TCPivo: A High-Performance Packet Replay Engine

Wu-chang Feng   Ashvin Goel   Abdelmajid Bezzaz   Wu-chi Feng   Jonathan Walpole
OGI @ OHSU

ABSTRACT

This paper describes the design and implementation of a high performance packet replay tool called `TCPivo`. `TCPivo` is implemented on commodity hardware using widely available open-source software and can be used as a cost-effective means for evaluating the performance of networking devices. To achieve high throughput and accuracy, `TCPivo` employs novel mechanisms for managing trace files and accurate low-overhead timers. In addition, through the use of low-latency kernel patches and priority scheduling, `TCPivo` can be made highly resilient to background system load. Using these mechanisms, the tool is able to support packet replay at rates well above OC-3 on an x86-based server.

1. INTRODUCTION

In order to properly evaluate the design of network devices such as routers, switches, and firewalls, system architects can employ a wide-range of tools ranging from simulation to actual physical testing of the device. At one end of the spectrum, designers can evaluate their systems by completely simulating both the hardware itself and the network traffic being processed. Examples of this approach include the `ns` simulator [1] and the IXP network processor simulator [2]. While simulation offers a completely reproducible environment, it can be prohibitively slow and inaccurate [3]. Cycle-level simulations can be several orders of magnitude slower than the hardware itself and simulated traffic often does not have the same properties as actual traffic. Another approach for evaluating devices is to use the device itself and employ a traffic generator to emulate network load. For example, one could implement a network device using a network processor such as the IXP and evaluate it using a commercially available hardware packet generator such as those provided by IXIA [4]. This approach allows for evaluations to be done in real-time and in a reproducible manner. However, besides being expensive [1], such tools have questionable accuracy when emulating real traffic since they stochastically generate traffic versus replaying an actual trace. Stochastic generation misses important low-level features of traffic including address mixes, protocol mixes, and per-application traffic characteristics [5]. Packet generators such as these also can not reproduce the modulation of packets as they pass through numerous hops in the network. Features such as ACK compression [6] are commonly found in real traces, but are not easily reproduced using synthetic generators.

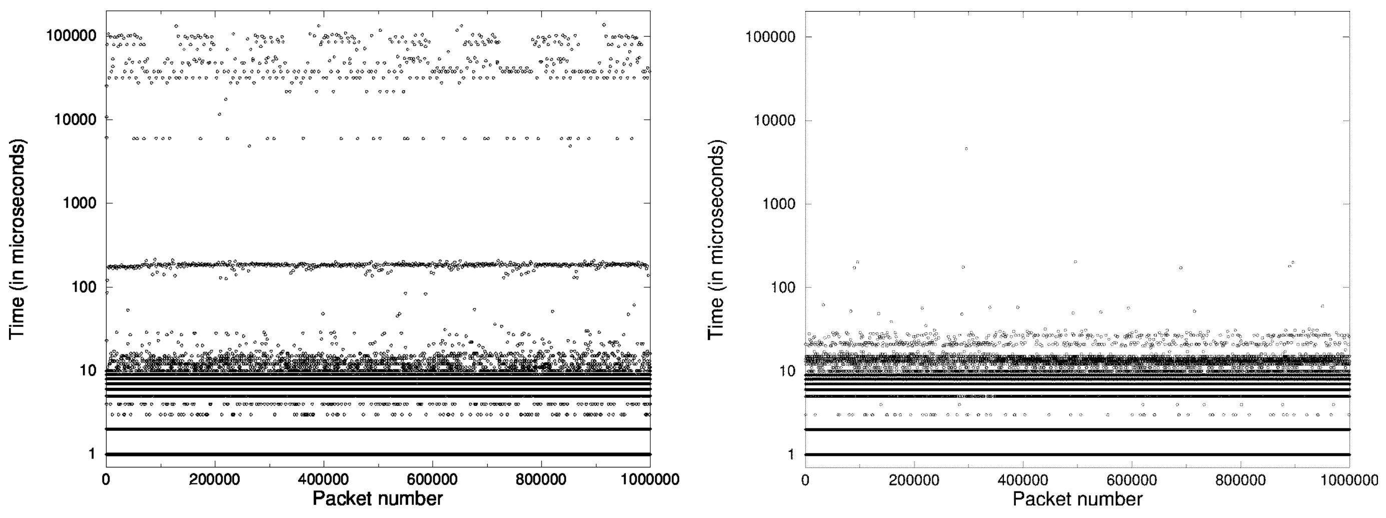
While simulation and synthetic emulation both have their uses, another approach is to take the actual device and to test it with a trace-driven packet generator. In this approach, a trace is collected and stored to disk using a tool such as `tcpdump` and then later replayed against a target device. When driven by a representative library of traces, such an approach is fast, reproducible, and highly accurate in terms of address mixes and packet loads, thus enabling designers to accurately test route caching architectures, packet classification algorithms, queue management algorithms, scheduling algorithms, and buffer provisioning in a very realistic environment. There are two things one needs to do such an evaluation: a high-performance packet collection engine and a high-performance packet replay engine. While tools for high-performance packet collection exist [7, 8], similar replay engines do not. In this paper, we describe the design of `TCPivo`, a high-performance replay engine that accurately reproduces traffic recorded from a variety of existing trace collection tools [9, 10, 8, 11].

2. TCPIVO DESIGN

There are many ways to build a high-performance packet replay tool. In the design of `TCPivo`, we constrain ourselves to commodity hardware (i.e. x86-based systems) and readily available, open-source software (i.e. Linux and its available patches). While specialized hardware and proprietary real-time operating systems can be used to build a much more powerful tool, our approach is cost-effective and allows users to run on pre-existing systems. Of the variety of open-source operating systems, we select Linux due to its level of hardware support and its widespread use. A key question in our approach is whether or not commodity hardware and software have the ability to accurately replay a packet stream at

ACM SIGCOMM 2003 Workshops August 25&27, 2003, Karlsruhe, Germany
Copyright 2003 ACM 1-58113-748-6/03/0008 ...$5.00.

