17th ACM Symposium on Operating Systems Principles (SOSP ’99)
Published as *Operating Systems Review*, **34**(5):232–246, Dec. 1999

Soft timers: efficient microsecond software timer support for network processing

Mohit Aron   Peter Druschel
Department of Computer Science, Rice University
{*aron,druschel*}*@cs.rice.edu*

Abstract

This paper proposes and evaluates soft timers, a new operating system facility that allows the efficient scheduling of software events at a granularity down to tens of microseconds. Soft timers can be used to avoid interrupts and reduce context switches associated with network processing without sacrificing low communication delays.

More specifically, soft timers enable transport protocols like TCP to efficiently perform rate-based clocking of packet transmissions. Experiments show that rate-based clocking can improve HTTP response time over connections with high bandwidth-delay products by up to 89% and that soft timers allow a server to employ rate-based clocking with little CPU overhead (2–6%) at high aggregate bandwidths.

Soft timers can also be used to perform network polling, which eliminates network interrupts and increases the memory access locality of the network subsystem without sacrificing delay. Experiments show that this technique can improve the throughput of a Web server by up to 25%.

1 Introduction

We propose and evaluate *soft timers*, an operating system facility that allows efficient scheduling of software events at microsecond (μsec) granularity.

The key idea behind soft timers is to take advantage of certain states in the execution of a system where an event handler can be invoked at low cost. Such states include the entry points of the various OS kernel handlers, which are executed in response to system calls, exceptions (TLB miss, page fault, arithmetic) and hardware interrupts. In these “trigger states”, the cost of saving and restoring of CPU state and the shift in memory access locality associated with the switch to kernel mode have already been incurred; invoking an additional event handler from the trigger state amortizes this overhead over a larger amount of useful computation.

Of course, the times at which a system enters a trigger state are unpredictable and depend on the workload. Therefore, soft timers can schedule events only probabilistically: A soft timer event may be delayed past its scheduled time by a random but bounded amount of time. In practice, trigger states are reached often enough to allow the scheduling of events at intervals down to a few tens of μsecs, with rare delays up to a few hundred μsecs. As we will show, soft timers allow the scheduling of events at these intervals with very low overhead, while the use of a conventional hardware interrupt timer at the same rate would result in unacceptable overhead on the system.

We explore the use of a soft timers facility to perform two optimizations in the network subsystem. Soft timers enable a transport protocol like TCP to efficiently perform *rate-based clocking*, i.e., to transmit packets at a given rate, independent of the arrival of acknowledgment (ACK) packets. Rate-based clocking has been proposed as a technique that improves the utilization of networks with high bandwidth-delay products [25, 18, 1, 10, 5]. Our experiments show that a Web server that employs rate-based clocking using soft timers can achieve up to 89% lower response time than a server with a conventional TCP over networks with high bandwidth-delay product.

A second optimization is *soft timer based network polling*. Here, soft timer events are used to poll the network interface, thus avoiding interrupts. Experiments show that the performance of a Web server using this optimization can increase by up to 25% over a conventional interrupt based implementation.

The rest of this paper is organized as follows. In Section 2, we provide background and motivation for this work. The soft timers facility is presented in Section 3. Applications of soft timers are discussed in Section 4. We present empirical results obtained with a prototype implementation of soft timers in Section 5, discuss related work in Section 6 and conclude in Section 7. Background information on the need for rate-based clocking can be found in the Appendix.

2 Background and motivation

Modern CPUs increasingly rely on pipelining and caching to achieve high performance. As a result, the speed of program execution is increasingly sensitive to unpredictable control transfer operations. Interrupts and context switches are particularly expensive, as they require the saving and restoring of the CPU state and entail a shift in memory access locality. This shift typically causes cache and TLB misses in the wake of the entry and the exit from the interrupt handler, or the context switch, respectively.

The cost of interrupts and context switches is generally not a concern as long as they occur on a millisecond (msec) timescale. For instance, disk interrupts, conventional timer interrupts used for time-slicing and the associated context switches typically occur at intervals on the order of tens of msecs.

However, high-speed network interfaces can generate interrupts and associated context switches at intervals on the order of tens of μsecs. A network receive interrupt typically entails a context switch to a kernel thread that processes the incoming packet and possibly transmits a new packet. Only after this thread finishes is the activity that was originally interrupted resumed.

As we will show, these interrupts and context switches can have a significant impact on the performance of server systems performing large amounts of network I/O. Even a single Fast Ethernet interface can deliver a full-sized packet every 120μsecs and Gigabit Ethernet is already on the market. Moreover, many high-end Web servers already have backbone connections to the Internet at Gigabit speed.

2.1 Rate-based clocking

Achieving high network utilization on networks with increasingly high bandwidth-delay products may require transport protocols like TCP to perform *rate-based clocking*, that is, to transmit packets at scheduled intervals, rather than only in response to the arrival of acknowledgment (ACK) packets.

Current TCP implementations are strictly *self-clocking*, i.e., packet transmissions are paced by the reception of ACK packets from the receiver. Adding the ability to transmit packets at a given rate, independent of the reception of ACK packets (*rate-based clocking*), has been proposed to overcome several known shortcomings of current TCP implementations:

• Rate-base clocking can allow a sender to skip the slow-start phase in situations where the available network capacity is known or can be estimated. This can lead to significantly increases in network utilization and achieved throughput, particularly when traffic is bursty and the network's bandwidth-delay product is high. Such conditions arise, for instance, with Web (HTTP) traffic in today's Internet [25, 18].

• Rate-based clocking can overcome the effects of *ACK compression* and *big ACKs*. Either phenomenon may cause a self-clocked sender to transmit a burst of packets in close succession, which can adversely affect network congestion.

• Rate-based clocking allows a TCP sender to shape its traffic in integrated services networks [10].

Rate-based clocking requires a protocol implementation to transmit packets at regular intervals. On high-bandwidth networks, the required intervals are in the range of tens to hundreds of μsecs. For instance, transmitting 1500 byte packets at 100Mbps and 1Gbps requires a packet transmission every 120 μsecs and 12 μsecs, respectively. Server systems with high-speed network connections transmit data at these rates even in today's Internet. As we will show in Section 3, conventional facilities for event scheduling available in general-purpose operating systems today cannot efficiently support events at this granularity. A more detailed discussion of the need for rate-based clocking can be found in Appendix A.

To summarize this section, interrupts and context switches are increasingly costly on modern computer systems. At the same time, high-speed network interfaces already generate interrupts and associated context switches at a rate that places a significant burden on server systems. Rate-based clocking in transport protocols, which has been proposed as a technique to increase network utilization and performance on high-speed WANs, necessitates even more interrupts when implemented using conventional timers.

In the following section, we present the design of the soft timers facility, which enables efficient rate-based clocking and can be used to avoid network interrupts.

3 Design of the soft timers facility

In this section, we present the design of soft-timers, a mechanism for scheduling fine-grained events in an operating system with low overhead.

Conventional timer facilities schedule events by invoking a designated handler periodically in the context of an external hardware interrupt. For example, an Intel 8253 programmable interrupt timer chip is usually supplied with a Pentium-based CPU. The former can be programmed to interrupt the processor at a given frequency.

Unfortunately, using hardware interrupts for fine-grained event scheduling causes high CPU overhead for the following reasons:

• On a hardware interrupt, the system has to save the context of the currently executing program and, after executing the interrupt handler, restore the interrupted program's state.

• Hardware interrupts are usually assigned the highest priority in the operating system. Thus, irrespective of the process currently running on the CPU, the interrupt handler is allowed to interrupt the execution of the former. In general, the data and instructions touched by the interrupt handler are unrelated to the interrupted entity, which can adversely affect cache and TLB locality.

In summary, the overhead of saving state, restoring state and the cache/TLB pollution associated with interrupts limits the granularity at which a conventional facility can schedule events. In Section 5 we show that the total cost of a timer interrupt in a busy Web server amounts to on average 4.45 μsecs on a 300MHz Pentium-II machine running FreeBSD-

2.2.6. This cost is insignificant when interrupts are being generated every msec but it is unacceptable when interrupts need to be generated (say) every 20 μsecs.

The key idea behind soft timers is as follows. During normal operation, a system frequently reaches states in its execution where an event handler could be invoked with minimal overhead. Examples of such opportune *trigger states* are

- at the end of executing a system call, just before returning to the user program,
- at the end of executing an exception handler, such as the ones triggered by a memory exception (e.g., TLB[1] or page fault) or an arithmetic exception (e.g., divide-by-zero),
- at the end of executing an interrupt handler associated with a hardware device interrupt, just before returning from the interrupt,
- whenever a CPU is executing the idle loop.

