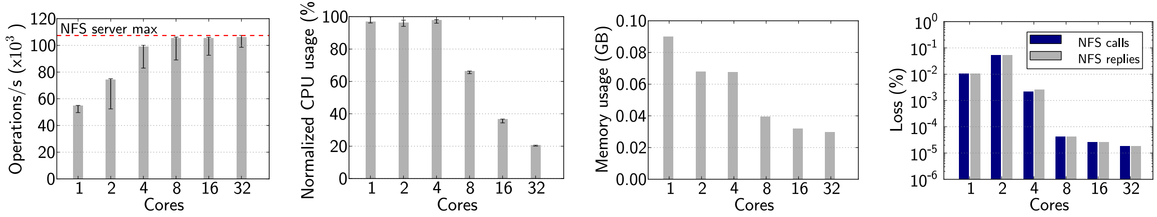
As shown in Figure 5(a), Chronicle with one core could support 3.05Gb/s. Adding a second core did not help much with throughput, although it did help with better coverage (Figure 5(d)). We suspect that polling the NICs by the four Packet Reader modules left little time for other modules. Near maximum CPU utilization for these configurations, shown in Figure 5(b), illustrates this point. However, for the 4- and 8-core configurations,

[2] The bars in figures 5(a), 5(b), 6(a), and 6(b) denote the average values, and the error bars show the minimum and maximum.



(a) Max. sustained throughput (b) CPU utilization (c) Max. buffer pool utilization (d) NFS capture loss rate

Figure 5: Maximum sustained throughput evaluation.



(a) Max. sustained IOPS (b) CPU utilization (c) Max. buffer pool utilization (d) NFS capture loss rate

Figure 6: Maximum sustained IOPS evaluation.

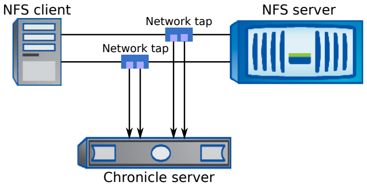


Figure 7: Experiment setup.

there were enough spare CPU cycles to sustain 5.43Gb/s and the near maximum 13.68Gb/s respectively. Adding an extra thread per core or the second CPU (i.e., the 16- and 32-core configurations) did not significantly increase the maximum sustained throughput because with 8 cores we could almost handle the maximum rate supported by the NFS server.

Figure 5(c) shows the highest usage of the buffer pool to handle the maximum throughput configurations. Although our application scenario is mostly concerned with high throughput and not processing latency (except in reading packets), the relatively low buffer utilization suggests that Chronicle processed packets very quickly. It is worth noting that the 16- and 32-core configurations had considerably less memory utilization than the 8-core configuration, because with more computational resources Chronicle could process packets much faster.

Another important metric is the loss rate in capturing NFS operations. These losses can happen either as a result of Packet Readers not getting scheduled fast enough to empty the NIC ring buffers or as a result of capture via lossy methods (e.g., port mirroring). We compared the number of NFS operations seen by the NFS server with the number of operations captured in the DataSeries traces to measure Chronicle's loss rate. For all configurations in Figure 5(d), Chronicle had a negligible loss rate. Most notably, for the 32-core configuration at 14.0Gb/s, Chronicle missed only 84 out of the total 48,600,042 NFS operations. An interesting conclusion we can draw from the results in Figure 5 is that a hardware configuration with 1GB of RAM dedicated for Chronicle, and an 8-core CPU with hyper-threading enabled, should handle 14Gb/s relatively loss-free, provided that there is a high-quality data feed (Section 7.2.3).

### 7.2.2 Maximum IOPS

The goal of the experiments described in this section was to stress Chronicle with an increasingly higher number of NFS operations until Chronicle reached its limit and could no longer keep up. For the experiments described in Section 7.2.1, the NFS client issued 64KB read and write operations to maximize throughput. To maximize IOPS, the client issued 1B read and write operations. Figure 6(a) shows the maximum sustained IOPS Chronicle could handle for different numbers of cores. The results suggest that with 8 cores and only 40MB of buffer space, Chronicle could handle the maximum IOPS supported by the NFS server (106 kIOPS) relatively loss free. The CPU utilization for the 8-core setup also implies that only 5 out of 8 cores were fully utilized. Therefore, Chronicle could potentially support

