Characterising the Behaviour and Performance of SIFTR v1.1.0

Lawrence Stewart, Grenville Armitage, James Healy
Centre for Advanced Internet Architectures, Technical Report 070824A
Swinburne University of Technology
Melbourne, Australia
lastewart@swin.edu.au, garmitage@swin.edu.au, jhealy@swin.edu.au

***Abstract*—Experimental research into TCP behaviours requires an in depth view of the networking stack's internal state held for each TCP connection of interest. SIFTR (Statistical Information For TCP Research) [1] is a recently released, freely available FreeBSD 6.2 kernel module that intercepts TCP packets as they traverse the network stack within the kernel. The performance characteristics of SIFTR were obtained by stress testing the software under a range of conditions. This information can be used as a basis for estimating experimental error inherent in collected SIFTR data, as well as broadly determining whether SIFTR might be suitable for a particular task. The experimental methodology is also completely described, so that SIFTR's operational limitations can be measured on different testbeds. With SIFTR running on 2004-era commodity PC hardware, configured with maximum data logging granularity, up to 100 TCP flows can achieve aggregate throughput of at least 204Mbps with a worst case skip rate of 16.2 skipped packets per Mbps throughput. SIFTR was also evaluated on a second testbed consisting of much newer 2007-era dual core commodity PC hardware. With SIFTR running on this hardware, configured with maximum data logging granularity, up to 100 TCP flows can achieve aggregate throughput of at least 565Mbps with a worst case skip rate of 31.1 skipped packets per Mbps throughput.**

***Index Terms*—Experimental research, TCP, FreeBSD, SIFTR**

I. Introduction

Experimental research into TCP behaviours requires an in depth view of the networking stack's internal state held for each TCP connection of interest. Obtaining such state information is tricky, and relies on an in depth knowledge of the operating system's kernel and low-level programming skills to extract the relevant information.

Projects relevant to this type of TCP research are few and far between, with the most prominent being the web100 [2] kernel patch and libraries for the Linux operating system. Unfortunately, web100 does not yet support other open source operating systems commonly used for networking research (such as the BSD variants).

CAIA's NewTCP project [3] mandated the use of the FreeBSD [4] operating system as the platform for performing our TCP research. Background research undertaken at the beginning of the project revealed no equivalent to web100 for FreeBSD, and the effort required to port web100 to FreeBSD was deemed to be too large a task for us to undertake. We came to the conclusion that writing our own FreeBSD kernel code to export the necessary information required to perform our research was the most feasible option.

SIFTR (Statistical Information For TCP Research) [1] is a recently released, freely available FreeBSD 6.2 kernel module that intercepts TCP packets as they traverse the network stack within the kernel. On interception, a log message is generated containing information about the TCP connection the packet relates to, and written to a plain text log file on the computer's hard drive. It provides the ability to make highly granular measurements of TCP session state information. This is achieved by inserting a new function into the path of AF_INET (IPv4) [1] packets traversing the FreeBSD network stack. IP packets carrying TCP traffic are scrutinised by this function to extract detailed information about the TCP connection the packet is associated with. This detailed information is then written to a log file on the computer's file system for post analysis.

The module has only been tested on FreeBSD 6.1-RELEASE and 6.2-RELEASE thus far, but should work with all FreeBSD 6.x releases, and possibly earlier (5.x) or up and coming (7.x) releases.

Using SIFTR in experimental research requires the

[1]IPv6 support could be added with minimal effort, perhaps in a future release

researcher to be able to quantify the error introduced by the measurement tool on the experiments. This report aims to identify the key operational limitations of SIFTR and quantify them for two specific testbeds. This is achieved by examining the performance of the testbeds with and without the presence of SIFTR in various configuration states. The experimental methodology is also completely described, so that SIFTR's operational limitations can be measured on different testbeds.

This report is structured as follows: section II broadly introduces SIFTR, section III describes the configuration of the experimental testbed, section IV outlines the testing methodology used to critically evaluate SIFTR, section V analyses the testing results, and finally section VI concludes with a summary of key findings and outlines possible further work.

II. Introducing SIFTR

SIFTR works by inserting itself between the IPv4 and TCP layers in the FreeBSD TCP/IP network stack. Figure 1 provides a high level overview of the FreeBSD TCP/IP stack and Figure 2 illustrates where SIFTR inserts itself.

Packets enter the stack from the various data link layer drivers present in the system. Assuming they are IPv4 packets, they are passed to the ip_input() function for IP decoding. If the IP packet is destined for the host and contains a TCP segment, the segment is passed to the tcp_input() function for processing. For each TCP connection the host is an endpoint for, a TCP protocol control block is maintained in the kernel's memory. The control block holds all of the necessary state required to maintain a valid TCP connection. The tcp_input() function queries and possibly updates the values in the control block the packet is associated with. After the TCP layer has processed the segment, any payload will be made available to the application consuming the data via the sockets application programming interface (API).

Payload generated by an application flows down the stack, first through the sockets API, then through the transport layer tcp_output() function, through the network layer ip_output() function and finally out of the machine via the network driver operating at the data link layer.

SIFTR uses inbound and outbound TCP packet events to trigger a capture of the state of the TCP control block related to the packet. In effect, it samples the state of the control blocks of active TCP connections at intervals related to the frequency with which packets are being sent or received for the connection.

