PC Based Precision Timing Without GPS

Attila Pásztor
EMULab at the Department of Electrical & Electronic Engineering
The University of Melbourne, Victoria 3010, Australia
and Ericsson Hungary R&D
a.pasztor@ee.mu.oz.au

Darryl Veitch
EMULab at the Department of Electrical & Electronic Engineering
The University of Melbourne, Victoria 3010, Australia
d.veitch@ee.mu.oz.au

ABSTRACT

A highly accurate monitoring solution for active network measurement is provided without the need for GPS, based on an alternative software clock for PC's running `Unix`. With respect to clock *rate*, it's performance exceeds common GPS and NTP synchronized software clock accuracy. It is based on the TSC register counting CPU cycles and offers a resolution of around 1ns, a rate stability of 0.1PPM equal to that of the underlying hardware, and a processing overhead well under 1μs per timestamp. It is scalable and can be run in parallel with the usual clock. It is argued that accurate rate, and not synchronised offset, is the key requirement of a clock for network measurement. The clock requires an accurate estimation of the CPU cycle period. Two calibration methods which do not require a reference clock at the calibration point are given. To the TSC clock we add timestamping optimisations to create two high accuracy monitors, one based on Linux and the other on Real-Time Linux. The TSC-RT-Linux monitor has offset fluctuations of the order of 1μs. The clock is ideally suited for high precision active measurement.

Keywords

Timing, Synchronization, Network Measurement, PC clocks, software clock, NTP, GPS

1. INTRODUCTION

Because of their power, flexibility, and low cost, personal computers are increasingly being called upon to perform a variety of tasks for which they were never designed. One of these is as engines for the measurement of network traffic. Both passive and active measurement require network monitoring, which involves timestamping packets as they arrive on the network interface. In addition, active measurement requires the sending of a stream of test packets at carefully determined target times, calculated to achieve specific network probing objectives. To fufill each of these two roles in software on a standard PC, the availability of an accurate, reliable, and high resolution *software clock* is primordial.

Two of the most fundamental attributes of any clock are its *rate* and *offset*, that is its departure from the true time[1] at a given time. It is commonly accepted, or assumed, that in general neither of these are particularly reliable for software clocks in PC's. Of the two, the offset has traditionally generated the most concern, and so mechanisms such as the Network Time Protocol (NTP) are widely used to enforce synchronization [1, 8]. However, for network measurement, we point out that it is not offset but *the rate characteristics which are of central importance*. In this paper we focus on providing a software clock with highly smooth and accurate rate performance, based purely on standard hardware. Our approach is very generic, so that the clock could also be used in other contexts, especially when smooth rate performance is a prime consideration. We illustrate and optimize the use of the clock however in the context of network measurement, especially active probing, to which it is ideally suited.

There are three reasons for the focus on rate. First, many measurements, including the important round-trip *delay* and *inter-arrival times* of packets, are made at a single physical location using the same clock, so that offset synchronization is not an issue. Second, any measured delay series can be decomposed into *delay variation* (the difference of delay) and a constant. Of these two components, only the latter is subject to offset synchronization error, however because delay values are large, high accuracy is then not of crucial importance. For example for one-way delays in the range of 100's of milliseconds, an offset error of 1ms is acceptable. On the other hand the delay variation, although immune to offset error, is vulnerable to rate variations in the clocks at either end, and as its value may be under a millisecond, it requires far higher timestamping accuracy. Finally, many metrics, and methods, are already based implicitly or explicitly on delay variation or inter-arrival times, which are each independent of a constant offset difference. All of the more advanced techniques of active probing for example are based on fine properties of the delay variation and/or inter-arrival time series ([7, 5, 11]). Only in relatively few (though important) cases is the actual delay **value** required, but then, as we observed, very high accuracy is not crucial.

Our motivation for this work was to circumvent the disadvantages of the existing timestamping techniques used in inexpensive PC based network measurement, particularly active probing. Of these, the simplest is to use the standard software clock with network based NTP synchronization. Under optimal conditions, such as 100Mb/s LAN access to a synchronized primary NTP server, NTP can bound the offset to the order of a millisecond from Coordinated

[1] Newtonian space-time will be assumed in this paper.