In these trigger states, invoking an event handler costs no more than a function call and no saving/restoring of CPU state is necessary. Furthermore, the cache and TLB contents in these trigger states have already been disturbed due to the preceding activity, potentially reducing the impact of further cache pollution by the event handler. Performance results presented in Section 5 confirm this reasoning.

Whenever the system reaches one of the trigger states, the soft-timer facility checks for any pending soft timer events and invokes the associated handlers when appropriate. As such, the facility can execute pending events without incurring the cost of a hardware timer interrupt. Checking for pending soft timer events in a trigger state is very efficient: it involves reading the clock (usually a CPU register) and a comparison with the scheduled time of the earliest soft timer event [2]. As we will show, performing this check whenever the system reaches a trigger state has no noticeable impact on system performance.

A disadvantage of the soft-timer facility is that the time at which an event handler is invoked may be delayed past its scheduled time, depending on how much time passes between the instant when a soft timer event becomes due and the instant when the system reaches a trigger state.

The maximal delay experienced by a soft timer event is bounded, because the soft timer facility still schedules a periodic hardware interrupt that is used to schedule any overdue events. The key point to notice is that as long as a system reaches trigger states with sufficient frequency, the soft timer facility can schedule events at much finer granularity than would be feasible using a periodic hardware interrupt.

Results presented in Section 5 show that a 300Mhz Pentium II system running a variety of workloads reaches trigger states frequently enough to allow the scheduling of soft-timer events at a granularity of tens of μsecs.

[1] In some architectures (e.g., Pentium), TLB misses are handled in hardware; in these machines, TLB faults cannot be used as trigger states.

[2] A modified form of timing wheels [24] is used to maintain scheduled soft timer events.

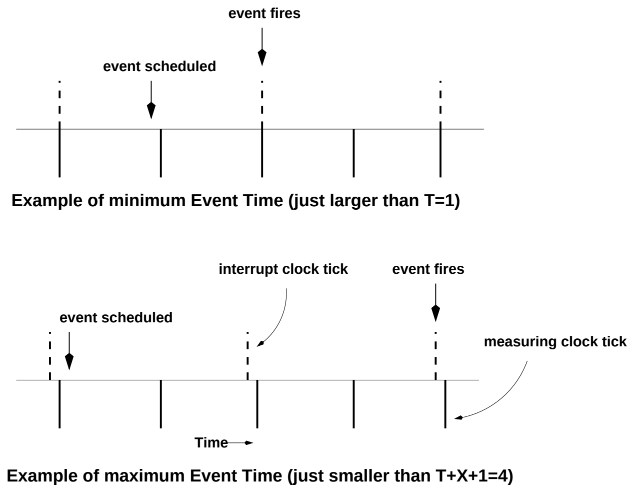
event fires
event scheduled
Example of minimum Event Time (just larger than T=1)
interrupt clock tick
event fires
event scheduled
measuring clock tick
Time
Example of maximum Event Time (just smaller than T+X+1=4)

Figure 1. Lower and upper bounds for event scheduling

The soft-timer facility provides the following operations.

- `measure_resolution()`. Returns a 64-bit value that represent the resolution of the clock (in Hz).
- `measure_time()` returns a 64-bit value representing the current real time in ticks of a clock whose resolution is given by `measure_resolution()`. Since this operation is intended to measure time intervals, the time need not be synchronized with any standard time base.
- `schedule_soft_event(T, handler)`: schedules the function *handler* to be called at least T ticks in the future (the resolution of T is specified by `measure_resolution()`).
- `interrupt_clock_resolution()`: gives the expected minimal resolution (in Hz) at which the facility can schedule events and equals the frequency of the system's periodic timer interrupt, which is used to "back up" soft timers.

The soft timer facility fires an event (by calling the corresponding handler) when the value returned by `measure_time()` exceeds the value stored at the time the event was scheduled by at least $T + 1$ (the increment by one accounts for the fact that the time at which the event was scheduled may not exactly coincide with a clock tick). If X is the resolution of the interrupt clock relative to the measurement clock (i.e., $X \equiv measure\_resolution()/interrupt\_clock\_resolution()$), then the soft timer facility puts the following bounds on the actual time (in ticks of the measurement clock) when the event fires:

$$T < Actual\ Event\ Time < T + X + 1$$

Figure 1 gives examples of the above bounds when $T = 1$ and $X = 2$. It is to be noted that the increment by one is negligible if the measurement clock is significantly finer than the interrupt clock (as is the case in most modern systems).

The reason for the upper bound is that the soft-timer facility uses a periodic timer interrupt to check for overdue soft-timer events. However, the actual time at which the handler is invoked is likely to be much closer to T. Expressed differently, if we assume that

$$Actual\ Event\ Time = T + d$$

where d is a random variable in the range $[0..X + 1]$, then the probability distribution of d would be uniform if a conventional timer interrupt based facility was used. With typical values for the measurement resolution and interrupt clock resolution of 1 MHz (1μsecs) and 1 KHz (1msec), respectively, X is 1000 and the maximal delay is 1001 μsecs.

With soft timers, the probability distribution of d is dependent on the system's workload, which influences how often trigger states are reached. Results shown in Section 5 show that among a variety of workloads, the worst case distribution of d results in a mean delay of 31.6 μsecs and is heavily skewed towards low values (median is 18 μsecs). Therefore, applications that can benefit from fine-grained events on the order of tens of μsecs in the common case, but can tolerate rare delays up to the resolution of the system's interrupt clock (typically 1msec), are well served by soft timers.

4 Applications of soft timers

In this section, we describe two applications of soft timers, rate-based clocking and network polling. In Section 5, we will present empirical results to evaluate the use of soft timers in these applications.

4.1 Rate-based clocking

As discussed in Section 2.1, achieving high utilization in networks with large bandwidth-delay products may require transport protocols like TCP to perform rate-based clocking. In a conventional implementation of rate-based clocking, a periodic hardware timer event must be scheduled at the intended rate of packet transmissions. At network speeds of several hundred Mbps and a packet size of 1500 Bytes (Ethernet), this would require timer interrupt rates of one every few tens of μsecs. Given the overhead of hardware timer interrupts (e.g., 4.45μsecs), this would lead to unacceptable overhead.

We observe that transmitting multiple packets per timer event would lead to bursty packet transmissions and defeat the purpose of rate-based clocking, which is to transmit data at a relatively constant rate. However, packet transmissions on different network connections that have separate bottleneck links could be performed in a single timer event.

Soft timers allow the clocked transmission of network packets at average intervals of tens of μsecs with low overhead. Due to the probabilistic nature of soft timer event scheduling, the resulting transmission rate is variable. In Section 5, we will empirically show the statistics of the resulting transmission process.

An interesting question is how a protocol implementation should schedule soft timer transmission events to achieve a given target transmission rate. Scheduling a series of transmission events at fixed intervals results in the correct average transmission rate. However, this approach can lead to occasional bursty transmissions when several transmission events are all due at the end of a long interval during which the system did not enter a trigger state. A better approach is to schedule only one transmission event at a time and let the protocol maintain a running average of the actual transmission rate. The next transmission event is then adaptively scheduled in the context of the previous event handler to smooth the rate fluctuations.

Our prototype implementation employs a simple algorithm for scheduling the next transmission. The algorithm uses two parameters, the target transmission rate and the maximal allowable burst transmission rate. The algorithm keeps track of the average transmission rate since the beginning of the current train of transmitted packets. Normally, the next transmission event is scheduled at an interval appropriate for achieving the target transmission rate. However, when the actual transmission rate falls behind the target transmission rate due to soft timer delays, then the next transmission is scheduled at an interval corresponding to the maximal allowable burst transmission rate.

We will experimentally evaluate the use of soft timers for rate-based clocking in Section 5.

4.2 Network polling

In conventional network subsystem implementations, the network interfaces generate a hardware interrupt to signal the completion of a packet reception or transmission[3]. Upon a receiver interrupt, the system accepts the packet, performs protocol processing and signals any blocked process that has been waiting to receive data. Upon a transmit interrupt, the system decreases the reference count on the transmitted packets' buffers, possibly deallocating them. In busy systems with high-speed network interfaces (e.g., server systems), network interrupts can occur at a rate of one every few tens of μsecs.

Another approach to scheduling network processing is polling, where the system periodically reads the network interfaces' status registers to test for completed packet receptions or transmissions. In a pure polling system, the scheduler periodically calls upon the network driver to poll the network interfaces.