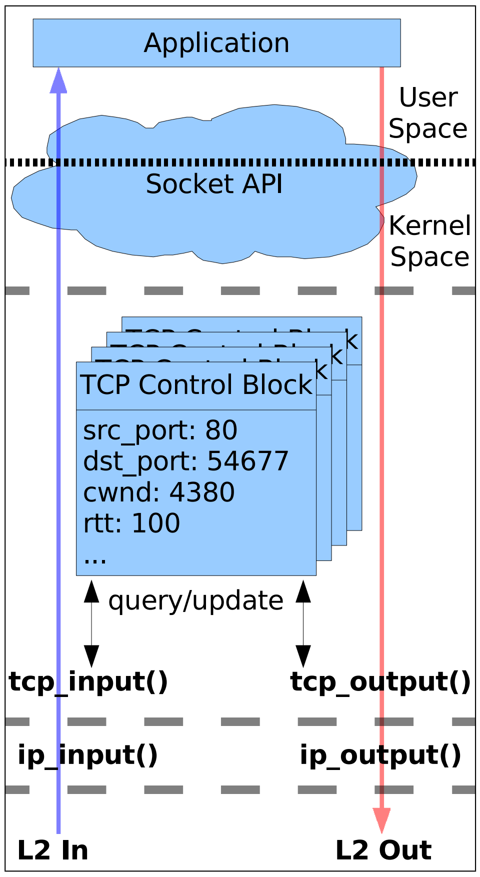

Fig. 1. Overview of the FreeBSD TCP/IP network stack

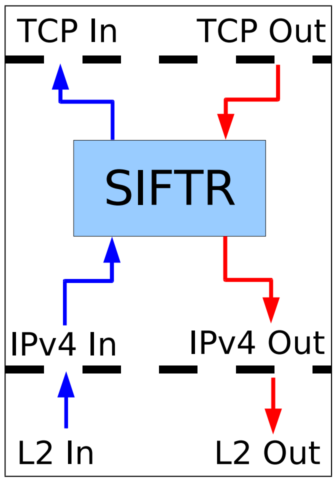

Fig. 2. SIFTR in the kernel

SIFTR was designed to minimise the delay introduced to packets traversing the network stack. This design called for a highly optimised and minimal hook function that extracted the minimal details necessary whilst holding the packet up, and passing these details to another thread for actual processing and logging.

Figure 3 illustrates the multithreaded internal flow of execution within SIFTR.

Packets enter SIFTR via a hook function at the IPv4 layer of the network stack. Assuming the packet is TCP, the TCP control block corresponding with the flow the packet belongs to is obtained using a hash table lookup. The current state of the control block is then copied into a generic pkt_node structure. This structure is passed by the network thread marshalling the packet through the network stack to the SIFTR pkt_manager thread by way of a shared queue. Once the pkt_node has been inserted into the queue, the network thread exits the hook function and continues processing the packet as required. It is then up to SIFTR's pkt_manager thread to pull the pkt_node structures from the queue and process them in its own time.

This multithreaded design does introduce contention issues when accessing the shared queue between the threads of operation. When the hook function tries to enqueue a pkt_node structure, it must first acquire an exclusive lock to access the queue. Likewise, when the pkt_manager thread attempts to dequeue a pkt_node structure, it must also acquire an exclusive lock to do so. If one thread holds the lock, another thread cannot access the queue.

To minimise delay, the SIFTR hook function will only attempt to acquire the lock once, and if it fails, will drop the pkt_node structure and allow the IP packet to exit the hook function. SIFTR refers to this outcome as a "skipped packet", because the copied control block details triggered by the packet will not be processed by the pkt_manager thread. Note that SIFTR always ensures that IP packets are allowed to exit the hook function and continue through the stack, even if they could not successfully trigger a pkt_node structure to be added to the queue.

The number of flow packets that trigger a log message to be generated for that flow is controlled by the SIFTR packets-per-log (PPL) configuration variable. As a result of SIFTR performing its processing in a separate thread, this variable only applies to packets that successfully insert a pkt_node structure into the shared queue. If a packet is skipped (does not successfully insert a pkt_node structure into the shared queue), SIFTR currently has