Universal Time (UTC). Although this may seem sufficient for many network measurement purposes, it should be put in the context of controlled packet sending, for example for 100Mb/s Ethernet the transmission times of minimum (28 byte) to maximum (1500 byte) packets range from around 2 to 130μs. Furthermore, the accuracy of NTP can be far worse than 1ms, and is in any case unreliable, as it is subject to variations in network conditions, including outages [12]. Because of this, and particularly when measuring one-way *delay*, where different clocks are used for the departure and arrival timestamps of the measured packets, Global Positioning System (GPS) based timing is commonly used as a means of improving synchronization, to around 10μs in common GPS-`Unix` based solutions. However, this solution is a logistically difficult addition to an otherwise simple infrastructure, as it typically requires roof access for reliable continuous satellite coverage. Installation costs to machine rooms can run to many thousands of dollars! The exploitation of radio based alternatives relies on the presence of the appropriate network and also implies additional hardware. In this paper we describe a simple alternative software clock which is more reliable, at least as accurate for rate based needs, and has higher resolution than the solutions discussed above, but which requires no special hardware and can be easily installed in parallel to the usual clock.

The essence of this clock is very simple. The CPU clock cycle (TSC) register is used to keep track of time at very high resolution, for example 1 nanosecond for a 1 gigahertz processor. The updating of this register is a hardware operation, and reading it and storing its value involves only very fast memory accesses. Provided that we have an accurate estimate of the true duration of a clock cycle, the stored value can be readily converted to an accurate relative time. We take advantage of the fact that CPU clocks have high *oscillator stability* to validly assume that the corresponding conversion constant is, to high precision, constant over quite long time periods, well beyond those needed for most high resolution measurement needs. We then propose two methods for estimating this constant to the required accuracy. The first involves using NTP – not to directly modify the clock using the NTP algorithm(s), but to indirectly estimate the clock skew over large time intervals. The second involves using a single reference timing source to remotely calibrate other machines across a network.

Using the TSC register simply to increase resolution is not new. What is novel is the exploitation of the fundamental reliability of the available hardware to build a clock which has a range of important benefits, and avoids a number of pitfalls of the existing software clock. Its rate accuracy is inherently more accurate than that of the usual GPS + software clock combination, but in fact in typical `Unix` systems the accuracy of both clocks is limited by the operating system and hardware noise. In particular the delays incurred in accessing the clock, the timestamping operation itself, suffers from system 'noise' in the form of process scheduling, and the effects of interrupt latency. For the purposes of traffic measurement we reduce the noise by performing an ultra low cost timestamping instruction in the driver code to the network interface card. By using Real-Time Linux we can in addition control scheduling problems, resulting in orders of magnitude improvement over the traditional solution. Again, we emphasize that the above provides for accurate *rates* in individual or geographically separated clocks, and therefore for approximately constant, but non-zero, comparative offset. That is, synchronized rates, but not values. We also however offer a discussion on improved solutions to the quite separate problem of offset synchronization.

Validation of timing methods would not be possible without a reliable timing reference. We used a 'DAG3.2e' series measurement card designed for high accuracy and high performance passive monitoring of 10/100 Mbit/s Ethernet, synchronized to a GPS receiver, yielding a time stamping accuracy around 100ns [9]. The card was connected to a Magellan OEM/5V GPS receiver, the same one used by the primary NTP server the host was connected to, located in the same building and only 1.5ms away.

2. UNIX TIMING

To discuss issues related to clock accuracy in detail more precise terminology is needed. Clock *resolution* is the smallest unit by which a clock's time is updated. The *offset* $\theta(t)$ is the difference between the time reported by the clock and the true reference time t at a particular moment t. The *skew* γ is roughly the difference between the clock's rate and a reference rate. To define it more precisely consider the following general model for the offset

$$\theta(t) = \gamma * t + \omega(t), \tag{1}$$

where the skew is just the coefficient of the deterministic linear part, $\omega(t)$ being a random remainder which encapsulates the deviations from the 'simple skew' model $\theta(t) = \gamma * t$. The *oscillator stability* [3] characterizes $\omega(t)$ via the family, indexed by time-scale τ, of relative offset errors:

$$y_\tau(t) = \frac{\theta(t+\tau) - \theta(t)}{\tau} = \gamma + \frac{\omega(t+\tau) - \omega(t)}{\tau}. \tag{2}$$

In other words $y_\tau(t)$ is the total rate error when measured over time scale τ. The series $y_\tau(t)$ (which has mean γ if we assume zero mean for $\omega(t)$), can be thought of as the skew γ with some random variation of zero mean. A particular variance estimator of the variance of $y_\tau(t)$, known as the *Allan variance*, plotted over a range of τ values, is a traditional characterization of oscillator stability [3]. A study over a range of time-scales is essential as the source and nature of timing errors vary according to the measurement interval. For example it is important to note that the rate quantity $y_\tau(t)$ only really corresponds to the intuitive skew concept over timescales large enough to make γ apparent. At small timescales, γ will be invisible, and the 'rate' error will essentially correspond to the high frequency noise.