Pure polling avoids the overhead of interrupts and it can reduce the impact of memory access locality shifts by (1) testing for network events at "convenient" points in the execution of the system, and by (2) aggregating packet processing. By performing polling when the scheduler is active, packet processing is performed at a time when the system already suffers a locality shift. By polling at an appropriate average rate, multiple packets may have completed since the

[3]Some interfaces can be programmed to signal the completion of a burst of packet transmissions or receptions.

last poll, thus allowing the aggregation of packet processing, increasing memory access locality.

However, the disadvantage of pure polling is that it may substantially increase communication latency by delaying packet processing. As a result, hybrid schemes have been proposed. Traw and Smith [23] use periodic hardware timer interrupts to initiate polling for packet completions when using a Gigabit network interface. This approach involves a tradeoff between interrupt overhead and communication delay. Mogul and Ramakrishan [17] propose a system that uses interrupts under normal network load and polling under overload, in order to avoid receiver livelock. When processing of a packet completes, the system polls the network interface for more outstanding packets; only when no further packets are found are network interrupts re-enabled.

Soft timers offer a third design choice. With soft timer based network polling, a soft timer event is used to poll the network interfaces. As in pure polling, network interrupts are avoided and memory access locality is improved because network polling and processing is performed only when the associated soft timer event expires and the system reaches a trigger state. However, since soft timer events can be efficiently scheduled at μsec granularity, communications latency can be close to that achieved with interrupt driven network processing in the common case.

In general, the soft timer poll interval can be dynamically chosen so as to attempt to find a certain number of packets per poll, on average. We call this number the *aggregation quota*. An aggregation quota of one implies that one packet is found, on average, per poll.

We will experimentally evaluate the use of soft timers for network polling in Section 5.

5 Experimental results

In this section, we present experimental results to evaluate the proposed soft timer facility. We quantify the overhead of our proposed soft timer facility and compare it to the alternative approach of scheduling events using hardware timer interrupts. We also present measurements that show the distribution of delays in soft timer event handling, given a variety of system workloads.

Finally, we evaluate the performance of soft timers when used to perform rate-based clocking and network polling.

5.1 Base overhead of hardware timers

Our first experiment is designed to determine the base overheads of a conventional hardware interrupt timer as a function of interrupt frequency.

The experimental setup consists of four 300MHz Pentium-II machines, each configured with 128MB of RAM and connected through a switched 100Mbps Ethernet. We ran the Apache-1.3.3 [3] Web server on one of the PII machines while the other three PII machines ran a client program that repeatedly requested a 6 Kbyte file from the Web server. The number of simultaneous requests to the Web server were set such that the server machine was saturated.

The FreeBSD-2.2.6 OS runs on the server machine. The kernel uses its standard timer facilities to schedule all events in the system. However, an additional hardware timer interrupt was scheduled with varying frequency. A “null handler” (i.e., a handler function that immediately returns upon invocation) was invoked whenever this timer interrupt fires, to isolate the overhead of the timer facility alone.

We then measured the throughput of the Apache server in the presence of the additional hardware timer, as a function of frequency. By measuring the impact of hardware interrupts on the performance of a realistic workload, we are able to capture the full overhead of hardware timers, including secondary effect like cache and TLB pollution that result from handling the timer interrupt.

Figure 2 plots the throughput of the Apache Web server as the interrupt frequency of the hardware timer is increased to 100KHz. Figure 3 plots the percentage reduction in throughput and is indicative of the overhead imposed by the hardware interrupts. The results show that the interrupt overhead increases approximately linearly with frequency and can be as high as 45% at an interrupt frequency of 100KHz (one interrupt every 10 μsecs). From these results, it can be calculated that the average combined overhead per interrupt is about 4.45μsecs[4].

We repeated the experiment on a machine with a 500MHz Pentium III (Xeon) CPU running FreeBSD-3.3 and found that the interrupt overhead was 4.36μsecs. This indicates that interrupt overhead does not scale with CPU speed and suggests that the relative cost of interrupts increases as CPUs get faster. Finally, a similar experiment performed on an AlphaStation 500au (500MHz 21164 CPU) running FreeBSD-4.0-beta resulted in an interrupt overhead of 8.64μsecs. This indicates that the high overhead associated with interrupt handling is not unique to Intel PCs.

Note that the overhead of a timer interrupt can be lower on both platforms when the machine is idle, since the code, data and TLB entries used during interrupt handling remain fully cached. Our experiment tries to obtain a more meaningful measure of the overhead by evaluating the total impact of timer interrupts on the performance of a real workload that stresses the memory system. The results show that timer interrupts have a significant overhead.

5.2 Base overhead of soft timers

The next experiment determines the base overhead of soft timers. We implemented soft timers in the FreeBSD kernel. Trigger states were added in the obvious places described in Section 3. In practice, we found that the trigger interval distribution could be improved by adding a few additional trigger states to ensure that certain kernel loops contain a trigger state. Examples of such loops are the TCP/IP output loop and the TCP timer processing loop. Since Intel x86 CPUs handle TLB misses in hardware, these events could not be used as trigger states in our prototype.

The idle loop checks for pending soft timer events. However, to minimize power consumption, an idle CPU halts

[4] Measurements using performance counters to measure the average elapsed time spent in the interrupt handler confirm this result.

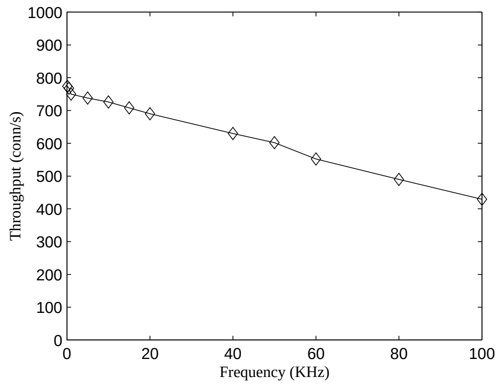

Figure 2. Apache throughput

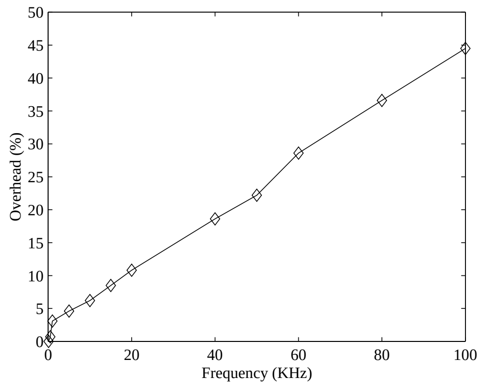

Figure 3. Base interrupt overhead

when either (a) there are no soft timer events scheduled at times prior to the next hardware timer interrupt, or (b) another idle CPU is already checking for soft timer events.

In our next experiment, we scheduled a periodic soft timer event such that a handler was invoked whenever the system reaches a trigger state. That is, we programmed the soft timer facility to invoke a soft timer event handler at the maximal frequency possible, given the Web server workload. As with the hardware timer, a "null handler" was invoked whenever the soft timer fired.

The soft timer handler invocations caused no observable difference in the Web server's throughput. This implies that the base overhead imposed by our soft timer approach is negligible. This is intuitive because the calls to the handler execute with the overhead of a procedure call, whereas a hardware interrupt involves saving and restoring the CPU state. With soft timers, the event handler was called every 31.5 μsecs on average. We observe that using a hardware interrupt timer at a frequency of one event every 30 μsecs (33.3 KHz) would have a base overhead of approximately 15%.

5.3 Soft timer event granularity under different workloads

Recall that once a soft timer event is due, the associated handler is executed at the earliest time when the system reaches a trigger state. The performance of a soft timer facility, i.e., the granularity and precision with which it can schedule events, therefore depends on the frequency at which the system reaches trigger states.

We measured the distribution of times between successive trigger states for a variety of workloads. Figure 4 shows the cumulative distribution function of time between successive trigger states.