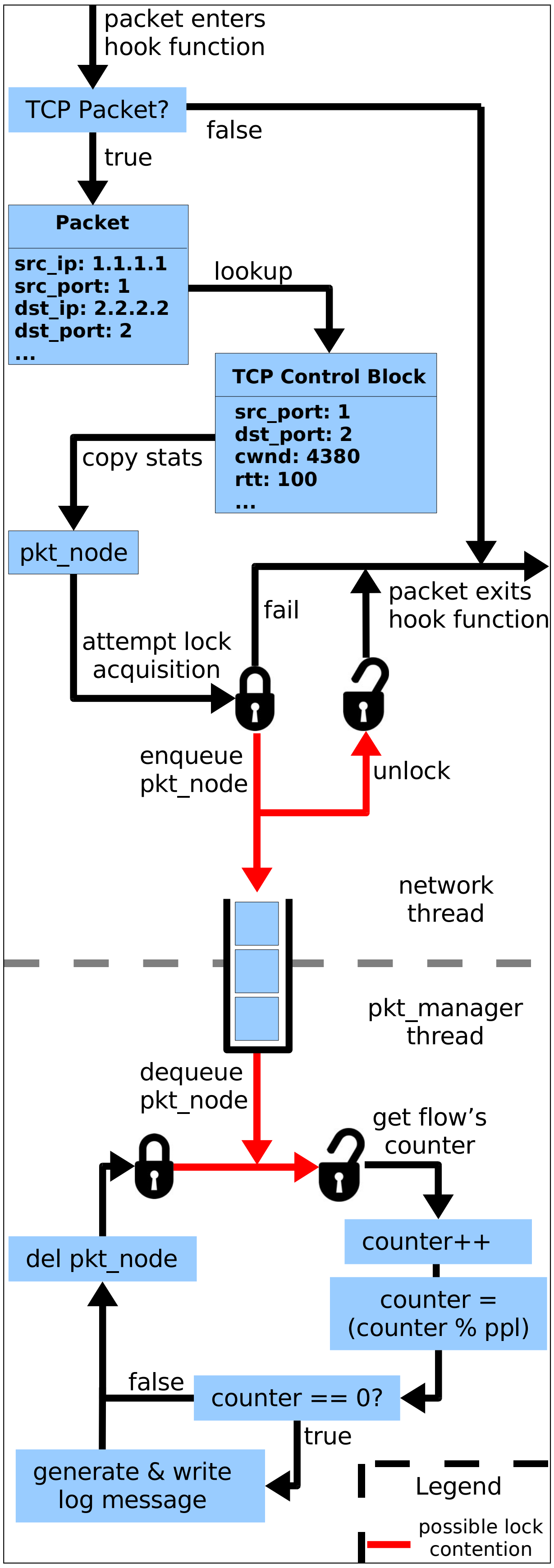

Fig. 3. SIFTR internal operations

no means of knowing which flow the skipped packet belonged to. For example, if PPL was set to 5 and the fourth packet of a particular flow was skipped, the pkt_manager thread would count the pkt_node struct generated by packet 5 as if it was triggered by packet four. Therefore, the sixth packet will actually trigger the log message instead of the fifth. This behaviour is discussed further in section V-C.

On the other side of the queue processing the pkt_node structures is the SIFTR pkt_manager thread. The thread performs an (almost) infinite loop, starting with acquiring the shared queue lock. If the lock is already held by a network thread in the hook function, the pkt_manager thread will wait for the lock to become available. Once acquired, the pkt_manager thread will dequeue a pkt_node structure from the queue and then release the lock so that other network threads can continue adding pkt_node structures to the queue. It then performs a hash table lookup to obtain the packet counter for the appropriate flow and increments the counter. The counter is then assigned the value of the remainder of the counter divided by the SIFTR PPL variable. If the remainder is not 0, the memory allocated to the pkt_node structure is released and the loop begins again. If the remainder is 0, a log message is generated and buffered for writing to disk prior to releasing the pkt_node's memory.

Network threads tied to TCP connections are the only network threads that will attempt to acquire the shared lock, as non TCP network threads exit the hook function at the start. Some experimentation revealed that access to the hook function by all network threads is not serialised i.e. multiple separate network threads can be executing within the hook function code at the same time. This has the effect of increasing the likelihood of contention for the shared queue lock with increasing numbers of TCP network threads. This behaviour and its impact on skipped packets is discussed further in section V-C.

One might ask why the SIFTR hook function actually processes every packet for every TCP flow instead of only the packets required to meet the log message quota dictated by the SIFTR PPL setting. We found that inserting the code into the hook function that is necessary to identify the flow a packet belongs and decide if the packet should trigger a log message significantly impairs the kernel's ability to process network data. Given that the hook function is in the fast path of packets traversing the network stack, every additional bit of processing adds up when it is performed on a per packet basis. This performance impairment is what prompted us to investigate a multithreaded design that moved as much of the processing away from the fast path as possible.

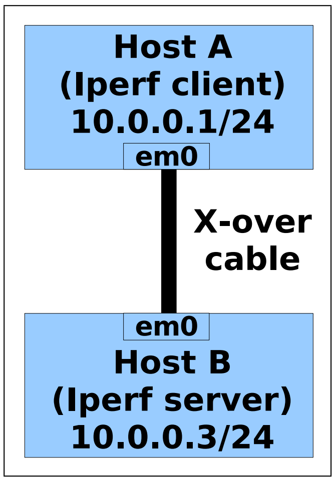

Fig. 4. SIFTR Testbed

More details about SIFTR can be found in the README, which is included in the SIFTR source distribution available at [1].

III. Experimental set up

Figure 4 illustrates the simple testbed layout that was used to test SIFTR. Two different sets of hardware were used to evaluate SIFTR's performance (refer to Table I and Table II for specifications). Each set of hardware used two identical PCs as host A and B, connected via a 2m CAT5e cross-over cable between their PCI Intel NICs. The onboard NICs were disabled during the testing.

All four PCs were configured identically except that testbed 2's PCs ran a symmetric multi-processing (SMP) kernel to take advantage of the CPU's two cores. The testbed configuration details are provided in Table III. Configuration options not explicitly stated in Table III should be assumed to have been left at their default values. The testbed PCs also had the kernel modification relating to the TCP host cache as discussed in [5] applied, with the prune time lowered to 5 seconds. This, combined with the *net.inet.tcp.hostcache.expire* sysctl variable being set to 1, ensured that the host cache was emptied within 5 seconds of a TCP flow terminating. Therefore, subsequent connections between the two hosts were not affected by the host cache and proceeded as though the two hosts had never communicated previously.

IV. Testing methodology

The way SIFTR inserts itself into the network stack affects the system's networking performance in two

Motherboard	HP Compaq D530C
CPU	Intel Pentium 4 2.66GHz
RAM	1GB (1 x 1GB) PC3200 DDR-400
HDD	Maxtor 40GB 6E040L0 UDMA100
NIC	Broadcom BCM5705 PCI gigabit Ethernet (onboard) Intel PRO/1000 GT 82541PI PCI gigabit Ethernet

TABLE I
TESTBED 1 PC SPECIFICATIONS

Motherboard	Intel Desktop Board DG965WH
CPU	Intel Core2 Duo E6320 1.86GHz 4MB L2 Cache
RAM	1GB (1 x 1GB) PC5300 DDR2-667
HDD	Seagate 250GB ST3250410AS SATA II
NIC	Intel 82566DC PCIe gigabit Ethernet (onboard) Intel PRO/1000 GT 82541PI PCI gigabit Ethernet

TABLE II
TESTBED 2 PC SPECIFICATIONS

OS	FreeBSD 6.2-RELEASE
Kernel Configuration	Used the GENERIC kernel configuration file for both testbeds with the “i486”, “i586” and “makeoptions DEBUG=-g” options removed. The testbed 2 PCs had “options SMP” added to their configuration.