[1]The pricing for an IXIA 1600 starts at $41,000



(a) First run (b) Second run

Figure 1: Read latency using `fread()`

Processor	1.8GHz Intel Xeon
Chipset	E7500
Memory	512MB DDR PC1600 SDRAM
FSB	400MHz
NIC	Intel 82544 1000Mbs
PCI	Full length 133MHz PCI-X
Disk	120GB 7200rpm UDMA/100
OS	Linux 2.4.20-pre1
File System	ext3

Table 1: Evaluation server

high-speed. With recent advances in processor and network technology, it is clear that the raw hardware power exists. However, it is not clear whether or not current software and operating systems can take full advantage of it.

There are four major issues that must be addressed in building the replay engine. The first issue is the prefetching and I/O management of the trace itself. The trace must be efficiently and predictably retrieved from storage (presumably a disk) in real-time. The second issue is the time-triggering of packet send events. To accurately reproduce packet inter-arrival times recorded in the trace, the system must have an accurate, low-overhead mechanism for timing packet sends. The third issue is the sending path itself. To achieve high-performance, the send path must be as fast as possible. The final issue is the scheduling of the process in relation to other applications and system activities. `TCPivo` must be allowed to run immediately whenever packets need to be sent. Driven by these individual design issues, the following section describes the mechanisms employed and demonstrates their efficacy.

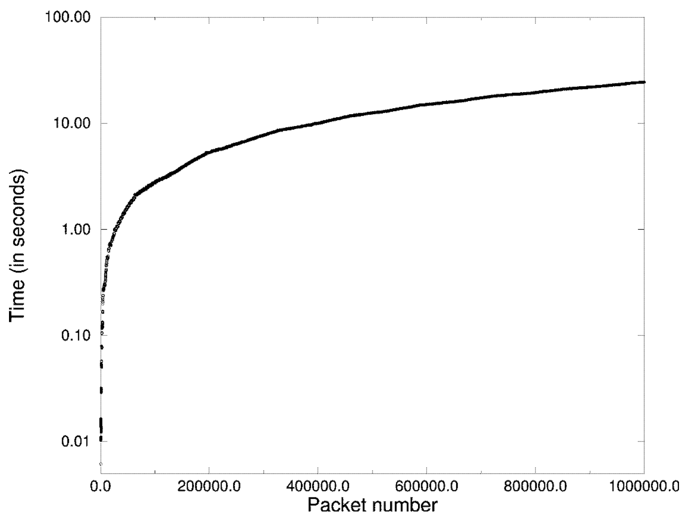
3. TCPIVO

In the following section, we describe the design decisions behind `TCPivo` and evaluate them using a set of fixed workloads. The evaluation of `TCPivo` was performed on an Intel-based server described in Table 1. To evaluate the performance of the approaches, we measured the difference between the time a packet should have been sent and the time it was actually sent using the `gettimeofday()` system call. In the benchmarks on our system, this call took on average, $1.16\mu s$ to run. The measurement of the packet's sending time was taken immediately after the sending system call (`sendto()`), thus introducing a slight measurement error equal to the time it took to perform the call. As shown in Section 3.3, this latency was about $5\mu s$ (microseconds). For all of our experiments, the workload used consisted of several fixed-interval, 64MB, 1 million packet traces [2]. The trace files used contained per-packet timestamps, MAC headers, and TCP/IP headers. In addition, each trace consisted of a continuous stream of packets sent at a periodic interval δ. Using these workloads, we then evaluated the tool by measuring the difference between the time a packet in the trace should have been sent and the time it was actually sent. We refer to this difference as the send-time error and denote it as ϵ.

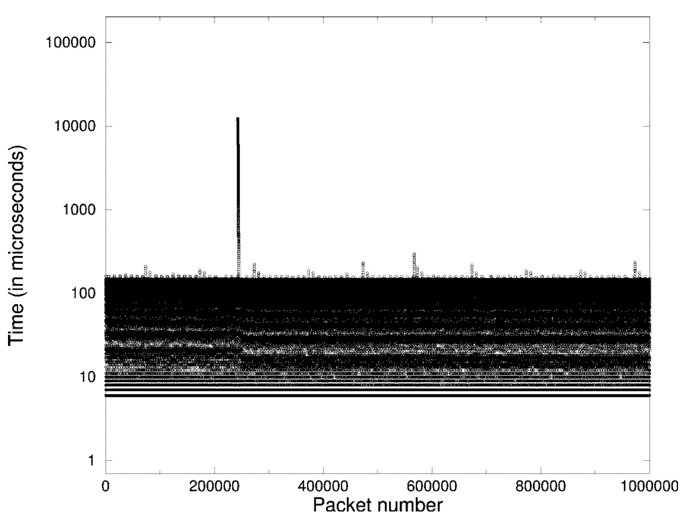
3.1 Trace file management

To properly send packets from the trace, the tool must first ensure that the trace data is readily available when packets are to be sent. In particular, packet information and packet sending times must be prefetched from the file system and made available to the replay engine just before it is needed. While a multithreaded design can solve this problem by employing a thread to do the prefetching, modern file systems typically do some form of predictive prefetching in order to minimize the latency of file I/O. To evaluate the effectiveness of the Linux's prefetching mechanism, we examined its ability to efficiently read the trace. Figure 1(a) shows the `fread()` latency for each packet in the

