# Full Packet Capture and Offline Analysis on 1 and 10 Gb/s Networks

E. Anderson, M. Arlitt
HP Laboratories Palo Alto
HPL-2006-156
November 1, 2006*

full packet capture, 10 Gb/s network, driverdump

This paper describes our experiences with implementing and using a network monitor built with commodity hardware and open source software to collect contiguous, multi-day, full packet traces from 1 and 10 Gb/s networks. The length of the traces is primarily limited by the capacity of the disks attached to the monitor, and the rate and size of packets on the network. On a 10 Gb/s enterprise network our monitor sustained packet capture rates of 160,000 pps (packets per second) and data capture rates of 0.7 Gb/s, and burst capture rates up to 550,000 pps and 3.7 Gb/s respectively (with minimal packet loss). In testing we have achieved sustained capture rates of up to 676,000 pps and 1.4 Gb/s. We found that our technique (*driverdump*) can sustain capture rates between 1.86x (large packets) and 5.98x (small packets) higher than the traditional *tcpdump* program; compared to the linux-specific *lindump* program, we achieve rates 1.48x (large packets) and 2.25x (small packets) higher. We describe the current bottlenecks with our monitor and elaborate on how to address them. We also discuss our tools and techniques for efficiently analyzing the multiterabyte traces we collected. In particular, we rely on DataSeries, a highly efficient trace storage format.

* Internal Accession Date Only
Approved for External Publication

# Full Packet Capture and Offline Analysis on 1 and 10 Gb/s Networks

E. Anderson and M. Arlitt
*HP Labs*
*Palo Alto, CA*

## Abstract

This paper describes our experiences with implementing and using a network monitor built with commodity hardware and open source software to collect contiguous, multi-day, full packet traces from 1 and 10 Gb/s networks. The length of the traces is primarily limited by the capacity of the disks attached to the monitor, and the rate and size of packets on the network. On a 10 Gb/s enterprise network our monitor sustained packet capture rates of 160,000 pps (packets per second) and data capture rates of 0.7 Gb/s, and burst capture rates up to 550,000 pps and 3.7 Gb/s respectively (with minimal packet loss). In testing we have achieved sustained capture rates of up to 676,000 pps and 1.4 Gb/s. We found that our technique (`driverdump`) can sustain capture rates between 1.86x (large packets) and 5.98x (small packets) higher than the traditional `tcpdump` program; compared to the linux-specific `lindump` program, we achieve rates 1.48x (large packets) and 2.25x (small packets) higher. We describe the current bottlenecks with our monitor and elaborate on how to address them. We also discuss our tools and techniques for efficiently analyzing the multi-terabyte traces we collected. In particular, we rely on DataSeries, a highly efficient trace storage format.

## 1 Introduction

In the summer of 2003 we were involved in a project at HP Labs to provide a remote computing service, and had to determine the feasibility of offering such a service. Since we would have to purchase millions of dollars of equipment to support the service, we needed to be confident in our assessment. As a result, we decided to collect packet traces from a customer's IT environment and analyze the traces. The analysis determined what effect reduced network bandwidth between file servers and remote compute nodes had on the throughput of a remote computing cluster. The challenge at the time was in collecting contiguous, multi-day, full-packet traces from the customer's 2 Gb/s (trunked 1 Gb/s) network. In subsequent studies the challenge became even greater as we needed to collect and analyze traces from 10 Gb/s networks, to understand the effects of both reduced network bandwidth and increased latency.

In a remote computing cluster, such as for animation rendering, computer aided engineering, and seismic analysis (e.g., reservoir simulation), jobs run in the computing cluster, read data from fileservers, and write results back out to the fileservers. Customers usually prefer an integrated solution wherein they only have to have one set of fileservers over a separated solution where they have to manually copy or partition their data. Therefore, the remote compute nodes will access the customers fileservers over a WAN. However, the increased latency and reduced bandwidth of the WAN connection could limit the performance of the remote compute nodes.

In our case, to assess the potential WAN bottleneck, we traced a collection of the customers nodes that were connected to their fileservers by two 1 Gb/s links. The customer was using NFS version 2, over UDP, but since we believed we might have to examine file data, we needed to capture all of the packet contents. This design decision was valuable, when we later had to analyze NFS version 3 over TCP traffic at a second customer site because when NFS runs over TCP, multiple requests and replies can be sent in a single IP datagram, requiring us to capture the entire packet. In order to validate that there would not be a WAN bottleneck at any time of the data, we wanted to capture multiple days of contiguous activity as the customer had indicated there was substantial daily cycles.

As network bandwidth increases, it becomes more difficult to monitor the traffic, and especially problematic to record all of the packets. In some cases, tradeoffs can be made in order to cope with the potentially larger volumes of packets and data traversing the link. For example, only selected packets or selected portions of the packets (e.g.,

TCP/IP headers) could be recorded. This reduces the volume of data that must be recorded, but at the cost of discarding data that may be needed. An alternative and complementary approach is to record only higher level information, such as characteristics of flows or connections. This approach can retain exactly the information desired, but requires analyses that are fast enough to run online. We felt that neither of these tradeoffs were acceptable for our situation, which left full-packet traces as our only viable option.

This paper provides several important contributions. First, it describes our experiences with designing, implementing, validating, and utilizing a network monitor for capturing contiguous, multi-day, full packet traces from both 1 Gb/s and 10 Gb/s networks. Second, it discusses the tools and techniques we used to efficiently analyze all of the collected data. Finally, the paper provides another example of how network monitoring can be used, outside of traditional applications.

The remainder of this paper is organized as follows. Section 2 examines related work. Section 3 discusses our methodology. Section 4 presents the design and implementation of our network monitor. Section 5 describes how we used and validated our monitor. Section 6 explains how we converted the raw data to a format more conducive to (timely) analysis. Section 7 discusses the types of analyses we performed, and illustrates the benefits of the conversion step. Section 8 highlights some of the learnings from this work. Section 9 concludes the paper with a summary of our work and a list of future directions.

## 2 Related Work

There are a number of companies that manufacture specialized network analyzer hardware (e.g., [18]). We did not choose this option for several reasons. First, these devices tend to be very expensive. Second, these devices are intended primarily for on-line analysis. This requires not only that the analyses be known and implemented in advance, but also that they be simple enough to run on the analyzer at line speed. In addition, on-line analysis does not permit "what if" scenarios to be explored on the same data set.

A more cost effective approach is to use a network processor-based tool in a commodity server platform to capture the network traffic. For example, Endace produces DAG (Data Acquisition and Generation) cards for capturing packet headers on high speed networks such as 1 and 10 Gb/s links [2]. These cards have been used by NLANR (National Laboratory for Applied Network Research) to capture contiguous packet header traces from 10 Gb/s networks [20]. Degioanni and Varenni modify the DAG card driver to improve the scalability of the card by distributing load across multiple CPUs [4]. However, we have not seen these types of devices used to capture either (complete) application level headers or complete packets for any duration of time.

The simplest and most cost effective approach is to run a software tool on a commodity server with off-the-shelf NICs (Network Interface Cards) to capture and record the network traffic. The obvious disadvantage of this approach is a software solution is less efficient than a hardware-based one. We initially tried `tcpdump` [24], although we quickly learned (as anecdotal information had suggested) that it could not provide the desired functionality at the network speeds we needed to work with. We found an alternative software tool called `lindump` [9], which we used initially. We elaborate on its strengths and weaknesses in Section 4.

An alternative approach is to use multiple computers to do the collection, and reduce the volume of data that eventually gets written to disk. This is the approach used by NG-MON [7]. It uses several computers to split the incoming traffic, and in stages reconstructs flows and determines a variety of statistics about each. This approach was not desirable in our case. First, we wanted as simple a solution as possible. Since the monitor was to be installed temporarily in a customer's environment, we needed it to be quick to install and configure. In addition, using multiple servers to capture the data requires accurate synchronization of the system clocks; although feasible, this is another issue to address. Finally, we preferred to do off-line analyses, so we could run more sophisticated analyses.