Boot loader tuning	kern.ipc.nmbclusters=100000
	vm.kmem_size=524288000
	vm_kmem_size_max=524288000

Sysctl tuning	kern.ipc.maxsockbuf=10485760
	net.inet.tcp.sendspace=1048576
	net.inet.tcp.rescvspace=2097152
	net.inet.tcp.hostcache.expire=1
	net.inet.tcp.inflight.enable=0

Intel NIC driver version	6.2.9
Intel NIC media	1Gbps (autoselect)
TCP MSS	1460 bytes
TCP benchmark tool	Iperf v2.0.2 with the CAIA patch (downloadable from [1]) applied

TABLE III
TESTBED PC CONFIGURATION

ways. Firstly, the PFIL [6] [7] architecture used by SIFTR places hook functions in the path of packets traversing the stack, as opposed to providing the hook functions with a copy of the packet and allowing the packet to continue on its way. This forces packets to wait in the stack while the hook function processes the packet. Secondly, SIFTR uses processor time to perform its operations, which reduces the processor’s ability to handle NIC interrupts and network stack operations amongst other things.

The decision was made to measure SIFTR’s impact in terms of the throughput achievable by the system. Initial exploratory testing on testbed 1’s PCs showed a direct relationship between SIFTR, its configuration variables and the test system’s achievable TCP throughput.

The Iperf [8] network testing software was used to generate the TCP flows across the testbed. The Iperf client and server were run on Host A and Host B respectively. Data flowed from the client to the server and acknowledgments flowed in the reverse direction.

A set of test cases was devised to investigate how SIFTR affected the throughput when run on the Iperf client (data sender), server (data receiver), and both at the same time, with differing numbers of concurrent flows and levels of data gathering granularity. The granularity was controlled by the *net.inet.siftr.ppl* SIFTR packets-per-log sysctl configuration option. Results are presented for PPL values of 1, 2, 5, 10, 50 and 100. Values greater than 100 were tested, but produced too few data points in the SIFTR log file to be worthwhile in any sort of experiment, and have been omitted. The number of concurrent flows was controlled using Iperf’s *-P* option, with results presented for 1, 2, 5, 10, 50 and 100 flows. Note that tests with a single flow did not specify the *-P* option, as Iperf defaults to use of a single flow.

The Iperf command run on the server was:

```
iperf -f b -s
```

The Iperf command run on the client varied, but was of the form:

```
iperf -f b -t 60 [-P x] -c 10.0.0.3
```

where *10.0.0.3* was the IP address of the Iperf server, and *[-P x]* was specified when more than a single flow was required.

Each test case was run for sixty seconds and repeated ten times, with a minimum six second pause between

	Client Throughput (bps)	
No. Flows	**Testbed 1**	**Testbed 2**
1	632715591.7	562767781.6
2	626038744.4	560847578.0
5	623709119.7	558416317.8
10	624715981.7	556940496.9
50	620669551.4	556278503.2
100	614400388.1	553566126.1

TABLE IV
MULTIFLOW, NO SIFTR BASELINE THROUGHPUT RESULTS

	Client Throughput (bps)	
	SIFTR Unloaded	**SIFTR Loaded and Disabled**
Testbed 1	632715591.7	632564556.2
Testbed 2	562767781.6	562802693.7

TABLE V
SINGLE FLOW, SIFTR UNLOADED VS SIFTR LOADED AND DISABLED THROUGHPUT RESULTS

each repetition and the start of a new test case. This pause ensured the host cache did not influence results as previously discussed.

For each test case repetition, the number of bytes transferred and throughput reported by both the client and server were recorded. For test cases that utilised SIFTR, the SIFTR log file was also archived for post analysis with a file name identifying the test case and repetition number the log belonged to.

V. PERFORMANCE ANALYSIS

A. *Testbed baseline*

A set of baseline test cases were run to establish each testbed's throughput with 1, 2, 5, 10, 50 and 100 concurrent flows between client and server without SIFTR running. The results for testbed 1 and testbed 2 are presented in Table IV.

The maximum throughput of testbed 1 is 632Mbps, and testbed 2 is 562Mbps. As the number of flows increases, the aggregate throughput decreases, at worst by an amount of approximately 2.9% on testbed 1 and 1.6% on testbed 2 when 100 concurrent flows are active. This gradual drop can be attributed to the proportional increase in user-space processing requirements to handle each of the flows, as well as the contention for processing and network resources on the testbed systems. The reduced drop on testbed 2 is likely attributable to the kernel's use of SMP to distribute the load between both CPUs concurrently.

Testbed 2 was expected to outperform testbed 1 given that its hardware was significantly newer. However, this is observed not to be the case. Some further investigation revealed a combination of factors caused this outcome. Firstly, the speed of the CPU cores on the testbed 2 machines is actually lower than the testbed 1 machines. The newer CPUs utilise larger layer 2 caches, better memory access and more processing cores to provide a more seamless multitasking experience. However, in situations like our baseline test cases, where there is only one main user process running, the newer CPUs appear unable to perform as well as the older, faster CPU. Secondly, FreeBSD SMP network performance is known to have some room for improvement compared to uni-processing network performance [9]. FreeBSD 6's SMP performance is significantly better than FreeBSD 5, but there are still improvements being made for the upcoming FreeBSD 7 release [10].

These results should be taken into consideration for comparison with the results obtained in the test cases that utilise SIFTR.

Given that SIFTR must run on the communicating TCP end-points in order to extract connection data, we did not test above 100 concurrent flows. A test scenario involving monitoring more than 100 flows originating from a single host is unlikely to be capable of achieving full performance. We therefore deemed 100 flows to be a reasonable limit, given the the majority of tests are likely to involve fewer than 10 flows from a single TCP end-point.

We also measured the throughput of the testbed for a single TCP flow when SIFTR was loaded into the kernel, but not enabled i.e. the sysctl *net.inet.siftr.enabled* variable was set to 0. The results are presented in Table V. Iperf will routinely report throughputs that differ by up to 1Mbps between test runs under the same test conditions, which is the cause of the minor differences in throughput results. We can therefore state that there is no significant difference between the testbed's performance when SIFTR is not in the running kernel at all, versus when SIFTR is loaded in the running kernel but not enabled. This is to be expected based on the actual design of the SIFTR software. It is therefore safe to load SIFTR into a running kernel without enabling it, as this will not affect the TCP throughput achievable by the kernel.