The workloads are as follows. "ST-Apache" corresponds to the Apache Web server workload from the previous experiment. In "ST-Apache-compute", an additional compute-bound background process is running concurrently with the Web server. "ST-Flash" is a Web server workload using a fast event-driven Web server called *Flash* [20]. "ST-real-audio" was measured with a copy of the RealPlayer [22] running on the machine, playing back a live audio source from

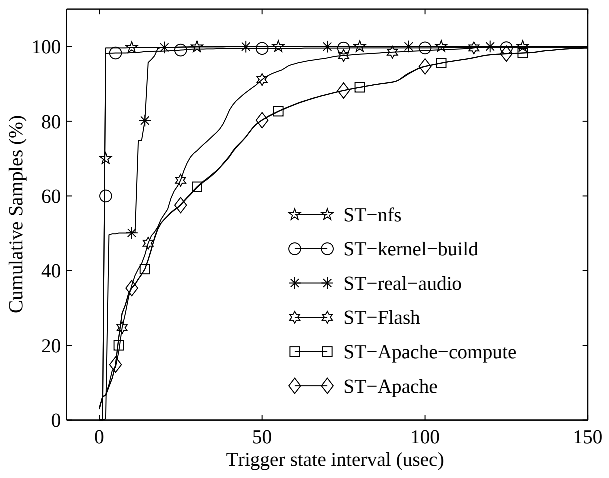

Figure 4. Trigger state interval, CDF

the Internet. "ST-nfs" reflects the trigger state inter-arrival times when the workload is a NFS fileserver. Finally, "ST-kernel-build" was measured while a copy of the FreeBSD-2.2.6 kernel was built on the machine from the sources.

Additional information about the distribution with each workload is given in Table 1. Two million samples were taken in each workload to measure the distributions.

The results show that under a workload typical of a busy Web server, the soft timer facility can schedule events at a mean granularity of tens of μsecs with negligible overhead and with delays over 100 μsecs in less than 6% of the samples. As shown below, this performance is sufficient to perform rate-based clocking of 1500 byte packets at several hundreds of Mbits/sec and it allows effective polling of network interface events at the same rate.

In a busy Web server, it is intuitive that the many network packet arrivals, disk device interrupts and system calls provide frequent trigger states. One concern is that the presence of compute-bound background computations may cause long periods where the system does not encounter a trigger state, thus degrading the performance of the soft timer facility.

To measure this effect, we added a compute-bound back-

	Max (μsec)	Mean (μsec)	Median (μsec)	StdDev (μsec)	$> 100\mu$sec (%)	$> 150\mu$sec (%)
ST-Apache	476	31.52	18	32	5.3	0.39
ST-Apache-compute	585	31.59	18	32.1	5.3	0.43
ST-Flash	1000	22.53	17	20.8	1.09	0.013
ST-real-audio	1000	8.47	6	13.2	0.025	0.013
ST-nfs	910	2.13	2	3.3	0.021	0.011
ST-kernel-build	1000	5.63	2	47.9	0.038	0.033
ST-Apache (Xeon)	1000	19.41	11	23	0.44	0.13

Table 1. Trigger state interval distribution

ground process to the Web server, which executes in a tight loop without performing system calls ("ST-Apache-compute"). The results show that the presence of background processes has no tangible impact on the performance of the soft timer facility. The reason is that a busy Web server experiences frequent network interrupts that have higher priority than application processing and yield frequent trigger states even during periods where the background process is executing.

"ST-nfs" is another example of a server workload. The NFS server is saturated but disk-bound, leaving the CPU idle approximately 90% of the time. The vast majority of samples indicate a trigger state interval around 2μsecs on this workload.

The RealPlayer ("ST-real-audio") was included because it is an example of an application that saturates the CPU. Despite the fact that this workload performs mostly user-mode processing and generates a relatively low rate of interrupts, it yields a distribution of trigger state intervals with very low mean, due to the many systems calls that RealPlayer performs.

Finally, we measure a workload where the FreeBSD OS kernel is built from the source code. This workload involves extensive computation (compilation, etc.) as well as disk I/O.

To determine the impact of CPU speed on the trigger interval distribution, we repeated the experiment with the "ST-Apache" workload on a machine with a 500MHz Pentium III (Xeon) CPU running FreeBSD-3.3. The summary information about the resulting distribution is included in Table 1. The results show that the shape of the distribution is similar to that obtained with the slower CPU, however the mean is reduced by a factor that roughly reflects the CPU clock speed ratio of the CPUs. This indicates that the granularity of soft timer events increases approximately linearly with CPU speed.

While our selection of measured workloads is necessarily limited, we believe that the soft timer facility can provide fine-grained event support across a wide range of practical workloads. The reason is that (1) most practical programs frequently make system calls, suffer page faults, TLB faults or generate other exceptions that cause the system to reach a trigger state and (2) the soft timer facility can schedule events at very fine grain whenever a CPU is idle.

In the most pessimistic scenario, all CPUs are busy, the executing programs make infrequent system calls, cause few page faults or other exceptions and there are few device I/O interrupts. These conditions mark the absence of significant I/O or communication activity in the workload, and can arise, for instance, in scientific applications. However, observe that μsec timers are used primarily in networking, and it is thus unlikely that any soft timer events are scheduled under such conditions.

5.4 Changes in trigger interval distribution over time

The trigger interval distributions shown in the previous section are aggregated over 2 million samples, corresponding to 4–64 secs of execution time for the various workloads. A related question is how the trigger interval distribution changes during the runtime of a workload. For instance, it is conceivable that context switching between different processes could cause significant changes in the trigger interval distribution. To investigate this question, we computed the medians of the trigger interval distributions during intervals of 1 ms and 10 ms. Results are plotted in Figure 5 for a period of 10 secs of the runtime of the "ST-Apache-compute" workload. The x-axis represents the runtime of the workload, the y-axis shows the median of the trigger interval distribution during a given interval (1 ms and 10ms).

With 1ms intervals, the bulk of the trigger interval medians are in the range from 14 to 26μsecs. A few intervals (less than 1.13%) have medians above 40μsecs. The medians for the 10ms intervals (which corresponds to a timeslice in the FreeBSD system), on the other hand, almost all fall into a narrow band between 17 and 19μsecs.

These results indicate that the dynamic behavior of the workload appears to cause noticeable variability in the trigger interval distribution over 1ms intervals. However, there is little variability in the trigger interval distributions over 10ms intervals.

5.5 Trigger interval distribution by event source

A related question is what fraction of trigger states is contributed by each event source and how that contribution affects the resulting trigger state interval distribution. To answer this questions, we separately accounted for trigger states by event source for the "ST-Apache" workload. Table 2 shows the fraction of trigger state samples contributed by each event source.

The sources "syscalls" and "traps" are self-explanatory. The source "ip-output" generates a trigger event every time

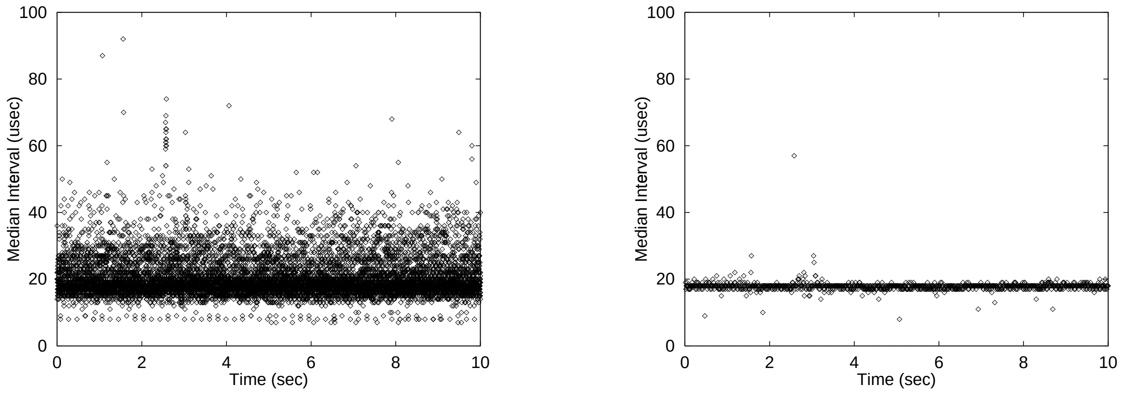

Figure 5. Trigger interval medians during 1 ms and 10 ms intervals, ST-Apache-compute workload

an IP packet (e.g., TCP segment) is transmitted. The source "tcpip-others" represents a number of other trigger states in the network subsystem, such as the processing loop for TCP timers. Network interface interrupts are reflected in the "ip-intr" source.

Source	Fraction of samples (%)
syscalls	47.7
ip-output	28
ip-intr	16.4
tcpip-others	5.4
traps	2.5

Table 2. Trigger state sources

Figure 6 shows the impact that each trigger source has on the trigger interval distribution. The graphs show the CDFs of the resulting trigger interval distributions when one of the trigger sources is removed. For instance, "no ipintr" shows the CDF of the resulting trigger interval distribution when there is no trigger state associated with network interrupts. "All" represents the original distribution for the "ST-Apache" workload from Figure 4. It is evident from the results that system calls and IP packet transmissions are the most important sources of trigger events in this workload.