[2]The tool supports the DAG format along with other trace file formats such as `tcpdump` and TSH via freely available conversion software [11]

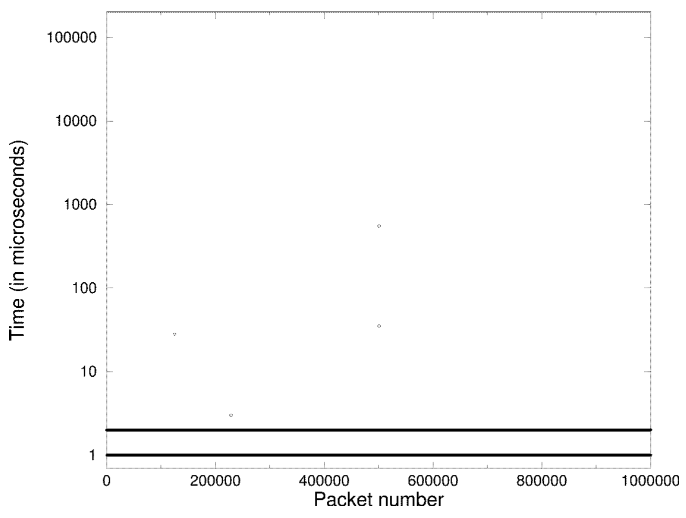


(a) First run with sends

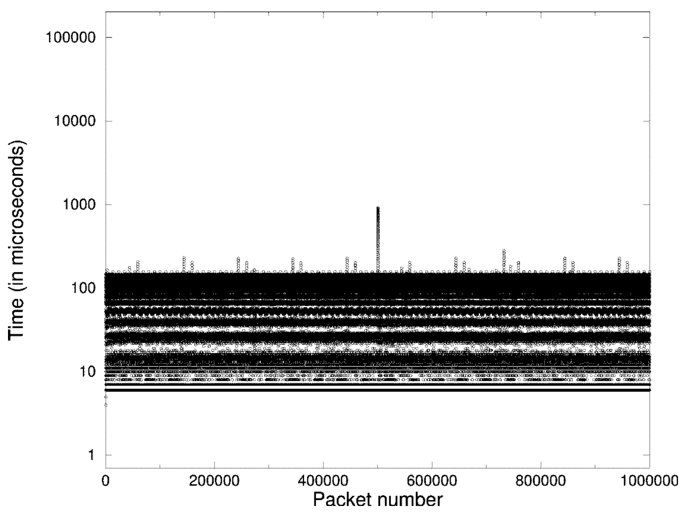


(b) Second run with sends

Figure 2: ϵ using `fread()` with δ=20μs



(a) Read latency



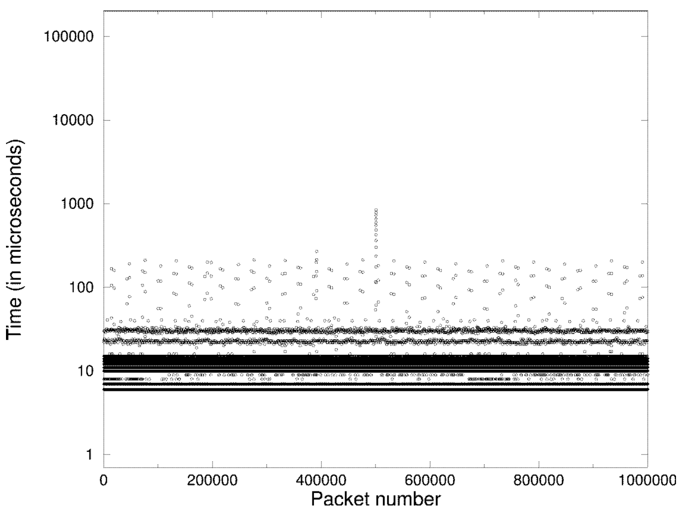
(b) ϵ with δ=20μs

Figure 3: Performance using `madvise()`

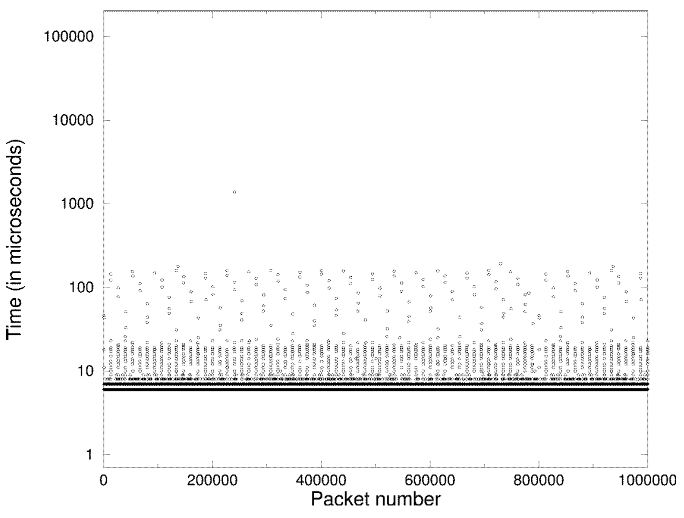
64MB, 1 million packet trace. The latency is measured via `gettimeofday()` calls placed before and after the `fread()`. As the figure shows, the `fread()` latency has extremely high variability and very high peaks of over $100ms$, indicating that the file system is doing an unacceptably poor job in prefetching from the disk. One way to address this problem would be to "warm up" the file cache with the trace file before running the experiment. Figure 1(b) shows the `fread()` latency when run a second time. As the figure shows, the latency is relatively fixed and is on the order of $10\mu s$, a clear indication that the contents of the trace have been already prefetched and stored in memory. There are, however, some rather large spikes in the figure. As we will show in Section 3.4, these spikes are caused by long, non-preemptible paths that are present in the default Linux kernel.