There are numerous challenges in collecting full packet traces at high speeds. Writing to disk is one potential bottleneck. Moore *et al.* attempt to avoid an I/O bottleneck by performing some on-line processing to reduce the volume of data that needs to be written to disk [17]. However, as Moore *et al.* observed, this can shift the bottleneck to other resources, such as the CPU. Similarly, high packet rates are an issue, as a CPU bottleneck can result from handling the individual packets. Viken and Heegaard also examine some other potential bottlenecks when operating at 1 Gb/s and higher [27]. In Section 4 we describe how we addressed such issues.

There are other groups working or performing high speed network monitoring, including Lobster [11], lambdaMon [15], Sprint IPMon [22], and AT&T Gigascope [8]. While all of these efforts have collected data from high speed networks, to the best of our knowledge, none of them have attempted continuous collection of full packets at high speed.

# 3 Methodology

In this section we describe the challenges we faced, and our solution to capturing and analyzing the traffic on both 1 Gb/s and 10 Gb/s enterprise networks.

## 3.1 Challenges

As we discussed in Section 1, as part of a feasibility study for providing a remote computing service we wanted to analyze a customer's actual network traffic to determine whether or not it was technically possible to keep a 1,000 CPU cluster busy via a wide-area network link to the customer's file servers. We believed there were two options available to us: analyze the network online, or capture traces of the network activity and perform the analysis offline. Each approach has its own advantages and disadvantages. In the first case, we avoid the need to capture voluminous raw data to disk. This avoids two potential problems: the technical challenge of writing large amounts of data to disk for an extended period of time, and the pragmatic issue of protecting the sensitive information within the trace. However, developing, testing and/or enhancing the analysis tools without any data from a "real" network is extremely difficult; many unexpected quirks occur in real traffic that are not anticipated, which will either cause the analysis tools to fail, or worse, to return inaccurate results. Furthermore, the complexity of the analysis that can be performed is limited by the capacity of the server monitoring the network. In the second case (capture traces and analyze offline), these challenges are inverted.

A related issue that also affected our decision was the anticipated utilization of the network link. A 1 Gb/s full duplex network has a theoretical capacity of 2 Gb/s. In our situation, the trunked network had a theoretical capacity of 4 Gb/s. If a such a network were heavily utilized, it would be difficult to sustain full packet tracing without incurring substantial packet loss, particularly with the commodity hardware that existed in 2003. However, it is common knowledge that many network links are not typically used at their full capacity all of the time, particularly not in both directions simultaneously. Thus, after some examination of the actual traffic in the initial environment, we believed that if we could develop a monitor that could handle sustained rates of several hundred Mb/s and burst rates near 1 Gb/s, that our system would perform satisfactorily in most situations that we would encounter. As a result of this conclusion, we decided to adopt the "capture-to-disk" approach.

A final issue is (passive) visibility into a network. One approach (*port mirroring*) is to have a switch forward a copy of all packets to the monitor. An alternative is to place a specialized device in the network (called a *tap*). Both approaches have advantages and disadvantages. We used port mirroring exclusively in our work, but our monitor will work with either.

The complete methodology that we adopted involved three stages. We elaborate on each stage in the following sections.

## 3.2 Data Collection

Due to the speed of the network, we expected disk bandwidth could be a bottleneck. We considered several options to address this concern: do not write all packets to disk; do not write full packets to disk; compress packets before writing to disk; or some combination of these. Since the traffic on the monitored link was (practically speaking) exclusively NFS traffic, we felt that the first option would not be particularly useful. Although sampling the NFS traffic would have reduced the number of packets written to disk, we would have wanted a sampling of the NFS transactions and not a sampling of the packets.

Similarly, we felt the second option could have helped, but we decided against it for several reasons. First, we would have needed to develop a tool for the online extraction of the NFS headers. Without traces of actual NFS traffic to assist with the development and testing of such a tool, this would be challenging. We found this problem when we converted our captured traces, but were able to fix our tools and just re-run the conversion. Second, we thought we might need more than just NFS headers for our analyses because we might have to examine related-data compression techniques, such as data-dependent chunking [14]. Third, some NFS replies, such as directory reads, have important information throughout the packet. As a result, our only viable option was to compress the packets before writing them to disk. We elaborate on this in Section 4.

## 3.3 Data Conversion

In our environment, we needed to remove all sensitive information from the trace (e.g., the customer's proprietary files). First, we extracted all of the IP, TCP and NFS header information that we wanted, and discarded all remaining data. This step removed sensitive data and substantially reduced the size of the resulting trace. Second, during the extraction of the IP, TCP and NFS information, sensitive information such as file names were encrypted. Encryption allowed us to identify identical files, hide sensitive actual names, and reconstruct filenames if needed during discussions with the customer.

The raw packet traces are written in the standard `pcap` format [24], making them compatible for use with existing public domain tools such as `tcpdump` [24],

`ethereal` [6], and `tcptrace` [25]. While we could write the preprocessed traces into `pcap` format as well, we chose to instead write them into DataSeries [3]. Since we intended to perform numerous specialized analyses, we felt that the benefits of DataSeries (described in Section 3.3.1) outweighed the advantages of using the `pcap` format for the final version of the traces and existing public domain tools for performing the analyses. We did, however, utilize publicly available tools to assist in the QA testing of our conversion tools.

#### 3.3.1 DataSeries

DataSeries is an open source, generic trace format developed at HP Labs [3]. It provides streaming access to database-like tables, and was developed specifically for handling large volumes of data. DataSeries was originally developed to generalize the block disk I/O trace specific format developed and used at HP Labs. DataSeries has since been used to capture process data, LSF scheduler information, `sar` data, e-mail messages for indexing, and as described in this paper, NFS traces. Since all of these trace types can be quite large, compression was integrated into DataSeries. At the same time, DataSeries provides the ability to quickly read selected fields within each trace. Finally, the DataSeries format is quite extensible, as its adaptation to formats other than block I/O traces has shown.

When data is stored in DataSeries, the following steps occur. Initially, the data is placed in an in-memory data structure called an *extent*. Each extent consists of two arrays: an array of fixed-size records (e.g., a structure for types such as integer and floating-point values), and an array of variable-sized data (e.g., strings). When the extent is ready to be written to disk, the data is packed (based on type definition), checksummed, compressed, checksummed again, and then written. The packing may perform data-specific filtering to improve the compression, such as storing the timestamp of one record relative to the timestamp in the previous record. Several different compression algorithms can be used, depending on the desired optimization. For example, to minimize storage space, the `bz2` algorithm [1] can be utilized; to minimize read time, the `lzo` algorithm [13] can be used; or to minimize compression time, the `lzf` algorithm [12] can be applied.

Reading an extent from disk involves verifying the checksum of the packed data, uncompressing the fixed and variable parts into memory, and verifying the checksum of the unpacked data. Besides the choice of compression algorithm, there are several techniques for reducing the time to retrieve values from a DataSeries file. First, indexing can be used to reduce the number of extents that have to be read. Second, the size and contents of each extent can be designed to improve the retrieval times. Despite the emphasis on data integrity checking, reading from DataSeries can be very fast. For example, on a mid-range server (e.g., a two-way 2.8 GHz Xeon server), simple scan-type queries can run as fast as the extents can be uncompressed (30-100 MB/s in our experience).

Another advantage of DataSeries is the common interface for developing analysis tools, regardless of the trace type. Our experience with using DataSeries with other trace types was another factor in our decision to use DataSeries as the final trace format, rather than a 'traditional' network trace format like `pcap`.

### 3.4 Data Analysis

As we alluded to in Section 3.3, we developed our own toolset for analyzing traces of NFS traffic. We started our conversion tools based on those used in [5], but we had to make substantial improvements to them to handle the significantly larger data rates at the customer's site. For example, Ellard reports that in their busiest 7 day trace, they captured 26.7 million operations; conversely, in one 6.3 day period, we captured $\sim$2.5 billion operations. Having roughly 100x more operations was one of the reasons that we used DataSeries rather than the text-based format used by their conversion and analysis tools. Finally, although existing public domain tools could have provided some of the simpler analyses, we were not aware of any tools that provided all of the analyses of interest, and we expected that initial analysis would lead to additional questions and analysis that we would need to develop.

## 4 Design and Implementation

In our initial feasibility study, we needed to capture full packets to disk from a 2 Gb/s (trunked 1 Gb/s links) network. Section 4.1 describes the hardware and software components of our original 1 Gb/s monitor. In a subsequent study, we required similar functionality for a 10 Gb/s network. The alterations to our network monitor are discussed in detail in Section 4.2.