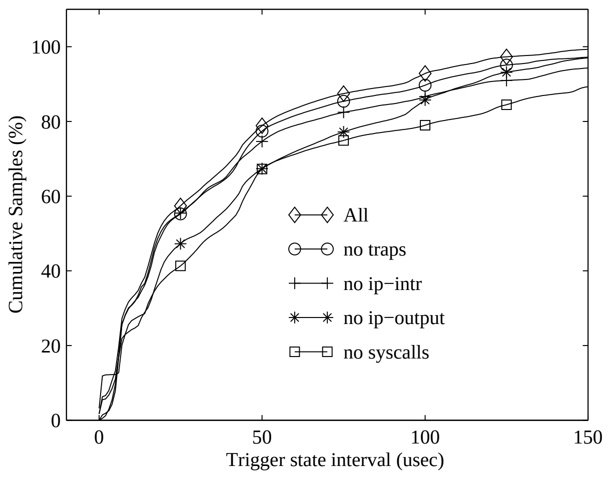

Figure 6. Impact of event sources on trigger interval, CDF (ST-Apache workload)

5.6 Rate-based clocking: timer overhead

In this section, we evaluate the use of soft timers to perform rate-based clocking in TCP. We show results that compare the overhead of performing rate-based clocking with soft timers versus hardware timer interrupts, we evaluate the statistics of the packet transmission process and we explore the potential for network performance improvements due to rate-based pacing.

Our first experiment is designed to explore the overhead of rate-based clocking in TCP using soft timers versus hardware timers. The experimental setup is the same as in the previous experiment except that the Web server's TCP implementation uses rate-based clocking using either soft timers or a conventional interrupt timer to transmit packets.

The soft timer was programmed to generate an event every time the system reaches a trigger state. One packet is transmitted whenever the handler is invoked and a packet is pending transmission. On a LAN, such as the one used in our testbed, FreeBSD's TCP implementation does not use slow-start. Thus, all packets are normally sent in a burst, as fast as the outgoing network link can transmit them. Since the transmission of a 1500 byte packet takes 120 μsecs on our 100Mbps network, the use of rate-based clocking has no observable impact on the network. Therefore, the experiment isolates the overhead of using soft timers versus hardware timers for rate-based clocking in TCP, but does not expose possible benefits of rate-based clocking.

Table 3 shows the performance results obtained in this experiment. We present results for both the Apache-1.3.3 Web server as well as the Flash server. For the results with hardware interrupt timers, the 8253 was programmed to interrupt once every 20 μsecs (50KHz frequency), causing the dispatch of a thread (BSD software interrupt) that transmits a packet. From the previous experiments, we know the base overhead for event dispatch at this rate is about 22%. The extra overhead indicated by the results is most likely due to cache pollution, since the computation performed by the handler is exactly the same as that performed during trans-

	Apache	Flash
Base Throughput (conn/s)	774	1303
HW timer throughput (conn/s)	560	827
HW timer Ovhd (%)	28	36
HW timer Avg xmit intvl (μsecs)	31	35
Soft timer throughput (conn/s)	756	1224
Soft timer Ovhd (%)	2	6
Soft timer Avg xmit intvl (μsecs)	34	24

Table 3. Overhead of rate-based clocking

mission of a packet in the original TCP implementation.

The results indicate that the effect of cache pollution with hardware timers is at least 4% ($28 - 22 - 2$) and 8% ($36 - 22 - 6$) worse than with soft timers for the Apache and the Flash server, respectively. The fact that Flash appears to be more affected by the cache pollution can be explained as follows. Apache is a multi-process server whose frequent context switching leads to relatively poor memory access locality. Flash, on the other hand, is a small, single-process event-driven server with presumably relatively good cache locality. It is intuitive, therefore, that the Flash server's performance is more significantly affected by the cache pollution resulting from the timer interrupts.

The results also show that the average time between transmissions with soft timers is only slightly higher than with the hardware timer when using the Apache server, and it is lower when using the Flash server. This result can be explained as follows. With hardware timers, the transmission rate is lower than the rate at which the 8253 chip was programmed because the transmission event handler may in general be delayed due to disabled interrupts. On the other hand, soft timers perform substantially better when the Flash server is used because that server is much faster than Apache and therefore generates trigger states at a higher rate. The combined effect is that soft timers with Flash result in a lower time between transmissions than the hardware timer.

In summary, the results of this experiment show that soft timers can be used to do rate-based clocking in TCP at rates that approach Gigabit speed with very low overhead (2-6% in our experiment). Using a conventional interrupt timer at this rate has an overhead of 28-36% in our experiment and is therefore not practical.

5.7 Rate-based clocking: transmission process statistics

As discussed in Section 4, our implementation of rate-based clocking based on soft timers uses an adaptive algorithm for scheduling transmissions, in order to smooth variations in the transmission rate caused by the probabilistic nature of soft timers. The algorithm keeps track of the actual sending rate, and whenever this rate falls behind the target sending rate, the next transmission event is scheduled so as to achieve the maximal allowable burst sending rate, until the actual sending rate once again catches up with the target sending rate.

We performed an experiment to determine the actual achievable transmission rate and the resulting statistics of the transmission process, as a function of the maximal allowable burst transmission rate, assuming a target transmission rate of one packet every 40μsecs and 60μsecs, respectively. The workload in this experiment was that of the busy Web server ("ST-Apache" in Figure 4), which is among the two workloads with the largest mean trigger state interval (i.e, worst case).

We assume in this experiment that the bandwidth of the network link attached to the sender is 1Gbps and the packet size is 1500 bytes. Therefore, the minimal interval setting of 12 μsecs reflects the maximal transmission rate of the network link. At this minimal interval setting, rate-based clocking is allowed to send packets at the link bandwidth whenever the actual rate is below the target transmission rate.

The results are shown in Tables 4 and 5 for target transmission intervals of 40μsecs and 60μsecs, respectively. For comparison, results for hardware timer based rate-based clocking were also included. The hardware timer was programmed to fire regularly at the target transmission interval.

The results show that soft timers can support rate-based clocking up to rates of one packet transmission every 40μsecs, if it is allowed to send bursts at the link speed of one packet every 12μsecs. As the minimal allowable burst interval is increased, the soft timers can no longer maintain an average transmission interval of 40μsecs, and drops to 65.9μsecs at a minimal allowable interval of 35μsecs.

At a target interval of 60μsecs, soft timers can maintain the average interval up to a minimal allowable burst interval of 30μsecs. The standard deviation is in all cases in the 30–35μsecs range and improves as the minimal burst interval increases, as expected.

We note that these measurements apply to rate-based clocking on a single connection. Soft timers can be used to clock transmission on different connections simultaneously, even at different rates. (A server may perform many transmission simultaneously, resulting in large aggregate bandwidths.) In this case, multiple packets may be transmitted on different connections in a single soft timer event (i.e., in the context of one trigger state).

With hardware timers, rate-based clocking falls short of the target transmission rate by 3μsecs and 3.6μsecs, respectively. The reason is that some timer interrupts are lost during periods when interrupts are disabled in FreeBSD. The hardware timers achieve a somewhat better standard deviation than soft timers, which is to be expected given the probabilistic nature of the latter.

We also note that the base overhead of using timer interrupt at the target transmission rates of 40 and 60μsecs is at least 13% and 8.5%, respectively (see Figure 3). Finally, we observe that only a single hardware timer device is available in most system. It is impossible, therefore, to use a hardware timer to simultaneously clock multiple transmissions at different rates, unless one rate is a multiple of the other. Moreover, reprogramming the timer device frequently to a different rate may be too expensive, due to the long latency associated with accessing device registers. In practice, this may cause additional deviation from the target transmission rate.

Combined with the high overhead, these concerns raise

	Soft timers		Hardware timers	
Min interval (μsec)	Avg interval (μsec)	Std Dev	Avg interval (μsec)	Std Dev
12 (line speed)	40	34.5	43.6	26.8
15	48	31.6	-	-
20	51.9	30.9	-	-
25	57.5	30.9	-	-
30	61	30.5	-	-
35	65.9	30.1	-	-

Table 4. Rate-based clocking (target transmission interval = 40μsecs)

	Soft timers		Hardware timers	
Min interval (μsec)	Avg interval (μsec)	Std Dev	Avg interval (μsec)	Std Dev
12 (line speed)	60	35.9	63	27.7
15	60	33.2	-	-
20	60	32.3	-	-
25	60	31.2	-	-
30	61	30.5	-	-
35	65.9	30	-	-

Table 5. Rate-based clocking (target transmission interval = 60μsecs)

questions about the feasibility of rate-based clocking with hardware timers at high network speeds. Soft timers, on the other hand, can support multiple transmissions at different rates and with low overhead.