To illustrate the impact of the file system when using a real replay workload, we replayed a large trace with a fixed sending interval δ of $20\mu s$. While there are many ways to implement the timing functions, in this experiment, the tool employed a polling loop that continually invokes `gettimeofday()` until the send-time has been reached. Figure 2(a) plots the send-time error ϵ when the experiment is performed for the first time. As the figure shows, the latency induced by the file system prevents the tool from keeping up as indicated by the monotonically increasing error. Figure 2(b) shows the send-time error when the experiment is run the second time. As the figure shows, the tool is able to handle the workload, reproducing the packet stream somewhat more accurately.

The above experiments indicate the utility of precaching the trace file into memory. Unfortunately, such an approach only works for small trace files that fit into the file cache. Because trace files can be much larger than resident memory, it is clear that some form of active, on-the-fly prefetch-



(a) δ=70μs



(b) δ=2500μs

Figure 4: ϵ using a polling loop

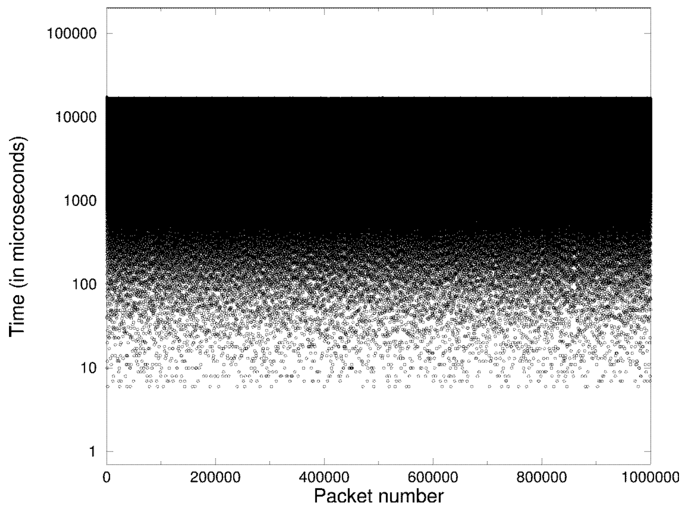
δ	Polling	`usleep()`	Firm timers
70μs	78%	40%	19%
2500μs	99%	4%	1%

Table 2: CPU utilization of different approaches

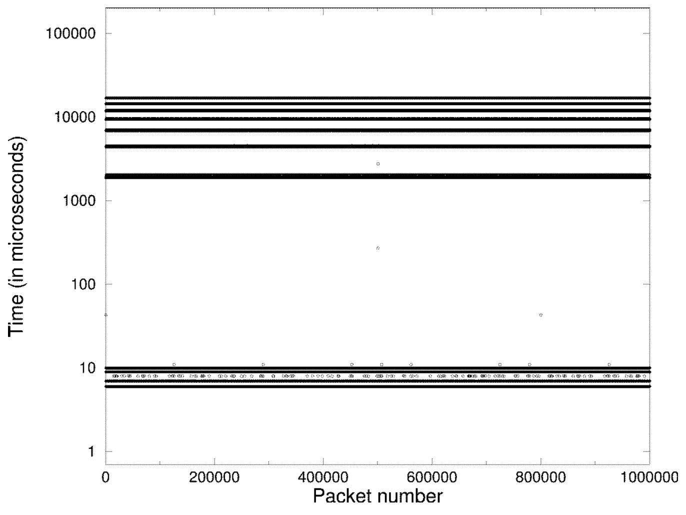
ing is required. One way to do this is to use a separate thread which actively loads the trace into memory. While this can effectively perform the task, it consumes processing resources and requires a locking mechanism to synchronize access to the shared memory. Both can limit the maximum replay throughput supported. A more effective way for doing this is to memory map upcoming parts of the trace using `mmap()` and then, using the `madvise()` call, to pass a behavioral hint to the kernel in order to allow it to aggressively DMA the parts of the trace into memory in the background. In this case, since the trace is stored contiguously, passing a sequential access hint is all that is required. The use of `mmap()` and `madvise()` is efficient and can be done within the logic of the main sending loop. In our implementation, a double-buffered approach is used where one buffer is used to prefetch the next part of the file while the other is being actively accessed. The size of each buffer was set to 32MB. Figure 3(a) shows the read latency using this approach. As the figure shows, the latency is extremely small and very predictable. Figure 3(b) plots ϵ for the `madvise()` approach using the same workload as before, but with a double-buffered prefetching implementation. As the figure shows, the file system is able to efficiently and predictably supply the tool with packets from the trace even at a δ of 20μs. Note that while the tool is able keep up with the workload, accuracy clearly degrades as the system is pushed to its limits. The sensitivity of the system at this point is evident in the increased packet jitter and in the small spike in latency observed half-way through the experiment. This spike is caused by calling `mmap()` to prefetch the next block of the trace.