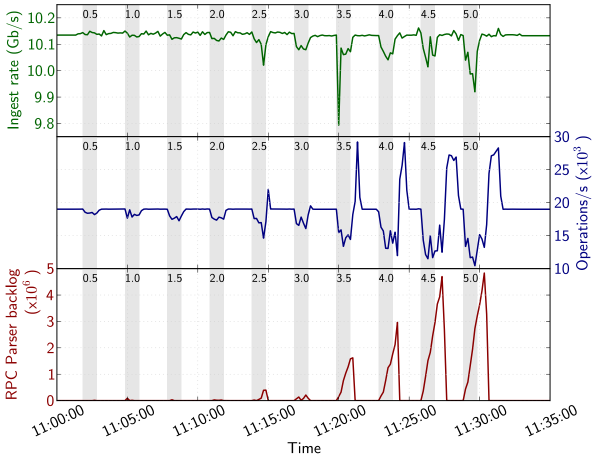


Figure 8: This figure illustrates a controlled experiment to study the impact of packet loss, when the network traffic rate is about 10Gb/s. The highlighted 1-minute intervals correspond to periods when packets got dropped at the rate of 0.5% to 5%. Although high loss rates caused significant backlog for RPC Parser, Chronicle performed well under normal network conditions and recovered quickly when the losses were intermittent.

much higher IOPS rates. In fact, when we traced a customer's metadata-intensive workload, which was generated by more than 3,000 clients, we saw that Chronicle could sustain 150 kIOPS.

The maximum sustained IOPS results illustrate an important point about Chronicle. Chronicle with 1 core can support twice as many small NFS operations as the 32-core setup of Section 7.2.1 (55,000 vs. 27,000 operations/s). Clearly, the cost of processing small PDUs is much less than that of processing large PDUs. Through CPU profiling and by examining the size of the Message queues for different modules, we have confirmed that when operating in the slow mode, RPC Parser consumes the most CPU cycles among all the modules. Recalling the discussion in Section 5.1.1, RPC Parser has to scan packet payloads in order to find the next RPC header while operating in the slow mode. When PDUs are small, it scans relatively few bytes before getting back to the fast mode. However, for large PDUs the module may potentially scan 64KB or more before it can find a header. Therefore, unlike in packet forwarding, where a high volume of small packets poses the largest overhead, a high volume of out-of-order packets belonging to large PDUs poses the biggest challenge to Chronicle.

### 7.2.3 The Impact of Packet Loss

The previous section discussed how packet loss can degrade Chronicle's performance. This section describes a controlled experiment to study this effect. For this experiment, we used the 32-core setup of Section 7.2.1 but made two changes. We limited the network traffic rate to about 10Gb/s, and we modified the Packet Readers to uniformly drop data packets (i.e., packets that are not empty acks) at specific time intervals. The 1-minute time intervals during which the Packet Readers induced packet loss are highlighted in Figure 8 and are annotated with the loss rates. The packet loss rates ranged from 0.5% to 5% and were interspersed with 2-minute intervals when there were no induced losses.

The top graph in Figure 8 shows the effective network traffic rate ingested by Chronicle during the course of the experiment. The middle graph illustrates the number of NFS operations that were processed by Chronicle. The dips in the graph correspond to the lower number of complete PDUs that Chronicle managed to find during the loss intervals. As loss rates increased, the dips became deeper and wider. They became deeper because there were fewer complete PDUs to be processed and they became wider because the RPC Parsers stayed in the slow mode longer (even beyond the 1-minute loss interval). However, as soon as Chronicle processed all the packets received during the loss intervals, it reverted to the fast mode and very quickly made up the lost ground. The spikes following the dips signify this behavior.

One metric that clearly captures the behavior of Chronicle under packet loss is the size of the Message backlog for the RPC Parsers (the bottom graph in Figure 8). Because an RPC Parser spends more time in the slow mode, the number of outstanding Messages in its queue grows. Although the backlog was negligible when packet loss was 2% or less, it grew very fast at higher rates. Because each Message in an RPC Parser's queue corresponds to one packet, the backlog had a direct impact on increased buffer pool utilization. The results in Figure 8 suggest that Chronicle can handle packet loss at low rates fairly well provided that the losses are intermittent and that there is a buffer pool of sufficient size to accommodate the additional processing of the out-of-order packets.

### 7.2.4 Trace Compression Ratio

Unsurprisingly, the size of a trace generated by Chronicle depends on the workload being captured. This section briefly discusses a 7-hour-long trace, captured from a production environment, to shed some light on the advantages of inline parsing, storing the checksums of read and write data, and inline compression over storing the raw network data, as was done in the previous efforts. For this trace, Chronicle processed 1.8TB of network traffic where 36% of the operations corresponded to NFS reads and writes. The total trace size generated

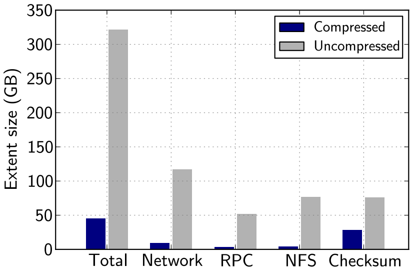


Figure 9: The size of different extents with and without compression.

by Chronicle amounted to 44.6GB, which is a 40x reduction over saving raw packets. The extents corresponding to the network header and data checksums amounted to 84% of the trace, while the extents storing the RPC and NFS fields accounted for the rest. The trace compression ratio varied from extent to extent. For instance, the extents storing the NFS, RPC, network, and checksum data had compression ratios of 20:1, 15:1, 12:1, and 3:1 respectively (Figure 9).