5.8 Rate-based clocking: network performance

Our next experiment attempts to quantify the potential impact of rate-based clocking on the achieved performance of a Web server over network connections with high bandwidth-delay products.

In our prototype implementation of rate-based clocking in TCP, we assume that the available capacity in the network is known. In practice, estimating the available capacity is not a trivial problem. Practical mechanisms for bandwidth estimation and other details of the integration of rate-based clocking into TCP require further research and are beyond the scope of this paper. Related work in this area is discussed in Section 6.

To show the potential effect of rate-based clocking on TCP throughput, we performed an experiment where a variable amount of data is transmitted over a network connection with high bandwidth-delay product. We model this connection in the laboratory by transmitting the data on a 100Mbps Ethernet via an intermediate Pentium II machine that acts as a “WAN emulator”. This machine runs a modified FreeBSD kernel configured as an IP router, except that it delays each forwarded packet so as to emulate a WAN with a given delay and bottleneck bandwidth. In our experiment, we choose the WAN delay as 50ms and the bottleneck bandwidth to be either 50Mbps or 100Mbps. As a result, the TCP connection between client and server machine has a bandwidth-delay product of either 5Mbits or 10Mbits. Network connections with these characteristics are already available in vBNS and will soon be available in the general Internet.

We performed HTTP requests across the laboratory “WAN” connection to an otherwise unloaded server. Either the standard FreeBSD TCP implementation was used, or alternatively our modified implementation, which avoids slow-start and instead uses soft-timer based rate-based clocking at a rate corresponding to the bottleneck bandwidth, i.e., one packet every 120μsecs (100Mbps) or 60μsecs (50Mbps), respectively. Since a persistent connection is assumed to be already established prior to starting the experiment, there is no delay due to connection establishment. The results are shown in Tables 6 and 7.

We see that rate-based clocking can lead to dramatic improvements in throughput, response time and network utilization on networks with high bandwidth-delay products. Response time reductions due to rate-based clocking range from 2% for large transfers to 89% for medium sized transfers (100 packets or 141 KBytes). These improvements are the result of rate-based clocking’s ability to avoid TCP slow-start, which tends to underutilize networks with large bandwidth-delay products on all but very large transfers.

Since the average HTTP transfer size is reported to be in the 5–13 KB range [4, 16], rate-based clocking can have a significant impact on the Web.

5.9 Network polling

Our final experiment evaluates the use of soft timers for network polling. We implemented network polling in the FreeBSD-2.2.6 kernel, using soft timers to initiate the polling. The polling interval is adaptively set to attempt to find a given number of received packet per poll interval, on average (aggregation quota).

In this experiment, a 333MHz Pentium II machine with 4 Fast Ethernet interfaces was used as the server. Four 300MHz PII machines were used as the client machines, each connected to a different interface on the server.

	regular TCP		rate-based clocking		
Transfer size (1448 Byte packets)	Xput (Mbps)	Response time (msecs)	Xput (Mbps)	Response time (msecs)	Resp. time reduction (%)
5	0.12	496	0.57	101.2	79
100	1.01	1145	9.36	123.7	89
1000	6.75	1714	34.07	340	80
10000	29.95	3867	46.33	2500	35
100000	45.54	25432	46.60	24863	2

Table 6. Rate-based clocking network performance (Bandwidth = 50Mbps, RTT = 100 msecs)

	regular TCP		rate-based clocking		
Transfer size (1448 Byte packets)	Xput (Mbps)	Response time (msecs)	Xput (Mbps)	Response time (msecs)	Resp. time reduction (%)
5	0.16	350	0.58	100.6	71
100	1.09	1056	10.34	112	89
1000	6.38	1815	51.94	223	87
10000	38.46	3012	86.77	1335	55
100000	81.37	14235	91.92	12601	11

Table 7. Rate-based clocking network performance (Bandwidth = 100Mbps, RTT = 100 msecs)

We measured the throughput of two different Web servers (Apache and Flash), given a synthetic workload, where clients repeatedly request the same 6KB file. The throughput was measured on an unmodified FreeBSD kernel (conventional interrupt based network processing) and with soft timer based network polling. Table 8 shows the results for the two different servers, for aggregation quotas ranging from 1 to 15, and for conventional (HTTP) and persistent connection HTTP (P-HTTP).

The throughput improvements with soft timer based polling range from 3% to 25%. The benefits of polling are more pronounced with the faster Flash server, as it stresses the network subsystem significantly more than the Apache server and, owing to its better locality, is more sensitive to cache pollution from interrupts. With P-HTTP, amortizing the cost of establishing a TCP connection over multiple requests allows much higher throughput with both servers, independent of polling.

The difference between the results for the conventional interrupt-based system and network polling with an aggregation quota of 1 (i.e., one packet per poll on average) reflects the benefit of avoiding interrupts and the associated improvement in locality. The network polling results with aggregation quotas greater than one reflect the additional benefits of aggregating packet processing.

In general, aggregation of packet processing raises concerns about increased packet delay and ACK compression. However, we believe that aggregation is practical with soft-timer based network polling, for two reasons. Firstly, soft-timer based network polling is turned off (and interrupts are enabled instead) whenever a CPU enters the idle loop. This ensures that packet processing is never delayed unnecessarily. Secondly, when rate-based clocking is used, packet transmissions are not paced by incoming ACKs. With rate-base clocking, it is therefore no longer necessary to preserve the exact timing of incoming ACKs, i.e., ACK compression is of lesser concern.

Finally, we observe that future improvements in CPU and network speeds will continue to increase the rate of network interrupts in conventional network subsystem implementations. Since the relative cost of interrupt handling is likely to increase as CPUs get faster (see Section 5.1), avoiding interrupts becomes increasingly important.

5.10 Discussion

Soft timers allow the efficient scheduling of events at a granularity below that which can be provided by a conventional interval timer with acceptable overhead. The “useful range” of soft timer event granularities is bounded on one end by the highest granularity that can be provided by a hardware interrupt timer with acceptable overhead, and on the other end by the soft timer trigger interval. On our measured workloads on a 300 MHz PII CPU, this useful range is from a few tens of μsecs to a few hundreds of μsecs. Moreover, the useful range of soft timer event granularities appears to widen as CPUs get faster. Our measurements on two generations of Pentium CPUs (300MHz PII and 500MHz PIII) indicate that the soft timer event granularity increases approximately linearly with CPU speed, but that the interrupt overhead (which limits hardware timer granularity) is almost constant.

Soft timers can be easily integrated with an existing, conventional interval timer facility. The interval timer facility provides conventional timer event services, and its periodic interrupt is also used to schedule overdue soft timer events. Conventional timers should be used for events that need to be scheduled at or below the granularity of the interval timer’s periodic interrupt. Soft timers should be used for events that require a granularity up to the trigger state interval, provided these events can tolerate probabilistic delays up to the granularity of the conventional interval timer.

	Interrupt Xput (req/sec)	Soft Poll Xput (req/sec)				
Aggregation	1	1	2	5	10	15
HTTP						
Apache	854 (1.0)	915 (1.07)	933 (1.09)	939 (1.10)	944 (1.11)	945 (1.11)
Flash	1376 (1.0)	1568 (1.14)	1620 (1.17)	1690 (1.23)	1702 (1.24)	1719 (1.25)
P-HTTP						
Apache	1346 (1.0)	1380 (1.03)	1395 (1.04)	1421 (1.06)	1439 (1.07)	1440 (1.07)
Flash	4439 (1.0)	4816 (1.08)	5071 (1.14)	5271 (1.19)	5376 (1.21)	5498 (1.24)

Table 8. Network polling: throughput on 6KB HTTP requests

6 Related work

The implementation of soft timers is based on the idea of polling, which goes back to the earliest days of computing. In polling, a main-line program periodically checks for asynchronous events, and invokes handler code for the event if needed.

The novel idea in soft timers is to implement an efficient timer facility by making the operating system "poll" for pending soft timer events in certain strategic states. These "trigger states" are known to be reached very frequently during execution. Furthermore, these states are associated with a shift in memory access locality, thus allowing the interposition of handler code with little impact on system performance. The resulting facility can then be used to schedule events at a granularity that could not be efficiently achieved with a conventional hardware timer facility.

Traw and Smith [23] use periodic hardware timer interrupts to initiate polling for packets completions when using a Gigabit network interface. This approach involves a tradeoff between interrupt overhead and communication delay. With soft timer based network polling, on the other hand, one can obtain both low delay and low overhead.