3.2 Timers and timing

In the previous experiments, the tool employed a polling loop that continually invoked `gettimeofday()` until the packet time was reached. This approach, while affording an extremely high degree of accuracy, is extremely CPU-intensive. To avoid this resource consumption, one could use the `usleep()` system call to put the sending process to sleep until the send time of the next packet. This approach saves a large amount of CPU time, but it does so at the expense of accuracy. In most cases, `usleep()` employs a timer mechanism which is triggered by a periodic tick interrupt. On x86-based machines this interrupt is generated by the Programmable Interval Timer (PIT) and has a period of 10ms. As a result, based on where the send-time falls on this interval, it can experience up to 10ms of jitter. Figure 4, Figure 5, and Table 2 show the send-time error ϵ and the CPU utilization of the tool using a polling implementation and a `usleep()` implementation. The experiments were run using fixed sending intervals of δ=70μs and δ=2500μs. To capture CPU utilization, the `time` program was used. While this facility can be somewhat error-prone [12], modern implementations reflect performance more accurately and are good for qualitative comparisons such as ours. As the figures and table show, the use of `usleep()` forces one to give up accuracy in return for efficiency. Using `usleep()`, the CPU utilization remains extremely low while ϵ fluctuates wildly due to the 10ms precision of the PIT. An interesting point to note is that when the real-time priority of a process is set in Linux 2.4-based kernels, the implementation of `usleep()` changes. For sleep times that are under 2ms, `usleep()` employs a polling loop to implement them. Figure 6 shows ϵ upon re-running the `usleep()` experiments with the real-time priority set. As the figure shows, the accuracy of the tool matches those of the polling implementation shown in Figure 4(a). As before, the CPU utilization of the tool jumps to 78%, also matching



(a) δ=70μs



(b) δ=2500μs

Figure 5: ϵ using `usleep()`

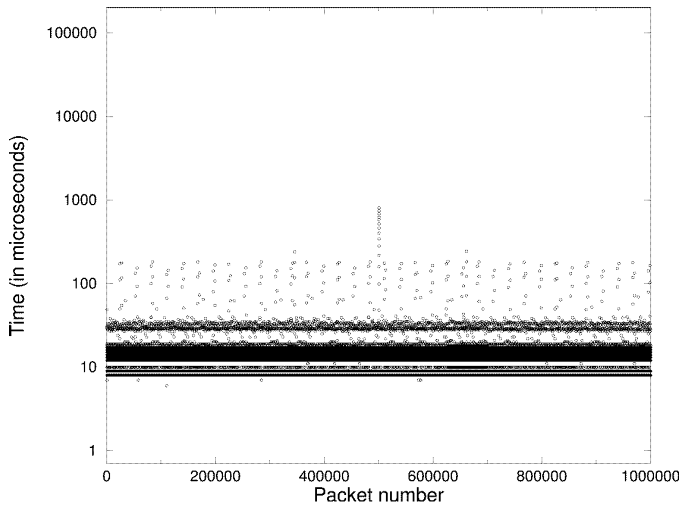


Figure 6: ϵ using `usleep()` at real-time priority δ=70μs

Task	Average time spent
Main loop	9.38μs
Data padding	1.45μs
Checksum calculation	1.27μs
`sendto()`	5.16μs
Trace read	1.30μs

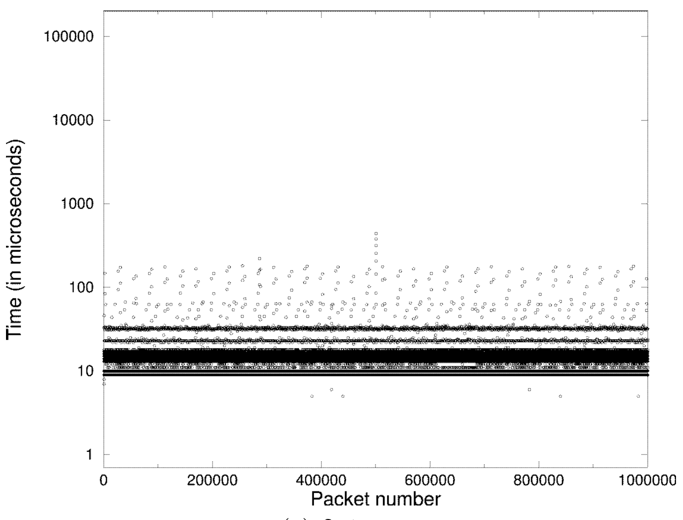
Table 3: Profile of sending loop

that of the polling implementation.

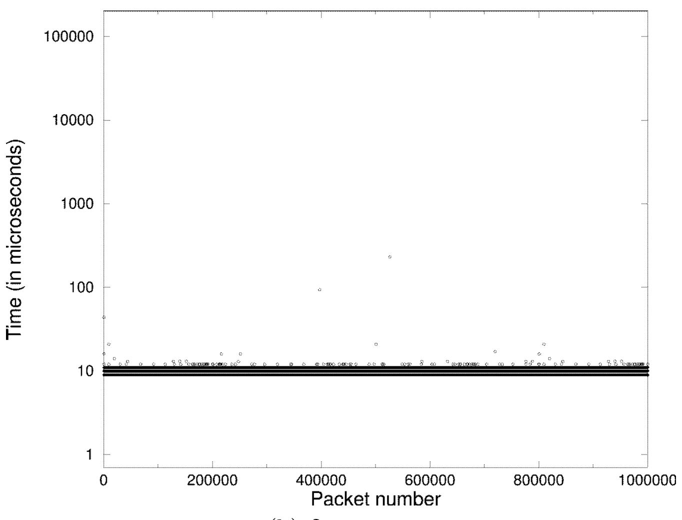
It is clear that decreasing the periodic tick interval would increase the accuracy of the timer immensely, however, doing so would also increase the interrupt handling overhead as well. To address this, modern real-time operating systems provide high resolution timers based on an aperiodic interrupt source [13]. For x86-based systems, the PIT or the CPU APIC (Advanced Programmable Interrupt Controller) can be programmed to generate aperiodic interrupts to support high-performance, time-sensitive applications such as `TCPivo`. Figure 7 shows the performance of the same workload using firm timers [14, 15]. Firm timers use a combination of the periodic PIT and an aperiodic APIC timer [16] to accurately and efficiently implement `usleep()`. Figure 7 and Table 2 show ϵ and the CPU utilization using firm timers over the two workloads. As the figure shows, this implementation achieves the best of both worlds. It retains the accuracy of the polling approach while matching the efficiency of `usleep()` across all time intervals. In addition, the figure also shows that as the packet inter-arrival times decrease towards the system limit, the timing becomes less accurate.