## 8 Lessons Learned

Our experience with Chronicle suggests that the actor programming model is an effective, programmer-friendly framework for workload characterization at line-rate. We believe that some of the techniques described in this paper have applicability beyond the NFS protocol. For instance, the fast- and slow-mode techniques to identify message boundaries (Section 5.1.1) have applicability to other network storage protocols such as iSCSI, SMB/CIFS, and the RESTful key-value store protocols. Similarly, the zero-copy application layer parsing technique (Section 5.2) has no limitations in supporting other protocols. The experiments described in sections 7.1 and 7.2 revealed that preserving cache locality should not come at the expense of balancing load across cores, particularly in the presence of competing load.

Chronicle has been deployed in a number of production environments to collect traces and perform sizing. For these deployments, the average traffic rates (3 to 6Gb/s) were lower than the results presented in this paper, and Chronicle could accurately capture the dynamics of the workload. One common theme among our deployments thus far has been the concentration of I/O by clients and by files. For instance, in a 3-day deployment there were 573 unique NFS clients, where the 25 most active clients accounted for more than 60% of read and write bytes served by the server. Accesses to files were also heavily concentrated. In a week-long deployment, the 25 hottest files, out of 9 million unique files accessed, accounted for 40% of total operations on the server. The insights facilitated by Chronicle can guide a storage administrator or a software-defined storage controller to dynamically tune a storage system. As an example, the knowledge of hot files and their access patterns can lead to better data caching and tiering solutions.

One powerful aspect of Chronicle is that it enables detection of problematic scenarios that are often not foreseen. For instance, we noticed that in a production environment, 8 clients out of more than 3,000 unique clients were reissuing read operations at the aggregate rate of 40 kIOPS for days. A closer examination revealed that these reads accounted for 31% of all the operations received by the server and that they were all failing due to a stale file handle!

Identifying misconfigurations is another application scenario for Chronicle. During one deployment, we observed that a server was serving *getattr* requests at the average rate of 56 kIOPS. Further analysis of the top 25 client-file pairs that were present in the getattr requests revealed that these requests were targeted at static files, with many being Linux system utilities that rarely get updated. Shockingly, there were on average 214 getattr requests per second for the top client-file pair! With insights from Chronicle traces, we were able to recommend configuring the NFS clients with correct attribute caching parameters to eliminate a sizable portion of unnecessary getattr requests. Another interesting finding was that for some clients more than 80% of read and write operations did not fall on 4096-byte boundaries. These misaligned I/Os are generally more expensive to serve by a block-based storage system and can be the result of nonbuffered I/Os at clients or incorrectly configured virtual disks for virtual machines.

## 9 Conclusions

This paper presented the design and implementation of Chronicle, an extremely flexible framework for characterizing workloads at line rate. We demonstrated that it is possible to capture and analyze NFS traffic at 14.0Gb/s using general-purpose CPUs, disks, and NICs. Chronicle's high-throughput architecture is facilitated by a pluggable, pipelined design that is based on actor programming model. Such a design enables seamless scalability to many cores where CPU-intensive operations such as stateful parsing, pattern matching, data checksumming, and inline compression can be done inline.

Chronicle's source code [2] is available under an academic, noncommercial license.

## 10 Acknowledgements

We would like to thank our shepherd, Dean Hildebrand, and anonymous reviewers for their helpful comments.

## References

[1] The Bro network security monitor. https://www.bro.org.

[2] Chronicle. https://github.com/NTAP/chronicle.

[3] DPDK: Data Plane Development Kit. http://dpdk.org.

[4] fio. http://freecode.com/projects/fio.

[5] NetFlow. http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html.

[6] AGHA, G. A. Actors: A model of concurrent computation in distributed systems. Tech. Rep. 844, Massachusetts Institute of Technology, 1985.

[7] ALDINUCCI, M., DANELUTTO, M., KILPATRICK, P., AND TORQUATI, M. *FastFlow: high-level and efficient streaming on multi-core*. Parallel and Distributed Computing, Chapter 13. Wiley, 2013.

[8] ANDERSON, E. Capture, conversion, and analysis of an intense NFS workload. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies* (2009).

[9] ANDERSON, E., ARLITT, M., MORREY, C. B., AND VEITCH, A. DataSeries: an efficient, flexible data format for structured serial data. *ACM SIGOPS Operating Systems Review 43*, 1 (2009).