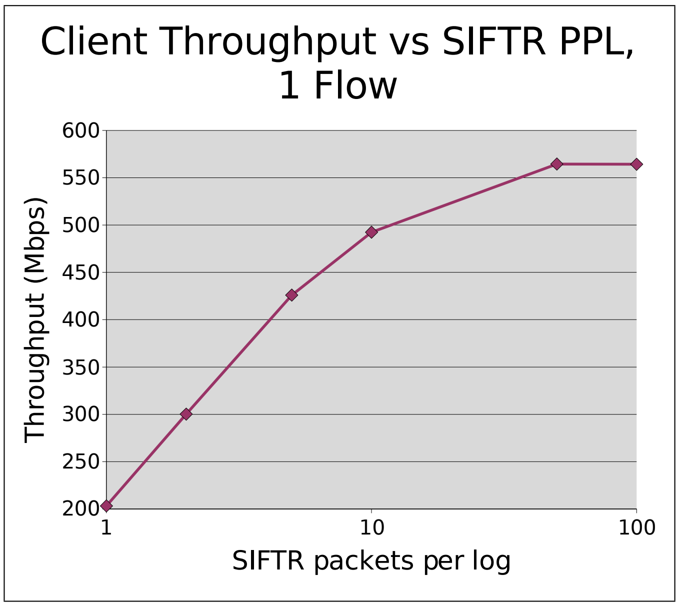

Fig. 5. Testbed 1 Client Throughput vs SIFTR packets-per-log

B. *SIFTR granularity testing*

The SIFTR packets-per-log (PPL) *net.inet.siftr.ppl* sysctl configuration variable controls how often SIFTR logs connection state. The logging is based on inbound and outbound TCP packet events. For example, setting *net.inet.siftr.ppl* to 1 causes a log message to be generated for each inbound and outbound packet belonging to a particular flow, that captures the TCP connection state for that flow. The frequency of this logging affects the amount of processing load placed on the system by SIFTR, and therefore affects the achievable throughput. For these series of tests, SIFTR was run solely on the Iperf client machine (data sender).

Figures 5 and 6 show the throughput vs SIFTR PPL results as reported by the Iperf client for a single flow on testbeds 1 and 2 respectively.

With SIFTR at its highest granularity setting of 1 packet per log on testbed 1, the client host is capable of achieving a throughput of almost 205Mbps. The achievable throughput grows reasonably consistently with increasing PPL, up to the point where there is negligible gain in throughput as PPL surpasses 50.

Testbed 2 exhibits vastly different characteristics. Running SIFTR actually improved the throughput performance of the testbed compared to the baseline tests. The client host achieved an average throughput of 574Mbps with SIFTR at its highest granularity setting of 1 packet per log, compared to the baseline of 562Mbps. This result was repeatable, and there were no anomalies in the data from the individual test runs on either the Iperf client or Iperf server that skewed the results.

Further investigation of the data sender's SIFTR log files revealed that the congestion window was collapsing back to one segment size at a fairly regular rate. This indicates scattered packet loss events occured during the test runs, which triggered TCP congestion avoidance behaviour. Whilst we have no retrospective way of being sure that packet loss events were occurring in the baseline tests, the evidence suggests that they were. SIFTR's design causes it to introduce a small, additional delay to each packet handled by the hook function. For the test results utilising SIFTR to exhibit higher throughput than the baseline tests, this would indicate that more TCP loss events were occurring in the baseline tests than the SIFTR enabled test cases. We can therefore suggest that the additional delay introduced by SIFTR was somehow reducing the strain on whatever bottleneck within the system was causing these packet loss events.

To test this hypothesis, we constructed a test kernel module with a hook function that simply iterated through an empty for loop ten thousand times and then allowed the packet to continue on its way. This module caused the throughput reported by Iperf during testing to exhibit the same behaviour as the SIFTR tests i.e. with the module loaded, the throughput reported by Iperf is higher than the baseline throughput when no kernel module is loaded. This does allow us to conclude that it is not SIFTR's code that is responsible for the unexpected results. Rather, some underlying hardware or software bottleneck (buffer perhaps?) is occassionally dropping a packet, but does so less frequently when packet inter-arrival times are widened by a small amount. A hook function holding a packet up for a few additional microseconds seems to be enough to allow better aggregate throughput due to a reduced number of loss events. This certainly warrants further investigation, but does explain the unusual test results.

Repeating the test cases shown in Figures 5 and 6 for varying numbers of concurrent flows produced the results shown in Figures 7 and 8 respectively.

For testbed 1, Figure 7 demonstrates that the throughput achieved by the client when SIFTR is in use is not overly coupled with the number of concurrent flows. In some cases, we even see a marginal increase in throughput when more concurrent flows are running with the same SIFTR PPL value. The reason for this marginal increase is unknown; perhaps the minor delay introduced by SIFTR in the networking stack causes the flows to