### 4.1 Our 1 Gb/s Network Monitor

We chose an HP DL580 G2 rack-mount server as the platform for our network monitor. This was a state-of-the-art x86-based server at the time of our initial feasibility study (summer 2003). The high-performance system architecture of the DL580 G2 motivated our choice. This system uses the Broadcom ServerWorks Grand Champion HE chipset, which supports up to four Intel Xeon

CPUs on a 400 MHz front-side bus. This chipset supports up to 64 GB of DDR-200 SDRAM (12.8 Gb/s per channel), on a memory bus with (up to) 51.2 Gb/s bandwidth. This chipset includes three Inter Module Bus (IMB) I/O interface units, each with a maximum 12.8 Gb/s bandwidth. Connected to each IMB is a PCI-X bus which has a peak transfer rate of 8.5 Gb/s. Each PCI-X bus has two slots. Two LSI Logitech LS153c1030 Ultra320 SCSI controllers were used, each providing peak data rates of 1.28 Gb/s on each of two channels (four channels in total).

Our monitor was configured with four 2.5 GHz Intel Xeon CPUs, four 2 GB DDR-200 SDRAM memory modules, and an Intel Pro 1000 1 Gb/s copper-based network interface card. The Intel Pro NIC occupied a slot on one of the PCI-X buses.

Connected to the SCSI controller were one or two HP MSA30 disk trays. Each tray contained 14x147 GB 10k RPM SCSI drives; this provided a total of 2 or 4 TB of raw storage capacity. In total, our monitor consumed 10U of rack space; four for the server, three each for the disk trays. This is about one quarter of a standard two meter rack. We felt this was not too intrusive to host in a customer's environment.

We installed a Linux 2.4.24 kernel on our monitor. This was a current kernel version at the time of our initial study. The only tuning we did was to set large default receive socket buffers, to balance the interrupts for the different devices across the CPUs, and to disable swapping.

As we mentioned in Section 2, we initially tried to use `tcpdump` to capture network traffic to disk. We quickly learned (as anecdotal evidence suggested we would) that `tcpdump` dropped a substantial number of packets as the traffic rates increased. This caused us to look for alternative packet capture tools, and eventually led us to `lindump`. In initial testing, we found `lindump` dropped far fewer packets than `tcpdump` when capturing full packets to disk at relatively fast rates. Our measurements indicated that `lindump` used about $1/3$ of the CPU per packet that `tcpdump` used. We modified `lindump` to create a continuous series of trace files of roughly equivalent sizes. We used 200 MB as the approximate size for each file in the trace. We then modified `lindump` to use `mmap` to write each file directly to the `tmpfs` file system [26]. This change improved performance by avoiding either per packet writes to the output file or an additional copy to an intermediate buffer to perform larger writes. The files in `tmpfs` were then compressed before they were written to disk, enabling us to take advantage of all four CPUs on the system. In an attempt to further reduce the chance of an I/O bottleneck, the writes were scheduled in a round-robin fashion, the hypothesis being that a SCSI channel would be ready to use again by the time the other three channels had been written to. At any point in time, one file in `tmpfs` was being written to by `lindump`, and the remainder were being compressed or written to disk. The actual number of files in `tmpfs` varied according to the sustained and burst packet rates. We utilized 7 GB of memory for `tmpfs`; this could buffer almost one minute of sustained 1 Gb/s traffic. We had to disable swapping to prevent the contents of `tmpfs` from being swapped during traffic bursts (which caused memory pressure).

As mentioned, the ability to compress the traces before writing them to disk both decreases the I/O bandwidth used and increases the duration of the collection. Our compression process worked in the following manner. Once a completed trace file was written to `tmpfs`, if no other trace file was being compressed, the new trace file was compressed with `gzip -9` (best compression) and then written to disk. If another trace file was already being compressed, the new trace file was compressed using `gzip -1` (fastest compression). If two `gzip` processes were already running, the new trace file was just copied to disk. Up to four simultaneous copies were allowed at a time. The constants used in this process were experimental, and mainly an artifact of having four CPUs and four SCSI channels; other settings or approaches might work better.

By compressing the captured packets before they are written to disk, we increase the effective storage capacity of our monitor. Obviously the duration of a trace is still dependent on both the transfer rate and the achieved compression rate. If we assume a sustained transfer rate of 30 MB/s (240 Mb/s) and a 50% compression rate, then our monitor (with both disk trays attached) has sufficient capacity for 2.3 days of contiguous network activity, which would meet our goal.

## 4.2 Our 10 Gb/s Network Monitor

For a subsequent study, we attempted to re-use our 1 Gb/s network monitor. Unfortunately, when we analyzed the resulting traces, we discovered that there were many missing NFS replies in the traces. Analysis of the burstiness in the captured traces showed that we were measuring bursts above 300,000 pps (packets per second), and discussions with networking experts taught us that the switch used by the first customer had per-card buffering whereas the switch used by the second customer had per-port buffering. Although the overall rate on a 0.1 second granularity was similar between the two customers, the per-port buffers were too small to absorb the bursts, and so traffic was being dropped.

Therefore, we needed to upgrade our monitor to support 10 Gb/s networks. The obvious change we had to make was to replace the 1 Gb/s NIC with a 10 Gb/s NIC. The new NIC was an Intel Pro 10 Gb LR (alpha), a fiber-

based card. We also enabled NAPI, to address potential livelock issues [10]. Initially, we hoped that the only software change that would be needed was to replace the `e1000` driver with the `ixgb` driver. However, when we tested this solution, we discovered that `lindump` was not able to keep up with the higher burst rates, and so we were merely going to succeed in moving the drops from the switch to the host.

Thus, an alternative approach was needed. Our solution was to modify the `ixgb` driver to perform the packet capture (the code added to the driver disabled the normal packet reception for efficiency reasons, so a separate driver would be needed for normal connectivity). The packet capture now mostly bypassed the normal kernel network processing stack. Using this modified driver, we were able to capture packets at higher rates with much lower loss. In the same manner as with our 1 Gb/s monitor, the captured packets were written into equally sized files in the `tmpfs` file system. These files were then compressed and written to disk using the same process as before.

In our opinion, implementing packet capture as part of the device driver is a significant yet necessary departure from the traditional approach. In the following subsections we elaborate on the motivations to move packet capture to the driver, and describe our specific implementation. In the remainder of this paper we refer to this approach as `driverdump`.

#### 4.2.1 Why Capture in the Device Driver

Capturing the data in the device driver is inherently more efficient than capturing the data at user level. There are a number of reasons for this. First, we can eliminate a copy between the kernel and user level. The user level programs will receive the data either through a socket interface (`tcpdump`), or through an mmapped kernel buffer (`lindump`). However in either case the kernel must copy the data into the buffer or to user level, and the user level program has to copy the data from that buffer into the file. Capturing data directly in the device driver allows us to eliminate one of those copies and copy the data directly into the file.

Second we can eliminate some of the OS overhead. In the simplest case, we can avoid sending the packet up the network processing stack, thereby avoiding any locking in the stack before the packet is simply dropped. In a more advanced implementation, we can recycle the kernel network structures (`struct sk_buff` in Linux) immediately in the driver by re-initializing the structure, thereby avoiding any locking in the kernel memory allocation techniques. Further, since we know that only some of the fields have been affected by the driver, we only need to initialize those fields. Finally, we can even avoid adjusting the buffers at all and simply update the NIC data structures to tell the device that the packet is now available avoiding any PCI page mapping overhead.

The primary downside of eliminating the OS overhead is that now the NIC can only be used for packet capture, whereas in a tool like `tcpdump` or `lindump`, the NIC can still be used for normal network traffic. However since we have multiple NICs on most machines, this is a minor downside to increase performance. The downside of the additional steps of eliminating overhead described above is that each one is more invasive in the implementation. Since each optimization is strictly an improvement over the previous, we implemented the last one described above.