Mogul and Ramakrishan [17] describe a system that uses interrupts under normal network load and polling under overload, in order to avoid receiver livelock. Their scheme disables interrupts during the network packet processing and polls for additional packets whenever the processing of a packet completes; when no further packets are found, interrupts are reenabled.

In comparison, soft timer based network polling disables interrupts and uses polling whenever the system is saturated (i.e., no CPU is idle). That is, polling is used even when the packet interarrival time is still larger than the time it takes to process packets. Moreover, soft timers allow the dynamic adjustment of the poll interval to achieve a predetermined packet aggregation quota.

A number of researchers have pointed out the benefits of rate-based clocking of TCP transmissions [25, 18, 1, 10, 5]. Our work shows that using conventional hardware timers to support rate-based clocking at high bandwidth is too costly, and we propose soft timers as an efficient alternative.

The use of rate-based clocking has been proposed in the context of TCP slow-start, when an idle persistent HTTP (P-HTTP) connection becomes active [19, 16, 12]. Visweswaraiah et. al. [25] observe that an idle P-HTTP connection causes TCP to close its congestion window and the ensuing slow-start phase tends to defeat P-HTTP's attempt to utilize the network more effectively that HTTP/1.0 [7] connections. A similar observation was made by Padmanabhan et. al. in [18]. Soft timers can be used to efficiently clock the transmission of packets upon restart of an idle P-HTTP connection.

Allman et. al. [1] show the limiting effect of slow-start and congestion avoidance schemes in TCP in utilizing the bandwidth over satellite networks. Using rate-based clocking instead of slow-start addresses the former concern. Feng et. al. [10] propose the use of rate-based clocking in TCP to support the controlled-load network service [26], which guarantees a minimal level of throughput to a given connection.

Balakrishnan et. al. [5] have proposed ACK *filtering*, a mechanism that attempts to improve TCP performance on asymmetric network paths by discarding redundant ACKs at gateways. They observe that this method can lead to burstiness due to the big ACKs seen by the sender and suggest pacing packet transmissions so as to match the connection's sending rate.

Besides an efficient timer mechanism, rate-based clocking also depends on mechanisms that allow the measurement or estimation of the available network capacity. A number of techniques have been proposed in the literature. The basic packet-pair technique was proposed by Keshav [14]. Hoe et. al. [13] propose methods to improve TCP's congestion control algorithms. They set the slow-start threshold (ssthresh) to an appropriate value by measuring the bandwidth-delay product using a variant of the packet-pair technique. Paxson [21] suggests a more robust capacity estimation technique called PBM that forms estimates using a range of packet bunch sizes. A technique of this type could be used to support rate-based clocking. Allman and Paxson [2] compare several estimators and find that sender-side estimation of bandwidth can often give inaccurate results due to the failure of the ACK stream to preserve the spacing imposed on data segments by the network path. They propose a receiver-side method for estimating bandwidth that works considerably better.

7 Conclusions

This paper proposes a novel operating system timer facility that allows the system to efficiently schedule events at a granularity down to tens of microseconds. Such fine-grained events are necessary to support rate-based clocking of transmitted packets on high-speed networks and can be used to support efficient network polling.

Unlike conventional timer facilities, soft timers take advantage of certain states in the execution of a system where an event handler can be invoked at low cost. In these states, the saving and restoring of CPU state normally required upon a hardware timer interrupt is not necessary, and the cache/TLB pollution caused by the event handler is likely to have low impact on the system performance.

Experiments with a prototype implementation show that soft timers can be used to perform rate-based clocking in TCP at granularities down to a few tens of microseconds. At these rates, soft timers impose an overhead of only 2–6% while a conventional timer facility would have an overhead of 26–38%. The use of rate-based clocking in a Web server can improve client response time over connections with high bandwidth-delay products by up to 89%.

Soft timers can also be used to perform network polling, thus avoiding network interrupts while preserving low communications delays. Experiments show that the performance of a Web server using this optimization can increase by up to 25% over a conventional interrupt based implementation.

Furthermore, the performance improvements obtained with soft timers can be expected to increase with network and CPU speeds. As networks and CPUs get faster, so does the rate of network interrupts. However, the speed of interrupt handling does not increase as fast as CPU speed, due to its poor memory access locality. The relative cost of interrupt handling therefore increases, underscoring the need for techniques that avoid interrupts.

Soft timer performance, on the other hand, appears to scale with CPU speed. Soft timers are cache friendly and faster CPU speeds imply that trigger states are reached more frequently, thus improving the granularity at which soft timers can schedule events.

Acknowledgments

We are grateful to the anonymous reviewers and our shepherd, Mendel Rosenblum, for their helpful comments. Thanks to Sitaram Iyer and Erich Nahum for their help with some of the experiments. This work was supported in part by NSF Grant CCR-9803673, by Texas TATP Grant 003604, by an IBM Partnership Award, and by equipment donations from Compaq Western Research Lab and from HP Labs.

References

[1] M. Allman, C. Hayes, H. Kruse, and S. Ostermann. TCP Performance over Satellite Links. In *Proceedings of 5th International Conference on Telecommunication Systems*, pages 456–469, Nashville, TN, Mar. 1997.

[2] M. Allman and V. Paxson. On estimating end-to-end network path properties. In *Proceedings of the SIGCOMM '99 Conference*, pages 263–274, Cambridge, MA, Sept. 1999.

[3] Apache. http://www.apache.org/.

[4] M. F. Arlitt and C. L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the ACM SIGMETRICS '96 Conference*, pages 126–137, Philadelphia, PA, Apr. 1996.

[5] H. Balakrishnan, V. N. Padmanabhan, and R. H. Katz. The Effects of Asymmetry on TCP Performance. In *Proceedings of 3rd ACM Conference on Mobile Computing and Networking*, pages 77–89, Budapest, Hungary, Sept. 1997.

[6] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, and R. H. Katz. TCP behavior of a busy internet server: Analysis and improvements. In *Proceedings of IEEE INFOCOM '98*, pages 252–262, San Francisco, CA, Apr. 1998.

[7] T. Berners-Lee, R. Fielding, and H. Frystyk. RFC 1945: Hypertext transfer protocol – HTTP/1.0, May 1996. ftp://ftp.merit.edu/documents/rfc/rfc1945.txt.

[8] L. Brakmo and L. Peterson. Performance Problems in 4.4BSD TCP. *ACM Computer Communication Review*, 25(5):69–86, Oct. 1995.

[9] L. Brakmo and L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, Oct. 1995.

[10] W. c. Feng, D. D. Kandlur, D. Saha, and K. G. Shin. Understanding and improving TCP performance over networks with minimum rate guarantees. *IEEE/ACM Transactions on Networking*, 7(2):173–187, Apr. 1999.

[11] K. Fall and S. Floyd. Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. *Computer Communication Review*, 26(3):5–21, July 1996.

[12] R. Fielding, J. Gettys, J. Mogul, H. Nielsen, and T. Berners-Lee. RFC 2068: Hypertext transfer protocol – HTTP/1.1, Jan. 1997. ftp://ftp.merit.edu/documents/rfc/rfc2068.txt.

[13] J. C. Hoe. Improving the Start-up Behaviour of a Congestion Control Scheme for TCP. In *Proceedings of the ACM SIGCOMM '96 Symposium*, pages 270–280, Stanford, CA, Sept. 1996.

[14] S. Keshav. A Control-Theoretic Approach to Flow Control. In *Proceedings of the ACM SIGCOMM '91 Symposium*, pages 3–15, Zürich, Switzerland, Sept. 1991.

[15] J. C. Mogul. Observing TCP Dynamics in Real Networks. In *Proceedings of the ACM SIGCOMM '92 Symposium*, pages 281–292, Baltimore, MD, Aug. 1993.

[16] J. C. Mogul. The Case for Persistent-Connection HTTP. In *Proceedings of the ACM SIGCOMM '95 Symposium*, pages 299–313, Cambridge, MA, Sept. 1995.

[17] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, Aug. 1997.

[18] V. N. Padmanabhan and R. H. Katz. TCP Fast Start: A Technique For Speeding Up Web Transfers. In *Proceedings of the IEEE GLOBECOM '98 Conference*, pages 41–46, Sydney, Australia, Nov. 1998.

[19] V. N. Padmanabhan and J. C. Mogul. Improving HTTP Latency. In *Proceedings of the Second International WWW Conference*, pages 995–1005, Chicago, IL, Oct. 1994.

[20] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceeding of the Usenix 1999 Annual Technical Conference*, pages 199–212, Monterey, CA, June 1999.

[21] V. Paxson. End-to-End Internet Packet Dynamics. In *Proceedings of the ACM SIGCOMM '97 Symposium*, pages 139–152, Cannes, France, Sept. 1997.