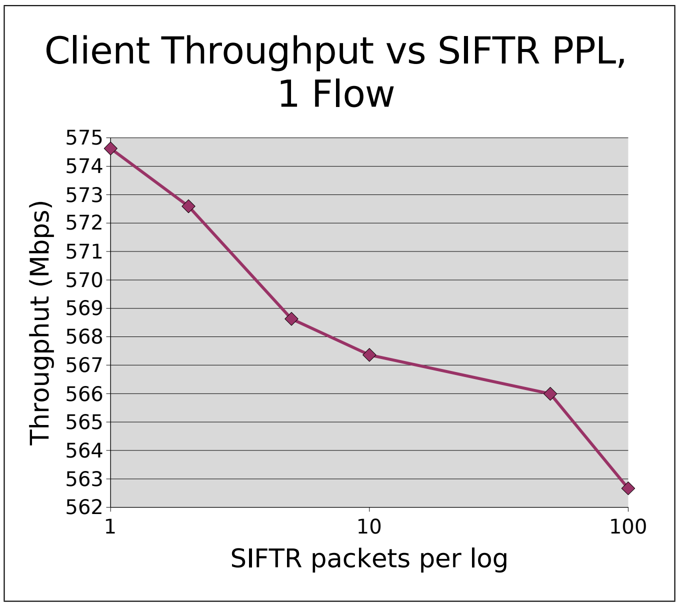

Fig. 6. Testbed 2 Client Throughput vs SIFTR packets-per-log

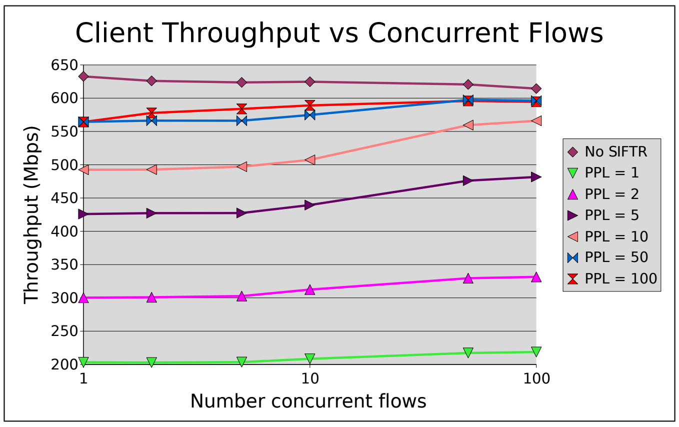

Fig. 7. Testbed 1 Client Aggregate Throughput vs Number Concurrent Iperf Flows

be less synchronised and therefore they maintain a more fluid transfer rate. Despite this, the conclusion can be drawn that SIFTR is more than able to process up to 100 concurrent TCP flows without reducing the throughput capacity of the testbed 1 machines.

Once again, testbed 2 exhibits different characteristics. Increasing PPL and numbers of concurrent flows both reduce the aggregate client throughput. This does suggest that throughput is negatively affected by the number of concurrent flows when running SIFTR in SMP kernels with high data rates. However, throughput for values of PPL up to 50 and for up to 100 concurrent flows is still,

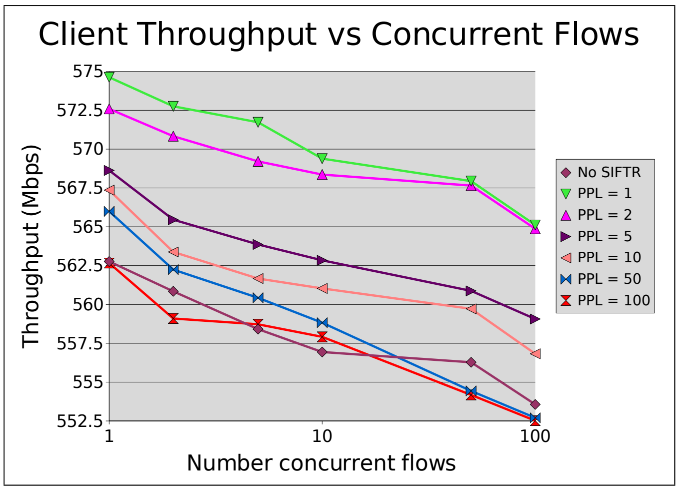

Fig. 8. Testbed 2 Client Aggregate Throughput vs Number Concurrent Iperf Flows

for most combinations tested, better than the baseline results (the “No SIFTR” plot line in Figure 8).

C. SIFTR skipped packet testing

There are a number of potential causes of skipped packets in the SIFTR code that are summarised and accounted for in the SIFTR unload log message. However, viewing the unload log messages for the test cases showed that lock contention accounted for more than 99% (and in most cases 100%) of all skipped packets. We will therefore only discuss skipped packets caused by lock contention in this section.

SIFTR’s design as shown in Figure 3, coupled with the knowledge that access to the SIFTR hook function is not serialised by the kernel, implies that lock contention, and therefore packet skip rate, is likely to be affected by two main factors: the number of active TCP network threads executing within the hook function, and the rate at which packets are entering the hook function.

An example will help illustrate the affect of the first factor on the probability of skipping a packet. If two network threads are executing in the hook function, and one acquires the lock and still holds the lock when the second thread attempts to acquire the lock, the second thread will skip enqueuing the pkt_node, and therefore a skipped packet event is recorded. It is therefore expected that the skipped packet rate will increase as the number of TCP network threads increases.

The second factor’s affect on the probability of skipping a packet is a little more obvious. Increasing the packet rate increases the amount of work imposed on the packet processing thread, which has to acquire and release the shared queue lock each time it dequeues a pkt_node structure. This leads to an increased probability that another network thread attempting to enqueue a pkt_node structure will be denied the lock and therefore skip the packet.

Figures 9 and 10 use a metric of "skipped packets per throughput" on the y-axis, to capture the relationship between throughput (essentially another way of representing packet rate) and the probability of skipping a packet. The number of concurrent Iperf flows does directly translate into an equivalent number of concurrent kernel network threads. As a result of varying PPL in these tests, we inadvertently vary throughput, which results in both skip rate factors affecting tests at the same time.