#### 4.2.2 Implementation Details

For our implementation, we have to make seven modifications to get the *driverdumping* (i.e., packet capture in the NIC driver) working. Three of those are to connect the `driverdump` code with the NIC driver, in particular including the header file, adding the `driverdump` structure to the NIC structure, and including the `driverdump` implementation in the NIC implementation. Four of the modifications hook into the driver. The first two hook into the initialization and cleanup so that we can initialize the `driverdump` structure and free the `driverdump` data when the module is unloaded. The third hooks into the driver's `ioctl` routine to allow the user-level program to control dumping. The fourth hooks into the packet reception code to call the `driverdump` reception code instead of the normal kernel path if driverdumping is enabled.

The majority of the `driverdump` implementation (about 1,000 lines of code, including comments) is driver-independent, and thus can be reused when porting to different drivers. In practice, the simplest implementation requires about 10 lines of (driver-specific) code be added to the driver. Eliminating all of the OS overhead (as described above), increases the number of changes. In the `ixgb` driver $\sim$100 lines of driver-specific code were added or changed to enable driverdumping. The most time consuming task was determining what modifications were needed to directly adjust the NIC data structures.

All control of dumping is handled through a single `ioctl`. The implementation has two main code-paths, the path that copies packets into the output files, and the path that sets up output files for writing. The portion that handles packet reception is relatively simple: it gets the current time, sets the capture and wire length values in the `pcap-header` structure, and copies that and the packet data into the output file. The one piece of complexity that occurs is that the file appears as dis-

contiguous pages, and on the i386 architecture not all of those pages can be mapped into the kernel address space at the same time. Therefore, the code for copying the data to the file has to handle the page boundary splits and the mapping and unmapping of pages.

The code that sets up the output files for writing is more complicated. It has to open the file specified by user level, set up all of the page data structures for the entire file, and initialize all of the buffer variables. It then marks the data structure as empty and available, and goes to sleep. Up to three buffers are used; an *empty* buffer available for use; an *active* buffer, to which packets are written; and a *filled* buffer, which is waiting to be removed and written to disk. The packet capture codepath will notice that there is an empty buffer, make it the active buffer, and begin copying packets into it. Once the active buffer is full, the capture code makes it the filled buffer and wake up the file handling thread. That thread will then remove itself from the filled variable, clean up the page structures, close the output file and return. If an empty buffer is not available, or the filled buffer is still present when the active buffer is full, then the driver will simply discard the packets.

The use of up to three buffers in the kernel allows us to avoid most drops as a result of running out of buffers. It also allows us to perform as much work as possible on another CPU, leaving only the mapping and unmapping of the file pages to the CPU actually processing packets.

## 5 Collection Results & Validation

In this section we examine the behavior of our network monitor. Section 5.1 provides empirical results from use in two customer deployments. Section 5.2 describes the tests we have conducted to quantify the performance of our monitor.

### 5.1 Empirical Results

Table 1 provides a high level summary of the empirical traces that we have collected. Overall, we have collected 1,556 hours (64.8 days) of traces on either 1 Gb/s or 10 Gb/s networks in live environments. A number of the initial traces we collected were quite short, as we were still testing the capabilities of the monitor. Some of the traces were collected on network links with lower utilization, which allowed us to collect reasonably long (full-packet) traces. The longest trace we collected was trace g in the 1 Gb/s environment; this trace is 286 hours in duration ($\sim$12 days), and contains 8.4 billion packets and 2 TB of data. Most of the traces on the 1 Gb/s network did not fill the disk trays. There are several reasons for this. In some cases we stopped the tracing to begin the conversion and analysis of that trace, or to to begin tracing a different link in the customer's environment. In others, the collection ended when the monitor was unable to handle a large burst of traffic.

| Network | Trace | Duration (h) | Packets (M) | Data (GB) |
|---|---|---|---|---|
| 1 Gb/s | a | 0.30 | 15.77 | 5.03 |
| | b | 0.12 | 13.63 | 4.37 |
| | c | 0.45 | 18.49 | 4.05 |
| | d | 0.28 | 13.17 | 3.08 |
| | e | 6.25 | 381.61 | 92.63 |
| | f | 45.48 | 2,644.63 | 631.95 |
| | g | 286.03 | 8,396.52 | 2,023.98 |
| | h | 137.78 | 4,851.95 | 1,169.95 |
| | i | 45.80 | 1,002.08 | 180.68 |
| | j | 23.07 | 613.75 | 106.40 |
| | k | 101.13 | 1,694.29 | 289.48 |
| | l | 73.48 | 460.17 | 322.59 |
| | m | 12.95 | 1,952.03 | 777.16 |
| | n | 117.87 | 5,628.07 | 1,857.26 |
| | o | 151.73 | 6,464.10 | 3,070.41 |
| | p | 6.58 | 952.69 | 304.69 |
| | q | 28.30 | 2,180.93 | 754.00 |
| | r | 34.67 | 3,381.71 | 880.18 |
| | s | 91.27 | 8,866.93 | 2,959.46 |
| | t | 26.32 | 2,998.53 | 876.12 |
| | u | 124.05 | 148.14 | 80.64 |
| | v | 39.60 | 1,297.96 | 752.57 |
| | w | 32.68 | 3,520.44 | 2,545,26 |
| | x | 20.33 | 1,884.48 | 1,317.43 |
| 10 Gb/s | A | 77.88 | 7,588.48 | 5,605,01 |
| | B | 75.37 | 7,031.86 | 4,929.39 |
| Total | | 1,555.78 | 74,002.42 | 31, 543.76 |

Table 1: Summary Trace Statistics

For the remainder of this section we select two traces from Table 1 to examine in more detail. From the 1 Gb/s network we use trace s. This trace is 91 hours in duration (3.8 days), and contains 8.9 billion packets and 2.9 TB of data. For the 10 Gb/s network, we use trace A, which is 78 hours in duration (3.25 days) and 5.6 TB in size. For the remainder of this paper, we refer to these two traces as the 1 Gb/s trace and the 10 Gb/s trace, respectively.

Before describing more detailed characteristics of these traces and their implications on the design of a high speed monitor, it is important to discuss the limitations of our approach (i.e., in using collected traces to discuss characteristics of the observed network). One potential problem is that our traces may not contain all of the packets that actually traversed the network link. This would cause both the packets per second and Mb/s rates to appear lower than they actually were. This problem could occur if either the switch mirroring the traffic or our monitor become overloaded and drop packets as a result. We determined that the driver dropped about 0.20% of the packets in the 1 Gb/s trace, and have lost the record of drops for the 10 Gb/s trace, however in both cases, we believe the aggregate loss rates between the driver and the switch are negligible, primarily because we were able to rebuild the NFS transactions in the traces. In preliminary tests on the 10 Gb/s network, we were not able to rebuild a substantial portion of the NFS transactions, which in-

formed us that our initial traces were missing a significant number of packets. This observation led us to develop packet-capture in the NIC driver, which remedied the problem we encountered. A second problem that can be encountered is skew in the timestamps that happens because the packets are timestamped when processed by the driver, not when they are received by the NIC. We can see this issue in the analysis results of the 1 Gb/s trace during periods of significant activity, and we expect that it occurred to some degree in the 10 Gb/s trace based on our measurements of the limits of the packet capture code. Skew can also be created in the network switch; if two packets come in on different links and leave on different links, they can transit a non-blocking switch at the same time. However, when the switch forwards the packets to the monitoring port, it will be forced to serialize them, thereby delaying one of the two packets. We discuss the timestamp issue in more detail below.

Figure 1 show the packets per second and Mb/s observed for the two enterprise traces. For the 1 Gb/s trace, the driver dropped about 0.20% of the packets, and we believe based on our analysis matching up request and response RPC messages that the switch dropped relatively few packets. Our record of drops for the 10 Gb/s trace have been lost, but based on our analysis matching up request and response RPC messages, we believe that this capture also experienced a negligible number of drops. Each graph shows the minimum, average, and maximum rate over a one hour period, using non-overlapping one minute samples. As expected, there is a clear daily trend in each data set. For the purpose of our feasibility studies, it was important to capture this, in order that we could more accurately assess whether a remote computing service could be kept busy, or if the network bandwidth or latency would become a bottleneck at any point in time (e.g., during sustained periods of activity).