3.3 Packet transmission

An efficient packet transmission loop is essential in supporting high-speed network replay. To make sending more efficient and to support playback at speeds faster than the disk's speed, `TCPivo` supports the ability to replay packets without the original payload by replacing the payload with an equal amount of null-padded data instead. Because one of the applications for this tool is to evaluate routers and switches, it is often not necessary to send the original packet payloads with the packets themselves. For all of the results in this paper, this option was used. Padding the payload allows the sending loop to be much faster, as full packet payloads do not need to be read from the file system. As a side-effect of using dummy data, however, a new, valid checksum must be calculated for the packet at run-time. Table 3 shows the latency profile of the tool using `madvise()`, firm timers, and padded payloads. The latencies were calculated by instrumenting a single component at a time with `gettimeofday()` calls across a large number of packet sends. As shown in the table, the main sending loop takes, on average, less than 10μs. Assuming MTU-sized packets, this

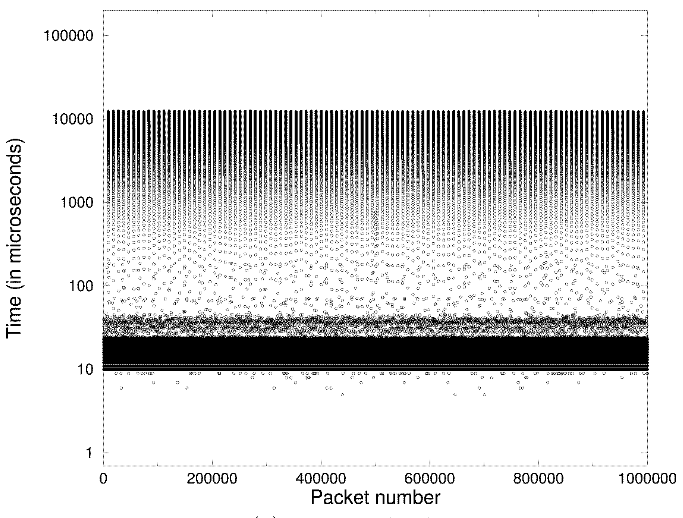


(a) δ=70μs

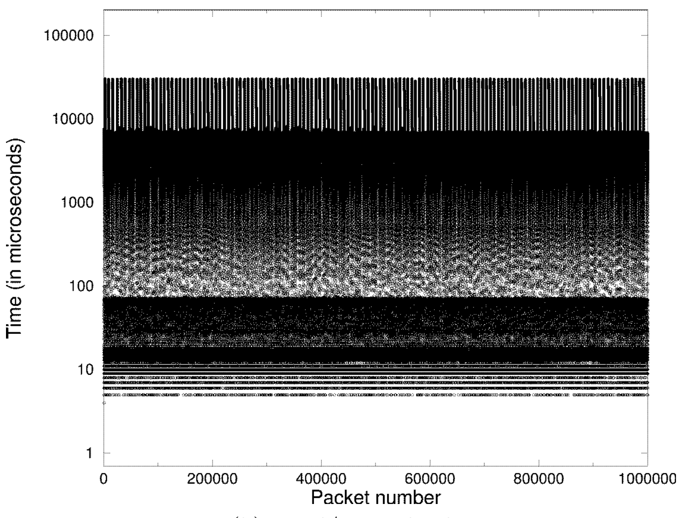


(b) δ=2500μs

Figure 7: ϵ using firm timers



(a) Memory load



(b) Read/Write load

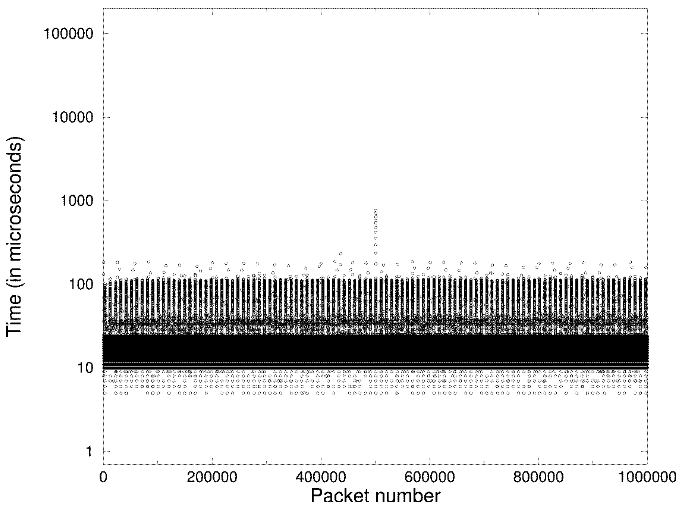
Figure 8: ϵ using firm timers with background load and δ=70μs

allows TCPivo to nearly saturate the gigabit ethernet link of our test system. When broken down further into components, most of the latency of the loop occurs with the call to send the packet with data padding and checksum computation each taking under 2μs to perform. As a further optimization, TCPivo supports a trace pre-processing step that pre-computes the checksum assuming padded data in order to reduce the latency of the sending loop further.