Figure 9 plots the SIFTR skipped packets per throughput vs the number of concurrent Iperf flows for different values of PPL on testbed 1. As predicted in the preceeding behavioural analysis, we see a changing skip rate as both the number of concurrent flows and PPL vary. Referring back to Figure 7, we observe that increasing PPL results in an increase in the achievable throughput on account of lowered processing load. We would therefore expect to see Figure 9 demonstrate an increase in skip rate as PPL, and therefore throughput, increase, which is the case. We also observe that as the number of concurrent flows increases the increased concurrent contention for the shared queue lock increases the skip rate. As a result of both skip rate factors affecting the tests simultaneously, the increase in skip rate follows a multiplicative rather than additive trend as throughput and number of concurrent flows increases.

The highest level of granularity with a single flow corresponds with the lowest skipped packet rate of 2.1 skipped packets per Mbps throughput. High granularity logging and large numbers of active flows or low granularity logging and small numbers of active flows will ensure the skipped packet rate remains below 20 skipped packets per Mbps throughput on testbed 1 style hardware.

Figure 10 plots the SIFTR skipped packets per throughput vs the number of concurrent Iperf flows for different values of PPL on testbed 2. As with testbed 1, the highest level of granularity with a single flow corresponds with the lowest skipped packet rate, which is 25.7 skipped packets per Mbps throughput for testbed 2.

Figure 8 demonstrates that increasing PPL has a negative effect on the throughput of testbed 2. As such,

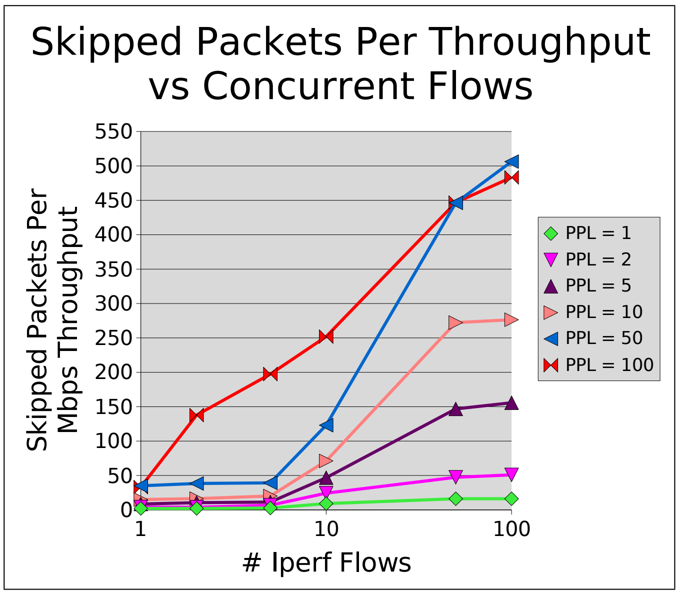

Fig. 9. Testbed 1 Client Skipped Packets per Mbps Throughput vs Number Concurrent Iperf Flows

we would expect Figure 10 not to exhibit an increase in skip rate as PPL increases. However, this is not the case. Referring back to Figure 3, we note that setting PPL greater than 1 allows the pkt_manager thread to avoid generating log messages for every $PPL - 1$ pkt_node structs processed by the thread. For testbed 2, this actually results in the pkt_manager thread acquiring the shared queue lock many more times per second, which in turn increases the probability of skipping a packet. This explains the significant jump in skipped packet rate for PPL=1 and PPL=2 from 25.7 to 68.8 skipped packets per Mbps throughput.

These results indicate that further optimisation of the pkt_manager thread may be possbile to reduce the skip rate.

D. SIFTR end-point testing

Whilst running SIFTR on the data sender will, in most cases, provide more useful data than on the data receiver, it is possible to use it on one or the other or both.

Figure 11 plots throughput vs PPL on testbed 1 for a single flow with SIFTR running on the sending host, receiving host, and both at the same time. Running SIFTR on the receiver clearly results in better throughput. This would indicate that sending data places more load on the testbed 1 systems than receiving it. Running SIFTR on either the sender or both sender and receiver exhibits similar results, though a slight throughput advantage is

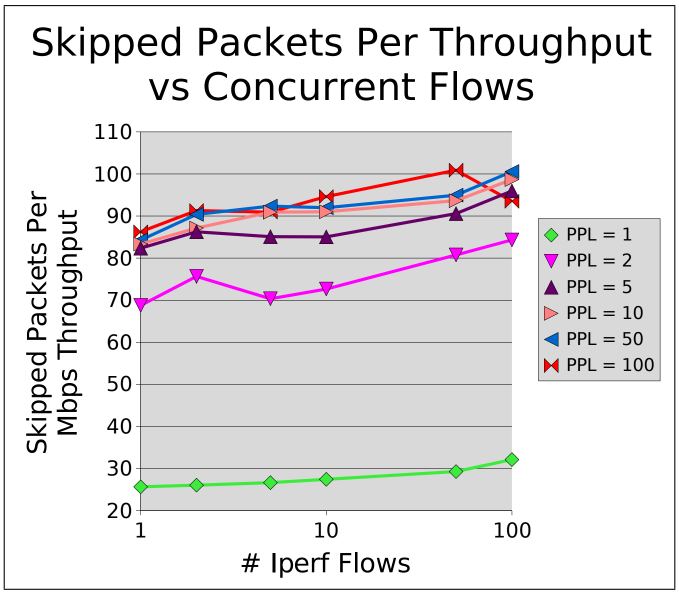

Fig. 10. Testbed 2 Client Skipped Packets per Mbps Throughput vs Number Concurrent Iperf Flows

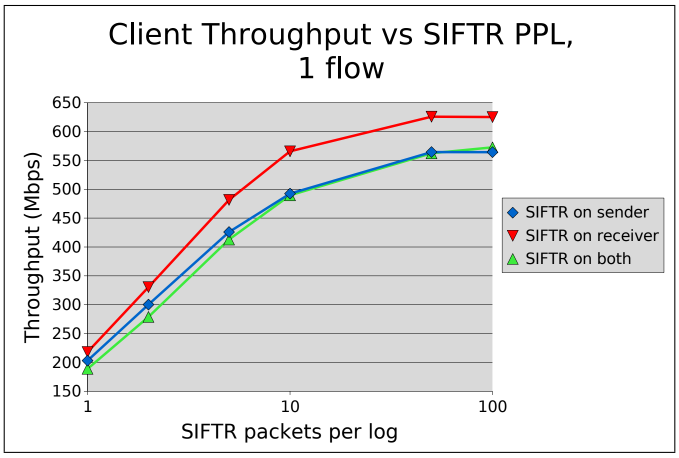

Fig. 11. Testbed 1 Client Throughput vs SIFTR packets-per-log

had by running only on the sender. However, for PPL values greater than or equal to 10, the two cases converge and exhibit identical throughput results. If the data gathered from the receiver is sufficient for a test case, it is clearly advantageous to run SIFTR on the receiver for optimal throughput on testbed 1 style hardware.