Figure 1 reveals a number of interesting observations which we anticipated, and which affect the design of a monitor for providing full packet capture-to-disk functionality. First, there can be a substantial number of (small) packets per second; Figure 1(a) shows we observed more than 160,000 packets per second, sustained over a one minute interval. Second, although we expected the network would not be fully utilized, there are times when it is quite busy. For example, in both traces we see one minute intervals where the sustained rate is over 700 Mb/s. Both of these characteristics need to be considered in the design of the monitor.

Another characteristic of high speed networks that not only affects the design of the monitor, but is important to capture are the bursts of packets over short time scales. Figure 2 shows the observed data rates over a range of time scales. For the 1 Gb/s trace, Figure 2(b) shows that there were bursts well above the peak sustained rate. For example, when we examine the trace in 100 ms intervals, approximately 0.001% of the intervals had data rates of $\sim$ 982 Mb/s, essentially the maximum we could observe since we received the traffic over a half-duplex 1 Gb/s link from the switch performing the port mirroring. If we consider even shorter intervals (e.g., 1 or 10 ms), the graph suggests that larger bursts occurred, both in the number of packets per second (Figure 2(a)) and the bandwidth (Figure 2(b), but in fact these are a result of occasional delays in the processing that caused multiple packets to be timestamped in the operating system closer together in time than they actually arrived.

Figure 2(d) reveals that much larger bursts occurred over short time scales in the 10 Gb/s trace. For example, if we consider 1 ms intervals, Figure 2(d) shows that approximately 0.1% of all intervals were 2.3 Gb/s or above. With an interval length of 100 $\mu$s, 0.01% of all intervals had data rates of approximately 3.4 Gb/s. The largest burst we observed was nearly 3.8 Gb/s.

## 5.2 Test Results

Subsequent to our experiences with full packet capture-to-disk tracing on deployed 1 Gb/s and 10 Gb/s networks, we have made several changes to the software on our monitor. In particular, we upgraded the kernel to version 2.6.16.15. We made the change to 2.6 from 2.4 for several reasons: the 2.6 kernel is reported to have better I/O performance, and it automatically balances IRQs (which we previously had to do manually). An unexpected benefit of the upgrade was `oprofile` support. Using `oprofile` [21] we identified a performance problem in the original implementation; the kernel functions that map and unmap pages in the interrupt handler used more CPU time than we had expected. We found that over 40% of the CPU time was spent in those kernel functions. Once we optimized the copy implementation to only map pages when a new page was being accessed (and being careful to re-map if we had left the interrupt handler), only 1-3% of the CPU time was spent in those routines.

A few changes were also needed to the capture driver. First, we had changed NICs on our monitor, from the alpha version Intel NIC we had borrowed from one of our colleagues to a commercially released version of the Neterion XFrame I card. Therefore we had to switch to the `s2io` driver rather than the `ixgb` driver. One unfortunate side effect of moving to the `s2io` driver is that the interface between the driver and the card is more complicated than in the `ixgb` driver. As a result, we have not yet implemented the updating of the ring pointers to bypass all of the OS overhead, and are just re-initializing the part of the `sk_buff` structure that is affected by the driver. Second, several bugs were discovered that were

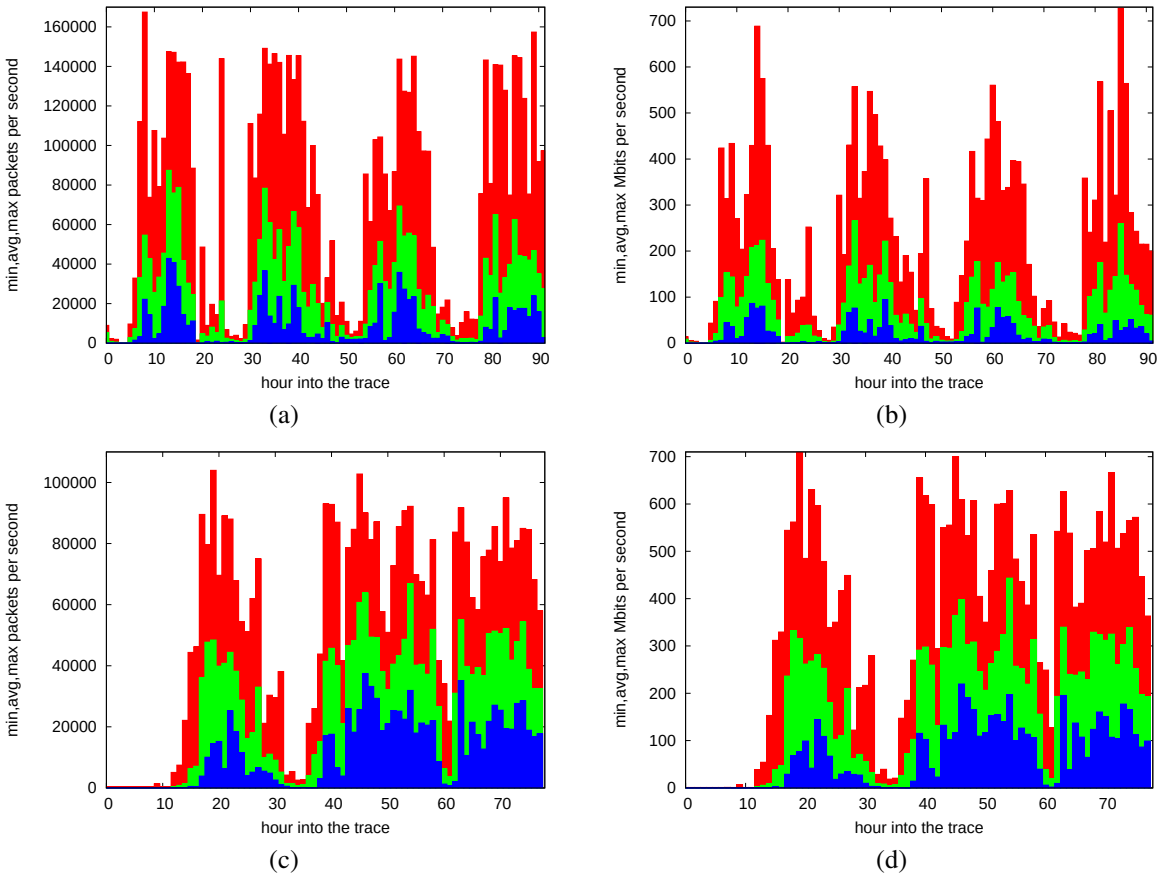



Figure 1: 1 Gb/s (a,b) and 10 Gb/s (c,d) trace results : (a,c) Packets per second; (b,d) Data transferred per second.

not triggered in the 2.4 kernel. Third, a significant part of the capture driver had to be rewritten as the interface to `tmpfs` in the kernel changed between 2.4 and 2.6.

We also modified the compression process. After the completion of our feasibility studies, we became aware of the `lzf` compression algorithm. We found that this algorithm was very fast, so we modified our compression process to use `lzf` in the cases where we had previously written uncompressed traces to disk (i.e., when there were already a `gzip -9` and a `gzip -1` process in progress).

Once all of these changes were complete, we performed a set of experiments to evaluate the performance of the monitor under controlled conditions, as well as to quantitatively compare the performance of our monitor to `tcpdump` and `lindump`. Section 5.2.1 describes our test environment. Section 5.2.2 discusses our experimental methodology. Section 5.2.3 summarizes our results.

### 5.2.1 Experimental Environment

Figure 3 shows the test bed we used for our experiments. We used six DL360 G3 servers, each with two 2.8 GHz Intel Xeon processors as the load generating clients. Each client had a dual port Broadcom 1 Gb/s network interface. Each client ran Linux (Debian stable with the 2.4-686 kernel), and used the `tg3` driver. The clients were connected to an HP Procurve 9304 switch. The clients generated UDP datagrams sent to a sixth DL360G3 server, that just responded to ARPs so that the load-generating clients could send it data; this server was also connected via a 1 Gb/s link; as a result, the switch dropped traffic to the server under load. Our monitor was connected to the switch by a 10 Gb/s link. The switch mirrored all of the traffic to or from any of the load generating clients to our monitor.