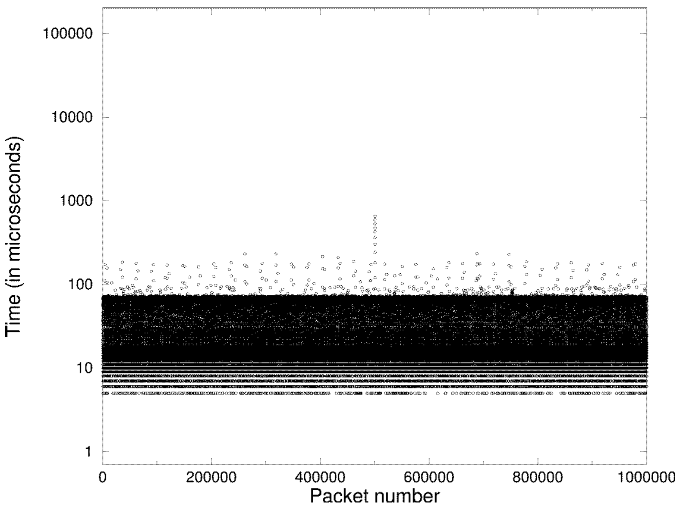
3.4 Scheduling

The previous experiments were all run using a dedicated server with very little competing load. Since in practice, there could be many other services and processes running, it is important that the tool perform predictably under load. The first step to ensuring correct behavior is to use the real-time scheduling priorities of Linux. While this can ensure TCPivo gets priority over other user processes, the tool is still at the mercy of the kernel which must schedule it to run at the right time. Unfortunately, since Linux 2.4-based systems contain relatively long, non-preemptible paths within the kernel, it is often the case that the tool can not run when it needs to. Figure 8 demonstrates this problem by re-running the experiment in Figure 7 using real-time priorities, firm timers, and δ=70μs, but with two different background loads. The first background load consists of a memory stress test in which a user process continuously reads and writes a buffer of 128MB to generate page faults. The second background load consists of a file system stress test in which a user process continuously reads and writes a file of 8MB. Figure 8 shows the results under these two loads. As the figures show, in both cases, the addition of this low priority, background load is enough to completely disrupt the performance of the tool.

To address this issue, several patches to the Linux kernel



(a) Memory load



(b) Read/Write load

Figure 9: ϵ using low-latency kernel and firm timers with background load and δ=70μs

have been developed. Among them are a preemptible kernel patch and a low-latency kernel patch [17, 18]. By making the kernel preemptible or by reducing the longest non-preemptible path through it, the tool can be made much more predictable. Figure 9 shows the performance of the same experiments using the low-latency patch to Linux. As the figure shows, although the added load still impacts packet replay, reducing the length of non-preemptible paths in the kernel allows it to maintain more accurate timing.

3.5 Overall accuracy on-the-wire

Up until now, we have only focused on everything up to the sending event. However, significant latency and jitter can occur between when the application issues the send and when the packet actually hits the wire. For example, the kernel may be pre-empted immediately after the sending event or buffering may occur at the network interface, causing the packet to hit the wire well after the application issues it. Since it is difficult to instrument the underlying device drivers and network cards to obtain actual sending times on the wire, we instead use a second machine on the same network to monitor the generated packet stream. In our experiment, an identical receiver machine as described in Table 1 is connected via a dedicated VLAN to the sender. Figure 10 shows the inter-packet arrival times measured at the receiver when the sender replays a workload of packets with inter-packet arrivals of δ=70μs. As the figure shows, additional jitter is experienced as the packets are modulated through the device drivers on both ends and network itself. For the most part, however, the timing remains tight. As part of future work, we plan on investigating sources of this error and possible fixes for it.

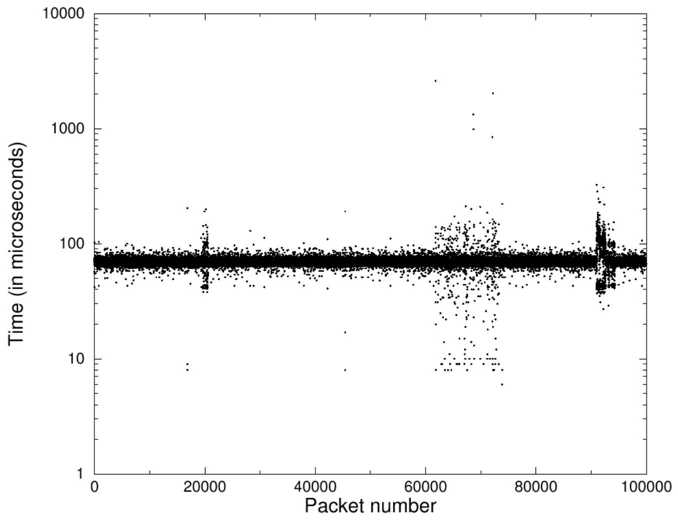


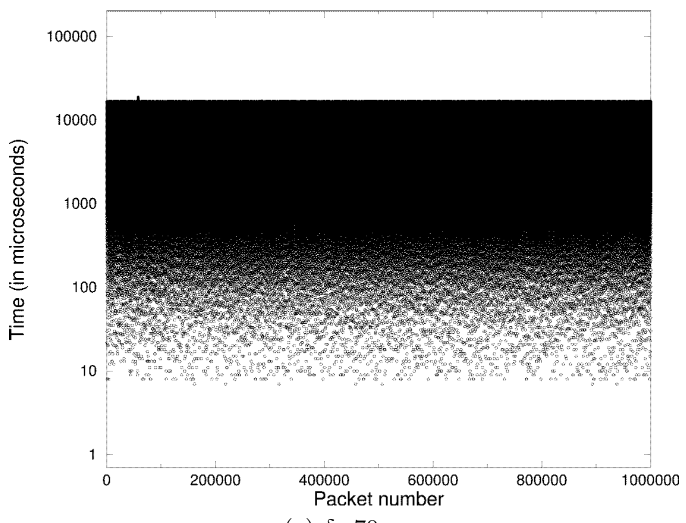
Figure 10: Inter-packet arrival times at receiver δ=70μs