[22] RealPlayer. http://www.realplayer.com/.

[23] J. M. Smith and C. B. S. Traw. Giving applications access to Gb/s networking. *IEEE Network*, 7(4):44–52, July 1993.

[24] G. Varghese and A. Lauck. Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 171–180, Austin, TX, Nov. 1987.

[25] V. Visweswaraiah and J. Heidemann. Improving restart of idle TCP connections. Technical Report 97-661, University of Southern California, November 1997.

[26] J. Wroclawski. RFC 2211: Specification of controlled-load network element service, Sept. 1997. ftp://ftp.merit.edu/documents/rfc/rfc2211.txt.

[27] L. Zhang, S. Shenker, and D. D. Clark. Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic. In *Proceedings of the ACM SIGCOMM '91 Symposium*, pages 133–148, Zürich, Switzerland, 1991.

A The need for rate-based clocking

In this appendix, we provide further motivation for rate-based clocking. We restrict ourselves here to a general discussion of how an appropriate timer facility can be used for rate-based clocking of transmissions. The details of how a specific protocols like TCP should be extended to add rate-based clocking may require further research; they are beyond the scope of this paper.

A.1 ACK compression and big ACKs

Previous work has demonstrated the phenomenon of *ACK compression*, where ACK packets from the receiver lose their temporal spacing due to queuing on the reverse path from receiver to sender [27, 15]. ACK compression can cause bursty packet transmissions by the TCP sender, which contributes to network congestion. Balakrishnan et. al. [6] have observed the presence of ACK compression in a busy Web server.

With rate-based clocking, a TCP sender can keep track of the average arrival rate of ACKs. When a burst of ACKs arrives at a rate that significantly exceeds the average rate, the sender may choose to pace the transmission of the corresponding new data packets at the measured average ACK arrival rate, instead of the burst's instantaneous rate as would be dictated by self-clocking.

A related phenomenon is that of *big ACKs*, i.e., ACK packets that acknowledge a large number of packets or update the flow-control window by a large number of packets. Upon receiving a big ACK, self-clocked senders may send a burst of packets at the bandwidth of the network link adjacent to the sender host. Transmitting such bursts can adversely affect congestion in the network. A detailed discussion of phenomena that can lead to big ACKs (i.e., ACKs that can lead to the transmission of more than 3 packets) in TCP is given in Section A.3.

Using rate-based clocking, it is possible to avoid sending packet bursts in the same way as was described above in connection with ACK compression.

A.2 Slow-start

Self-clocked protocols like TCP use a *slow-start* phase to start transmitting data at the beginning of a connection or after an idle period. During slow-start, the sender transmits a small number of packets (typically two), and then transmits two more packets for every acknowledged packet, until either packet losses occur or the estimated network capacity is reached. In this way, the sender increases the amount of data transmitted per RTT exponentially until the network capacity is reached.

The disadvantage of slow-start is that despite the exponential growth of the transmit window, it can take many RTTs before the sender is able to fully utilize the network. The larger the bandwidth-delay product of the network, the more time and transmitted data it takes to reach the point of network saturation. In particular, transmissions of relatively small data objects may not allow the sender to reach the point of network saturation at all, leading to poor network utilization and low effective throughput.

The bulk of traffic in the Internet today consists of HTTP transfers that are typically short (between 5KB and 13KB) [16, 4]. A typical HTTP transfer finishes well before TCP finishes its slow-start phase, causing low utilization of available network bandwidth and long user-perceived response times [16]. The magnitude of this problem is expected to increase as higher network bandwidth becomes available.

Slow-start serves a dual purpose. It starts a transmission pipeline that allows the sender to self-clock its transmission without sending large bursts of packets. At the same time, it probes the available network capacity without overwhelming the network. The key idea to avoid slow-start is the following. If the available network capacity is known or can be measured/estimated, then a TCP sender can immediately use rate-based clocking to transmit packets at the network capacity without going through slow-start [18].

The problem of measuring available network capacity has been addressed by several prior research efforts, for instance packet pair algorithms [14, 9, 13] and PBM [21]. Moreover, when starting transmission after an idle period, the network capacity during the last busy period can be used as an estimate for the current capacity [19, 16, 12]. Finally, in future network with QoS support, the available network capacity may be known *a priori*.

A.3 Causes of big ACKs

In the previous section, we discussed the effects of big ACKs on TCP connections. Here, we describe several phenomena that can cause big ACKs.

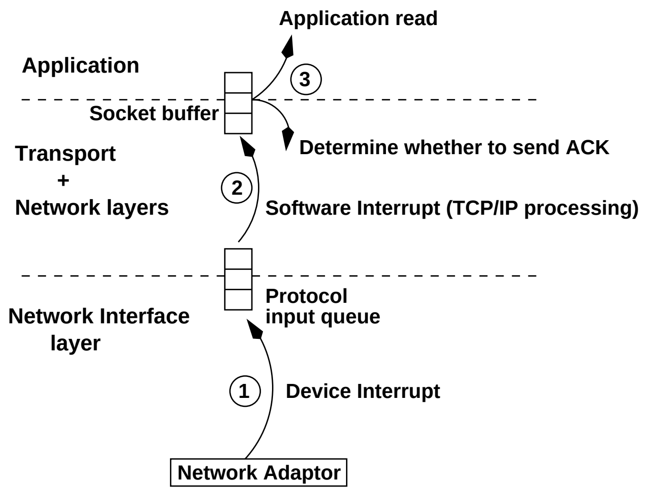

Figure 7. Packet processing path in OS

Figure 7 shows the processing of a packet, starting from its reception by the network adaptor to its delivery to the application. 1) A high priority device interrupt places the packet into the input queues of the IP protocol, 2) TCP/IP processing is done in the context of a software interrupt and the packet is placed in the application's socket buffer, 3) The application reads the data from its socket; in the context of this read, an ACK is sent back to the TCP sender if needed.

Upon reception of a packet acknowledging x packets, a TCP sender normally injects x new closely spaced packets into the network. In normal operation, x is 2 because TCP receivers usually delay every other ACK[5] to take advantage of piggybacking opportunities. We now present some scenarios that cause a TCP receiver to send big ACKs (ACKs that acknowledge more than 3 packets), causing the sender to inject a burst of packets that can adversely affect congestion in the network.

[5]The presence of TCP options causes TCP receivers to send an ACK for every 3 packets [8].

Figure 7 indicates that an ACK is sent by the receiver when the application reads the data from the socket buffer (or when the delayed ACK timer fires). If the interarrival time of packets is smaller than the packet processing time, then owing to the higher priorities of the interrupts as compared to application processing, all closely spaced packets sent by the TCP sender will be received before any ACK is sent. When the incoming packet train stops (due to flow control), the receiver will send a big ACK to the sender acknowledging all packets sent. The same happens if the delayed ACK timer fires first. The problem is self-sustaining because the TCP sender responds to the big ACK by sending a burst of closely spaced packets.

On a 300MHz Pentium II machine, the packet processing time can take more than 100 μsecs while the minimum interarrival time of 1500 byte packets on 100Mbps and 1Gbps Ethernet is 120 μsecs and 12 μsecs, respectively. This suggests that big ACKs can be prevalent in high-bandwidth networks.

The situation described above is not necessarily restricted to high-bandwidth networks. It can also happen when the receiver application is slow in reading newly arrived data from the socket buffers. This can happen, for example, when a Web browser (TCP receiver) is rendering previously read graphics data on the screen. During this time, ACKs for all packets from the Web server (TCP sender) shall be delayed until either the delayed ACK timer fires (once every 200ms) or the browser reads more data from the socket buffer. The ACK packet when sent would acknowledge a large number of packets.

While high bandwidth is not yet widely available in WANs, we have analyzed TCP packet traces on a 100Mbps LAN and have observed big ACKs on almost every sufficiently long transfer. We have also analyzed packet traces from the Rice CS departmental Web server. Our results show that 40% of all transfers that were greater than 20Kbytes showed the presence of big ACKs, thus confirming our hypothesis that big ACKs also occur on transfers over current low-bandwidth WAN links.

Brakmo and Peterson [8] have also observed these big ACKs in the context of recovery from large number of packet losses and reordering of packets. They propose to reduce TCP congestion window upon receiving a big ACK so that slow-start is used instead of sending packet bursts. Fall and Floyd [11] propose to use a maxburst parameter to limit the potential burstiness of the sender for packets sent after a loss recovery phase (fast recovery). While these techniques can limit the burstiness, they adversely affect bandwidth utilization as the network pipeline is drained of packets.