Each client ran a custom written perl script to generate the network traffic. The script filled each packet as half random, half zeros, in order that our monitor would achieve a realistic compression ratio (about 50%) on the resulting packet traces. In early testing we used packets filled with only zeros; for obvious reasons the resulting trace files were highly compressible as well as quick to compress. This skewed the performance results, and prompted the change in packet contents. Our script was also capable of generating traffic according to a schedule

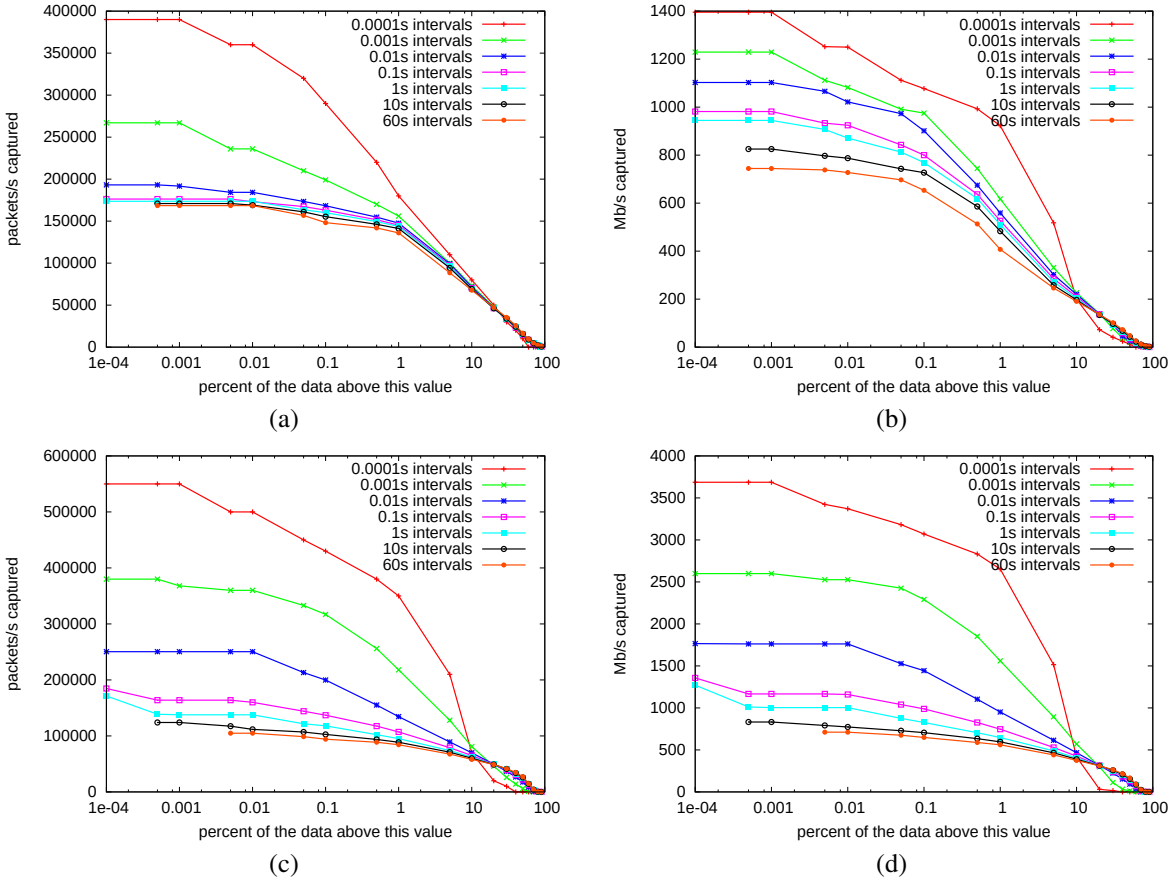



Figure 2: 1 Gb/s Trace (a,b) and 10 Gb/s (c,d) Observed Burstiness (a) packets per second; (b) Data transferred

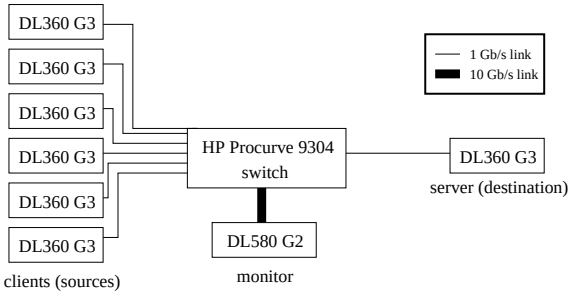



Figure 3: test environment

so that all six machines could burst for the same second, and then run at a slower rate otherwise. The machines synchronized according to their clocks, which were synchronized using NTP.

### 5.2.2 Experimental Methodology

Our experiments involve two different tests (small packet test, large packet test) and compare three different tools for capturing full packets to disk (`tcpdump`, `lindump` and `driverdump`).

For the small packet test, all six clients generated 64 byte packets as quickly as they could (∼150,000 pps each), which was sufficient to saturate all of the tools when capturing to disk. This test stresses the tool's ability to keep up with packet arrival events, and also emulates what would happen if we were only capturing packet headers. For the large packet test, we only used two clients; each generated 1514 byte packets as quickly as possible (∼1 Gb/s each), which was sufficient to saturate all of the tools when capturing to disk. This test stresses the compression and write to disk operations. With the tools under study, we attempt to capture as many of the packets as possible. The switch reports the number of packets it is mirroring, which we use to determine how many packets a particular tool is dropping.

In an attempt to provide as fair a comparison as possible, we try to use the tools in the same manner. In particular, since `lindump` and `driverdump` break up the stream of captured packets into multiple small (e.g., 200 MB) traces, we have `tcpdump` behave in a similar manner. With all tools the trace files are written to `tmpfs`, where our compression process then compresses them

| Tool | 64 byte packets | 1514 byte packets |
|---|---|---|
| tcpdump | 113,000 pps | 763 Mb/s |
| lindump | 300,000 pps | 957 Mb/s |
| driverdump | 676,000 pps | 1,430 Mb/s |

Table 2: comparison of capture-to-disk tools

and writes them to disk in the manner discussed in Section 4.1.

Finally we tested behavior under bursts with the driverdump tool. Each of six separate machines followed the same schedule of generating packets for 10 seconds, and idling for ten seconds. In the first *on* interval, three servers generated 1514 bytes packets as quickly as they could. (We could have used all six, but since driverdump could only sustain about 1.5 Gbps, there was no point). In the second *on* interval, all six servers generated 64 byte packets as quickly as they could. Finally in the third *on* interval the six servers each generated packets at 60,000 pps, with one server assigned to generating packets each of 64, 200, 576, 768, 1024, and 1514 byte packets.

#### 5.2.3 Experimental Results

Table 2 provides the average performance results we achieved for each tool. For the small packet test, tcpdump had (as expected) the poorest performance, averaging 113,000 packets per second over a 30 minute test. lindump averaged 300,000 packets per second, a $\sim$2.65x improvement over tcpdump. Our driver capture tool driverdump achieved a further 2.25x improvement, averaging 676,000 packets per second captured to disk. In the driverdump test we let the experiment run until the monitor ran out of disk space; this occurred after 50 hours and 23 minutes, during which time the tool captured almost 120 billion packets to disk.

One limiting factor in the tcpdump test is the driver path; only 385,000 pps on average are getting into the kernel. tcpdump itself then drops a significant number of the remaining packets. An interesting side effect of the reduced capture rate is that the CPU utilization required for compression is also reduced. We observed that gzip -1 is able to compress trace files faster than tcpdump can capture them, so our compression process never uses the lzf algorithm with tcpdump. We also observed that tcpdump is using 100% of two CPUs to capture 113,000 pps, whereas driverdump only used 100% on one CPU to capture at $\sim$ 6x faster. In the tcpdump case, one CPU is dedicated to running the thread that is copying packets from the driver to a user-level buffer, while the second CPU is completely occupied by tcpdump, copying the packets from the buffer to a file in tmpfs. With driverdump we avoid one copy which reduces the CPU utilization, as well as the memory bandwidth consumption.

In the tests with lindump, 100% of the CPU running the interrupt handler is used, and about 90% of the (second) CPU is used by the lindump process to capture at a rate of 300,000 pps. In this case the monitor was not able to compress all of the files, as there are not enough spare CPUs available.