4. RELATED WORK

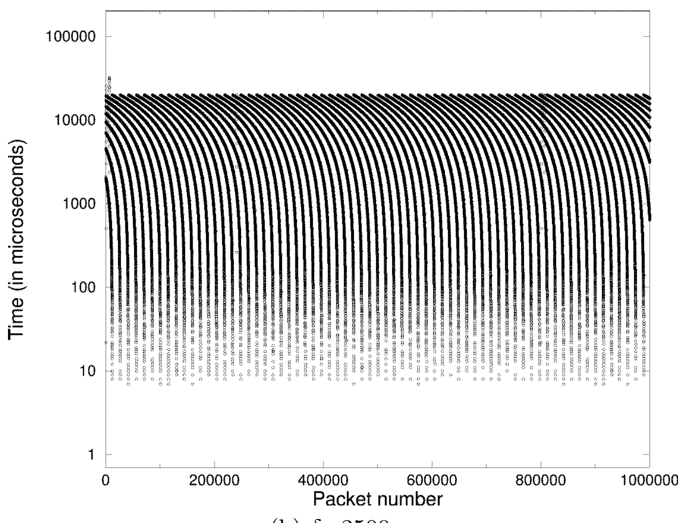
While we were finishing the development of `TCPivo`, another packet replay engine, `tcpreplay`, was released [19]. `tcpreplay` performs a similar function as `TCPivo`, however, many of the performance issues addressed in `TCPivo` are not addressed in `tcpreplay`. To demonstrate the impact of our design, we instrumented the latest version of `tcpreplay` (1.2a) with timing code and examined its performance. Figure 11 shows its performance using the 70μs and the 2500μs workloads used in the previous section. As the figure shows, the combination of poor file system prefetching and inaccurate timers adversely impacts the performance of the tool.

5. CONCLUSION

By addressing the issues of trace prefetching and timers and by employing a low-latency kernel, `TCPivo` is able to accurately replay network traces at high-speed on commodity hardware. As part of future work, we are continually working towards making the tool more accurate. To this end, we are working on alleviating issues such as PCI bus



(a) δ=70μs



(b) δ=2500μs

Figure 11: ϵ using `tcpreplay`

arbitration and PCI/NIC buffering that can impact the accuracy of the replayed stream. Once completed, we plan on releasing the source code for `TCPivo` [20] as well as all of the associated kernel patches. In addition, we hope to work with the developers of `tcpreplay` to merge the performance features of `TCPivo` into `tcpreplay`. Finally, we plan on pursuing additional mechanisms that will allow for multi-gigabit replay including removing unnecessary data copying via IO-Lite [21], examining kernel modifications for improving performance [22, 23], and examining the parallelization of the replay task itself on SMP servers.

6. REFERENCES

[1] S. McCanne and S. Floyd, "`http://www.isi.edu/nsnam/ns/`," ns Network Simulator, 1996.

[2] E. Johnson and A. Kunze, "IXP1200 Programming," 2002.

[3] V. Paxson and S. Floyd, "Why we don't know how to simulate the internet," in *Winter Simulation Conference*, 1997, pp. 1037–1044.

[4] Ixia, "Ixia 1600 Network Traffic Generator," `http://www.ixiacom.com/`.

[5] W. Feng, F. Chang, W. Feng, and J. Walpole, "Provisioning On-line Games: A Traffic Analysis of a Busy Counter-Strike Server," in *Internet Measurement Workshop*, November 2002.

[6] L. Zhang, S. Shenker, and D. Clark, "Observations and Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic," in *Proceedings of ACM SIGCOMM*, August 1991, pp. 133–148.

[7] J. Hay, W. Feng, and M. Gardner, "Capturing Network Traffic with a MAGNeT," in *Linux Showcase and Converence (ALS'01)*, 2001, pp. 259–270.

[8] NLANR, "Passive Measurement and Analysis," `http://pma.nlanr.net/PMA/`.

[9] tcpdump developers, "tcpdump/libpcap Public Repository," `http://www.tcpdump.org`, 2002.

[10] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," in *Proc. of Winter USENIX*, 1993.

[11] DAG developers, "DAG software," `http://dag.cs.waikato.ac.nz/dag/dag-soft.html`.

[12] S. McCanne and C. Torek, "A randomized sampling clock for CPU utilization estimation and code profiling," in *USENIX Winter*, 1993, pp. 387–394.

[13] S. Savage and H. Tokuda, "RT-Mach Timers: Exporting Time to the User," in *USENIX Mach Symposium*, April 1993.

[14] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole, "Supporting Time-sensitive Applications on General-Purpose Operating Systems," 2002.

[15] M. Aron and P. Druschel, "Soft Timers: Efficient Microsecond Software Timer Support for Network Processing," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 197–228, 2000.

[16] U. Walter, "APIC Timer Module for Linux," `http://www.oberle.org/apic_timer.html`, 2003.

[17] R. Love, "The Linux Kernel Preemption Project," `http://kpreempt.sourceforge.net/`.

[18] A. Morton, "Linux scheduling latency," `http://www.zip.com.au/~akpm/linux/schedlat.html`.

[19] tcpreplay developers, "tcpreplay," `http://tcpreplay.sourceforge.net`, July 2002.

[20] TCPivo developers, "TCPivo," `http://www.cse.ogi.edu/sysl/projects/tcpivo`, August 2002.

[21] V. Pai, P. Druschel, and W. Zwaenepoel, "IO Lite: A Unified I/O Buffering and Caching System," 1999.

[22] CITI, "The Linux Scalability Project," `http://www.citi.umich.edu/projects/linux-scalability`, 2002.

[23] A. Goel, "Time-Sensitive Linux," `http://www.cse.ogi.edu/sysl/`.