[10] BOLLA, R., AND BRUSCHI, R. PC-based software routers: High performance and application service support. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow* (2008).

[11] CHARIKAR, M., CHEN, K., AND FARACH-COLTON, M. Finding frequent items in data streams. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming* (2002).

[12] CHEN, B., AND MORRIS, R. Flexible control of parallelism in a multiprocessor PC router. In *Proceedings of the USENIX Annual Technical Conference* (2001).

[13] CONG, W., MORRIS, J., AND XIAOJUN, W. High performance deep packet inspection on multi-core platform. In *Proceedings of the 2nd IEEE International Conference on Broadband Network & Multimedia Technology* (2009).

[14] DESENSI, D. DPI over commodity hardware: implementation of a scalable framework using FastFlow. Master's thesis, University of Pisa and Scuola Superiore SantAnna, 2013.

[15] DOBRESCU, M., ARGYRAKI, K., IANNACCONE, G., MANESH, M., AND RATNASAMY, S. Controlling parallelism in a multicore software router. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow* (2010).

[16] DOBRESCU, M., EGI, N., ARGYRAKI, K., CHUN, B., FALL, K., IANNACCONE, G., KNIES, A., MANESH, M., AND RATNASAMY, S. RouteBricks: exploiting parallelism to scale software routers. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles* (2009).

[17] EGI, N., GREENHALGH, A., HANDLEY, M., HOERDT, M., HUICI, F., MATHY, L., AND PAPADIMITRIOU, P. Forwarding path architectures for multicore software routers. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow* (2010).

[18] ELLARD, D., LEDLIE, J., MALKANI, P., AND SELTZER, M. Passive NFS tracing of email and research workloads. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (2003).

[19] FUSCO, F., AND DERI, L. High speed network traffic analysis with commodity multi-core systems. In *Proceedings of the 10th ACM SIGCOMM Internet Measurement Conference* (2010).

[20] GIBBONS, P. B., AND MATIAS, Y. Synopsis data structures for massive data sets. In *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms* (1999).

[21] HAN, S., JANG, K., PARK, K., AND MOON, S. PacketShader: a GPU-accelerated software router. *ACM SIGCOMM Computer Communication Review 40*, 4 (2010).

[22] HEWITT, C. Viewing control structures as patterns of passing messages. *Artificial Intelligence 20*, 3 (1976).

[23] HOARE, C. A. R. Communicating sequential processes. *Communications of the ACM 21*, 8 (1978).

[24] KIM, J., HUH, S., JANG, K., PARK, K., AND MOON, S. The power of batching in the Click modular router. In *Proceedings of the 3rd ACM Asia-Pacific Conference on Systems* (2012).

[25] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click modular router. *ACM Transactions on Computer Systems 18*, 3 (2000).

[26] LEUNG, A. W., PASUPATHY, S., GOODSON, G., AND MILLER, E. L. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the USENIX Annual Technical Conference* (2008).

[27] MARIAN, T., LEE, K. S., AND WEATHERSPOON, H. NetSlices: scalable multi-core packet processing in user-space. In *Proceedings of the 8th ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (2012).

[28] MEYER, D. T., AND BOLOSKY, W. J. A study of practical deduplication. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies* (2011).

[29] NELMS, T., AND AHAMAD, M. Packet scheduling for deep packet inspection on multi-core architectures. In *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (2010).

[30] PFAFF, B., PETTIT, J., KOPONEN, T., AMIDON, K., CASADO, M., AND SHENKER, S. Extending networking into the virtualization layer. In *Proceedings of ACM Workshop on Hot Topics in Networks* (2009).

[31] RIZZO, L. Netmap: a novel framework for fast packet I/O. In *Proceedings of the USENIX Annual Technical Conference* (2012).

[32] RIZZO, L., CARBONE, M., AND CATALLI, G. Transparent acceleration of software packet forwarding using netmap. In *Proceedings of the 31st IEEE International Conference on Computer Communications (IEEE INFOCOM)* (2012).

[33] VASILIADIS, G., POLYCHRONAKIS, M., AND IOANNIDIS, S. MIDeA: a multi-parallel intrusion detection architecture. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (2011).

[34] WANG, J., CHENG, H., HUA, B., AND TANG, X. Practice of parallelizing network applications on multi-core architectures. In *Proceedings of the 23rd International Conference on Supercomputing* (2009).

[35] YU, M., JOSE, L., AND MIAO, R. Software defined traffic measurement with OpenSketch. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (2013).

[36] ZIMMERMANN, H. OSI reference model - The ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications 28*, 4 (1980).

NetApp, the NetApp logo, and Go further, faster are trademarks or registered trademarks of NetApp, Inc. in the United States and/or other countries.