We also examined how fast the OS itself could receive packets. We ran tcpdump in a mode to just drop all of the packets (selecting for a non-existent host), and found that the OS peaked at about 550,000 pps to just receive and discard the packets. Comparing this result to the driverdump performance illustrates how efficient our optimized capture code is; it is actually capturing and writing packets to disk faster than the standard OS network code could receive and discard the packets. This effect is achieved because the standard OS code needs to allocate, initialize and de-allocate memory for each packet, and in addition needs to perform a number of locks in order to transfer the packets up the networking stack so they can be discarded. Conversely, our dumping code avoids all of the allocation, de-allocation, and network stack locking. It also avoids most of the per-packet initialization costs. The cost of performing the copies for small packets is sufficiently less that these other costs that our driver can capture packets faster than the standard OS can drop them.

The large packet experiments indicate less of a difference between the tools. lindump is only 1.25x faster than tcpdump, and driverdump is only 1.49x faster than lindump. This is occurring because the capture is memory bandwidth limited. Measurements indicate that we only have about 6 Gb/s of copy bandwidth when performing small copies. Since driverdump is making at least two copies (one from the network to tmpfs, and one from tmpfs to disk) plus half a copy as the NIC writes the packet into memory, we can account for about 3.5 Gb/s of copy bandwidth being used. Since we are also using memory bandwidth to perform spin locks, page remapping, filesystem operations, process creation, etc., it is easy to see that the machine is becoming limited by its available memory bandwidth.

Another interesting effect which is penalizing driverdump is that as the capture rate increases, the fraction of the data that can be compressed is reduced. For tcpdump we used one gzip -1 process and up to two lzf compress processes. The two spare CPUs were enough to keep up with the compression rate. lindump was similar. When we tried to run driverdump with that configuration, we found that the compression was unable to keep up with the rate of data acquisition, and tmpfs overflowed. We instead had to move to using

up to two `lzf` compressions and up to 10 straight file copies to keep from overflowing the temporary staging area. Therefore `driverdump` was both capturing more data and using more memory bandwidth. If we ran `tcpdump` with `lzf` and copies, we reduced its capture performance by about 10%.

The other interesting difference was around the priority assigned to the thread polling the network interface. Linux implements both an interrupt-based and a polling based access to network devices, and dynamically chooses between them based on load to eliminate wasted network processing when the application is falling behind [10, 16]. For both `tcpdump` and `lindump`, leaving this thread at the lowest priority is the best choice because otherwise it performs a lot of useless work. Conversely for `driverdump`, the best choice is to make this thread highest priority because the code copying out to disk will adapt by using more copies and less compression. We accidentally left the priority of the thread as low for testing `driverdump` and observed that the compression processes were taking priority over the packet capture. However, making the polling thread high priority for either `lindump` or `tcpdump` caused the polling thread to overrun the user level process and waste time dropping packets.

Figure 4 shows the results of our burstiness experiments. We took a long trace of the data and calculated the average, minimum and maximum captured and offered rate at each point in the cycle. The offered rate was calculated by running driverdump in a "drop all" mode, and since the capture CPU remained below 100% busy for the entire time, we believe that the rates indicated are accurate. Figure 4 shows that under a bursty workload, `driverdump` is able to capture more than the sustained rate. In particular, the captured rate for 64 byte packets remains well above the sustained rate, although it is clearly slowing down over the burst interval. Similarly, on the bandwidth graph, the rate is above the sustained rate for a fraction of the measured period, although by the end of the burst it has slowed down to the sustained rate. The reason that `driverdump` slows down to the sustained rate for the larger packets by the end of the burst is because more data is being generated, so more compression and copying processes are started, and hence more interference occurs. Conversely the smaller packets only generate a few files to compress (at most 3 over the entire 10 second burst) and hence the compression does not interfere as much with the capture process. One possible implication of this result would be that the compression process should attempt to delay compressing the data under heavy load in the hope that the burst will end before the temporary filesystem fills. We have not explored this possibility.

Note that in a few cases, the graphs shows the maximum small packets captured rate above the small packets offered rate. The occurred for two reasons. First, both values were experimentally determined, and as can be seen from the error bars on the offered load, the load generators are inconsistent in how quickly they can generate small packets. Since the offered rate was only sampled over about 20 cycles and the captured rate was sampled over about 700 cycles, it is likely that in a few cases the load generators performed better during the capture test than the offered analysis. Second, the captured rate is the rate that the driver is capturing at, and if there is a backlog in the kernel buffers then the driver could capture packets at a rate faster than they actually arrived. Over the 11 hours of the test, it is likely this occurred a few times.

## 6 Data Conversion

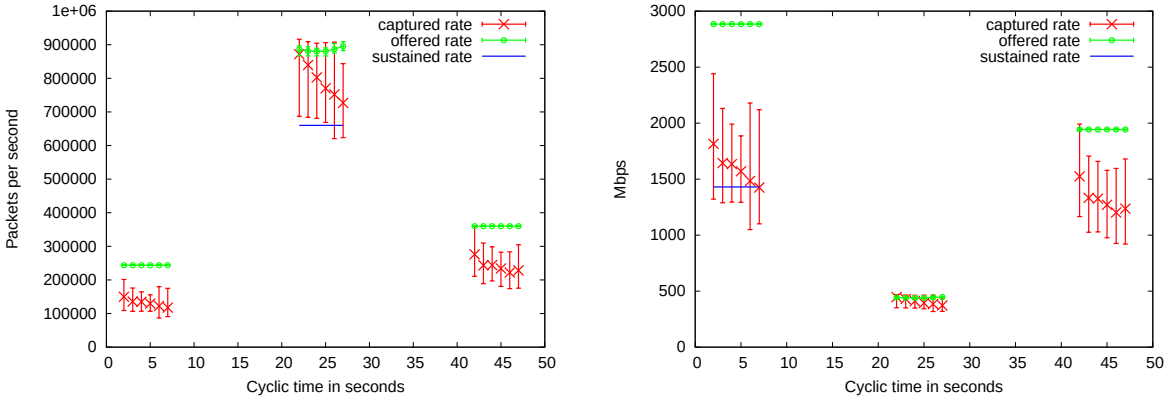
As we discussed in Section 3, we introduced an intermediate step between the collection and analysis of the data, to address a number of issues. In this section we describe how we convert large data sets.

Our conversion process requires two passes over the data. The first pass determines a unique identifier for each NFS request or response. To reduce the time to complete this pass, this can be (highly) parallelized. Counts of the unique transactions in each file are maintained and then merged into a start offset for each file.

The second pass assigns the unique identifiers and extracts the desired information about each transaction. Again, this step can be parallelized to reduce the overall conversion time. The offset calculated for each file in the first pass is used to enable assigning the unique identifiers in parallel. Also during this pass, information on each NFS request or response is written to one or more of the following four types of DataSeries extents: *NFS common*, *NFS attribute*, *NFS read-write* and *NFS mount*. The first of these extent types records generic information on each NFS transaction, such as the source and destination addresses, the transaction identifier, the operation type, and the payload length. The other NFS extent types record information specific to selected NFS operations, such as read/write operations, or operations that include attributes.

A fifth extent type was kept for packet-level information. This recorded network and transport-layer data such as the source and destination addresses and ports, the transport layer protocol used, the packet length, and the packet type (TCP, UDP, other).

When we converted the empirical data sets to DataSeries, we utilized 20-80 CPUs; this provided sufficient computational power for all of the conversion and compression tasks to make the file server the limiting resource. Both passes typically required 12-24 hours. We




(a) (b)

Figure 4: Burstiness results: (a) packets per second; (b) Data transferred per second

considered this a reasonable cost, as having the data in the DataSeries format saved a significant amount of time during the analysis stage, due to the efficiencies it provides, and so that we could store some of the NFS information encrypted so that other researchers were allowed to work with it to write analysis.

Once all of the data is converted to DataSeries, the final step (prior to running the analyses) is to index the data files. This process creates an index file that contains information on the minimum and maximum time stamps and identifiers in each extent. The index also contains the filename and modification time of each DataSeries file, so that it can be skipped during re-indexing if the file has not changed. The indexing process is relatively quick; for example, to index the 10 Gb/s trace took 47 minutes, and was I/O bound.