Figure 12 plots throughput vs PPL on testbed 2 for a single flow with SIFTR running on the sending host, receiving host, and both at the same time. Testbed 2 shows the receiver to be the processing bottleneck, rather than the sender as was the case with testbed 1. This is illustrated by the fact that the throughput grows as the value of PPL is increased when SIFTR is running on the receiver. Increasing PPL lowers the processing requirements on the host, and as this was the only change made between data points, we can attribute the increase in throughput to the increasing value of PPL. These results indicate that it is preferrable to run SIFTR on the data sender to achieve optimal throughput on testbed 2 style hardware. This is favourable given that we noted running SIFTR on the sender typically yields more interesting data anyway.

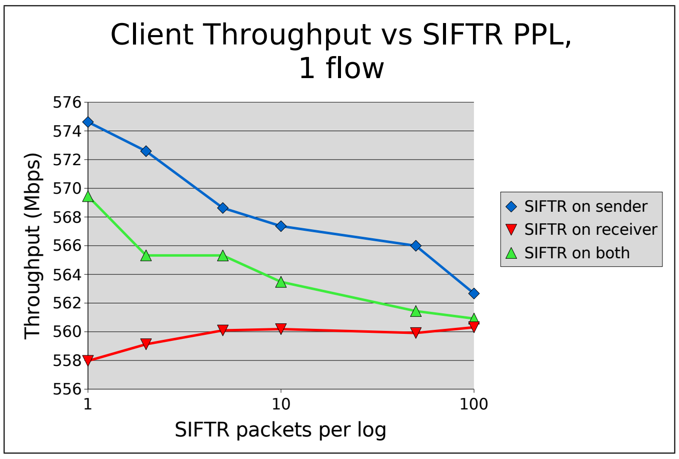

Fig. 12. Testbed 2 Client Throughput vs SIFTR packets-per-log

VI. Conclusion and Further Work

SIFTR intercepts TCP packets as they traverse the network stack within the FreeBSD kernel, logging a message containing information about the TCP connection the packet relates to. SIFTR's primary use is for empirical research into TCP behaviour, and as such is aimed at experimental researchers working with real systems and protocol implementations.

We have measured SIFTR's impact on the TCP throughput achieved by two different test systems. With SIFTR running on 2004-era commodity PC hardware, configured with maximum data logging granularity, up to 100 TCP flows can achieve aggregate throughput of at least 204Mbps over gigabit Ethernet with a worst case skip rate of 16.2 skipped packets per Mbps throughput. This represents the most processor intensive test case run, and therefore sets an upper bound on the throughput conditions SIFTR can accommodate on similar hardware.

SIFTR was also evaluated on a second testbed consisting of much newer 2007-era dual core commodity PC hardware. With SIFTR running on this hardware, configured with maximum data logging granularity, up to 100 TCP flows can achieve aggregate throughput of at least 565Mbps over gigabit Ethernet with a worst case skip rate of 31.1 skipped packets per Mbps throughput.

Further work needs to be undertaken to investigate why testbed 2 demonstrated increased TCP throughput when SIFTR was enabled in the kernel, compared to the baseline tests.

The results presented in this report have highlighted areas for possible further optimisation within SIFTR, and also constitute ongoing further work.

VII. Acknowledgments

This report has been made possible in part by a grant from the Cisco University Research Program Fund at Community Foundation Silicon Valley.

The authors would also like to acknowledge the help of the FreeBSD community, namely Pawel Jakub Dawidek for creating the kernio code used in SIFTR, and the users on the freebsd-hackers mailing list for the help and support they provided us during the creation of the SIFTR software.

References

[1] "NewTCP project tools," June 2007, http://caia.swin.edu.au/urp/newtcp/tools.html.
[2] "The Web100 Project," June 2007, http://web100.org/.
[3] "The NewTCP Project," June 2007, http://caia.swin.edu.au/urp/newtcp.
[4] "The FreeBSD Project," June 2007, http://www.freebsd.org/.
[5] L. Stewart, J. Healy, "Tuning and Testing the FreeBSD 6 TCP Stack," CAIA, Tech. Rep. 070717B, July 2007, http://caia.swin.edu.au/reports/070717B/CAIA-TR-070717B.pdf.
[6] FreeBSD Hypertext Man Pages, "PFIL," June 2007, http://www.freebsd.org/cgi/man.cgi?query=pfil&sektion=9.
[7] L. Stewart, J. Healy, "An Introduction to FreeBSD 6 Kernel Hacking," CAIA, Tech. Rep. 070622A, July 2007, http://caia.swin.edu.au/reports/070717B/CAIA-TR-070717B.pdf.
[8] , "Iperf - The TCP/UDP Bandwidth Measurement Tool," June 2007, http://dast.nlanr.net/Projects/Iperf/.
[9] "FreeBSD Network Performance Project (netperf)," August 2007, http://www.freebsd.org/projects/netperf/.
[10] Robert Watson, "FreeBSD Network Performance Project (netperf)," August 2007, http://www.watson.org/~robert/freebsd/netperf/freebsd7.txt.