## 7 Data Analysis

In our feasibility studies, we designed and implemented 16 different NFS analyses. Some of these examined characteristics of the files (e.g., age, size, etc.), while others examined higher level issues (e.g., the effect of server latency on throughput of the service). The analysis that convinced us we would be able to provide a remote computational facility was the one that looked at how recently files had been accessed and demonstrated that straightforward read caching would be sufficient. Most of these analyses were developed for our initial study and then reused in our subsequent study. All of the analyses were implemented in C++, and read the DataSeries files using the DataSeries library.

Due to the large size of the data sets, we attempt to exploit parallelism where possible. The biggest parallelism happens from evaluating over different time ranges, which indexing in DataSeries simplifies.

As an example of how DataSeries assists with the analysis of large data sets, we timed the server latency analysis on both the converted 1 Gb/s and 10 Gb/s traces running at the same time on a 4 CPU DL580 (same configuration as for the tracing machine) This analysis uses the NFS common extent. For the 10 Gb/s trace, in 4,063 wall clock seconds (about 1 hour) it read 17.8 GB of data (109 GB when uncompressed), and processed 2 billion "rows" (NFS transaction records), roughly 500,000 rows per second. Started at the same time, but running for 11,263 wall clock seconds (about 3 hours), the 1 Gb/s trace read in 45.5 GB of data (406 GB when uncompressed) and processed 7.6 billion rows, roughly 675,000 rows per second.

## 8 Observations and Lessons Learned

Based on our experiences with data collection and analysis of 1 and 10 Gb/s networks, we have identified a number of things that we would do differently, if we were to start over. We elaborate on several of these below.

One of our assumptions when we designed our monitor was that writing to disk would be the primary bottleneck. However, at times other resources proved to be a bottleneck as well. In particular, memory bandwidth became an issue at times. Using a modern server may alleviate this bottleneck, as many new servers have significantly higher memory bandwidths [23] than does our three year old server.

As network bandwidths increase, so to do the potential sizes of traffic bursts. Obviously, including more (and faster) memory would increase the duration that a large burst could be buffered, although the 'drain' period would increase if the memory bandwidth is a bottleneck. A related optimization would be to rate limit the compression processes during bursts. If the compression and

dumping processes were more tightly integrated, then during bursts the compression programs could run more slowly so as to reduce memory bandwidth contention.

On 10 Gb/s networks that are heavily utilized, the PCI-X bus on which our NIC resides could become the primary bottleneck. In particular, the theoretical maximum rate of the PCI-X bus is 8.5 Gbps, which is less than the maximum rate of a 10 Gb/s network. In theory, a modern PCI-Express x8 slot (32 Gb/s) would easily keep up with a 10 Gb/s network.

We expect that running `driverdump` on a 64-bit machine would provide additional performance improvements. In particular, on a 64-bit machine the kernel could keep all of the memory mapped into the kernel address space, avoiding the need to map and unmap pages.

Another significant optimization would be to have the driver write files directly to the output disks when the compression process falls far enough behind that the (new) files are not going to be compressed anyway. Since our driver always writes to `tmpfs`, uncompressed output is effectively copied twice. This change would have required much tighter integration between the compression and the capture portions of `driverdump`, which currently only interact through the filesystem.

If the NIC is programmable, the capture process could be moved directly onto the NIC. The NIC could write the data in `pcap` format into memory buffers that are then compressed and written out to disk. This should provide the highest performance single-box capture tool.

Some high-speed networks utilize 9000 byte MTUs (Maximum Transmission Units) rather than 1514 bytes. The only change required for our monitor to work on such networks is to configure the capture NIC to accept 9000 byte packets. However, we expect 9000 byte MTUs would exacerbate the memory bandwidth limitation.

## 9 Conclusions

In this paper we described our experiences with designing and using packet capture in a NIC driver (`driverdump`) to collect full packet traces on 1 and 10 Gb/s networks. On enterprise networks, we demonstrated multi-day packet captures of bursty traffic with minimal packet capture drops. We demonstrated the improvements `driverdump` provides over existing packet capture tools such as `tcpdump` and `lindump`. We discussed the current limitations of our monitor, and the opportunities that exist for overcoming them. Finally, we introduced the methodology and tools we utilize for efficiently analyzing the large data sets that we have collected using `driverdump` on live 10 Gb/s networks.

In the future we intend to upgrade the server platform on which we run `driverdump`, so that we might alleviate the current bottleneck (memory bandwidth) in order to evaluate the packet and byte capture rates that can be reached before the next bottleneck emerges. We also plan to investigate some if not all of the potential enhancements that we listed in the paper, to determine whether sustained full packet capture-to-disk is possible at line rate on heavily utilized 10 Gb/s networks.

## Acknowledgments

The authors would like to thank Rick Jones for lending us the 10 Gb/s NICs used in the experiments, and for his feedback on a preliminary version of this paper. Similarly, the authors would like to thank Bryan Stephenson, Alistair Veitch, and John Wilkes for their feedback.

## References

[1] bzip2 compression library, http://www.bzip.org/.
[2] DAG monitoring cards, http://www.endace.com/networkMCards.htm.
[3] DataSeries software, http://tesla.hpl.hp.com/public_software.
[4] L. Degioanni and G. Varenni, "Introducing Scalability in Network Measurement", *Proceedings of Internet Measurement Conference (IMC 2004)*, pp. 233-238, Oct. 2004.
[5] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer, "Passive NFS Tracing of Email and Research Workloads", *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST'03), pp 203–216* San Fransisco, CA, USA, March 2003.
[6] ethereal software, http://www.ethereal.com/.
[7] S. Han, M. Kim, H. Ju and J. Hong, "The Architecture of NG-MON: A Passive Network Monitoring System for High-Speed IP Networks", *Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems, Operations and Management: Management Technologies for E-Commerce and E-Business Applications*, pp. 16-27, 2002.
[8] T. Johnson, C. Cranor and O. Spatscheck, "Gigascope: A Stream Database for Network Application", *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 647-651, June 2003.
[9] lindump software, http://awgn.antifork.org/codes/lindump.c
[10] linux networking notes, http://limnos.csrd.uiuc.edu/notes/linux-networking/core.html
[11] LOBSTER: Large scale monitoring of Broadband Internet Infrastructures, http://www.ist-lobster.org/.
[12] lzf compression library, http://www.goof.com/pcg/marc/liblzf.html.
[13] lzo compression library, http://www.oberhumer.com/opensource/lzo.
[14] U. Manber, "Finding Similar Files in a Large File System", *Proceedings of the USENIX Winter 1994 Technical Conference, pp 1-10"*, San Fransisco, CA, USA, 1994.
[15] J. Micheel, "lambdaMON - a Passive Monitoring Facility for DWDM Optical Networks", *Passive and Active Measurement Workshop (PAM 2005)*, Boston, MA, April 2005.
[16] J. C. Mogul and K. K. Ramakrishnan, "Eliminating Receive Livelock in an Interrupt-Driven Kernel" *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp 217–252, 1997.
[17] A. Moore, J. Hall, C. Kreibich, E. Harris and I. Pratt, "Architecture of a Network Monitor", *Passive and Active Measurement Workshop (PAM 2003)*, pp. 77-86, April 2003.
[18] NavTel Interwatch network testing solution, http://www.navtelcom.com/.
[19] network trunking, http://en.wikipedia.org/wiki/Link_aggregation.
[20] OC192mon Project, http://moat.nlanr.net/NEW/OC192.html.
[21] oprofile system-wide profiler for Linux, http://oprofile.sourceforge.net/.
[22] Sprint IP Monitoring Project, http://ipmon.sprint.com/.
[23] Streams Benchmark, PC results, http://www.cs.virginia.edu/stream/peecee/Bandwidth.html
[24] tcpdump and libpcap software, http://www.tcpdump.org/.
[25] tcptrace software, http://www.tcptrace.org/.
[26] tmpfs file system, http://www-128.ibm.com/developerworks/library/l-fs3.html.
[27] B. Viken and P. Heegaard, "Performance evaluation of a low-cost network monitor", *Proceedings of 17th Nordic Teletraffic Seminar*, Fornebu, Norway, Aug. 2004. http://www.telenor.com/rd/pub/rep02/R_51_2002.pdf.
[28] Y. Zhou, T. Kelly, J. Wiener and E. Anderson, "An Extended Evaluation of Two-Phase Scheduling Methods for Animation Rendering", *11th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 2005)*, Cambridge, MA, June 2005.