Significant Time Interval	Interval Duration	Error rate, PPM 0.1	50	500
Interrupt latency	2μs	$\ll$ 1ns	0.1ns	1ns
Periodic interrupt period	10ms	1ns	0.5μs	5μs
Standard unit	1s	0.1μs	50μs	.5ms
Max. useful est^n interval	5.55h	**2ms**	1s	10s

Table 1: Absolute errors at key error rates and intervals.

To discuss offset, and its 'derivative', rate error, we use the dimensionless unit of *Parts Per Million* (PPM). Table 1 translates this into absolute error (offset) over key time intervals. The significance of the error rates in the table, discussed in detail below, are i) clock stability: 0.1PPM, ii) typical skew from nominal rate (or boot time measurement): 50PPM, and iii) software clock maximum adjustment: 500PPM.

2.1 Unix Clocks

All PCs have at least two clocks, the independent battery powered *Real-Time* or *Hardware Clock*, and the *System Clock*, often referred to as the *Software Clock*. Some processors, for example in

Pentium based PCs, also provide access to their clock cycle register (TSC), which can be utilized in different ways as a high resolution clock. In this paper we assume that the TSC is available, and furthermore that the base clock rate is constant (many notebook computers allow variable CPU rates). Our work is Linux based, although some tests were also performed under BSD, with very similar results. The description that follows will of course vary under different systems, but it is a common configuration that also serves to introduce the key concepts.

The real-time clock is independent and keeps track of time even when the system is turned off, and is typically not used while the system is running.

The software clock, which is accessed using the `gettimeofday()` system call, is based on a counter of timer interrupts. These are typically generated by dividing, via a 8254 timer chip, the 1.19318 MHz output signal of a standard quartz oscillator (residing on the motherboard), and are used by the operating system to schedule execution of both system and user tasks. The division is typically to `HZ`=100Hz, yielding interrupts every $\delta = 10$ms, which sets a coarse bound on the resolution of the clock. Furthermore, the oscillator has a skew, in the range of $\pm$50PPM (see below) which is not measured nor accounted for. This oscillator will be referred to as the 'standard oscillator'. Incrementing the counter is one of the highest priority periodic tasks.

A common enhancement to the base software clock or 'interrupt clock' is to exploit the TSC register to interpolate time between the interrupts. The resulting much greater resolution is capped at 1μs, the smallest unit available in the standard data structure for `Unix` timestamps, used in the interface definition of `gettimeofday()`. To use this technique the true period of a CPU cycle must be known. The procedure typically followed is to measure the number per μs once only at boot time and thereafter to use this constant to transform the 64 bit clock cycle register contents to time in μs units. This initial measurement however is prone to error as it is based on using the standard oscillator over a very small measurement interval of around 50ms. The 'interpolation–clock' therefore inherits the skew of the standard oscillator. Furthermore, since the period of the standard oscillator is close to 1μs, resolution effects result in an additional error of the order of 1μs per 50ms, or 20PPM (if the interrupt period is shortened by increasing the `HZ` kernel parameter in `Linux`, integer arithmetic effects can increase this to 100's of PPM). The overall skew of the interpolation–clock is thus that of the standard oscillator plus an additional systematic component of up to $\pm$20PPM.

We see that the standard software clock combines two different oscillators, with related but different skews which are either poorly estimated or unknown. At small timescales below the interrupt period the interpolation–clock skew will be in force, while at large time scales the other will dominate. Accessing the software clock via `gettimeofday()`, which activates a reading of the software clock counter, the TSC, and the interpolation calculation, takes on the order of 1μs to execute on our 600Mhz test machines. NTP tunes adjustments to the interrupt clock, but does not affect the interpolation clock.

2.2 Experiment Design

We describe the measurement protocols used in the paper. A periodic probe stream of 40 byte UDP packets of period T, serving as a set of opportunities to collect and compare the accuracy of timestamps on a controlled time-scale, is sent to a test machine. A high accuracy RT-Linux based sender was used, based on that reported recently in [10]. The packets reach the interface card in the test machine, and the DAG card via passive tap off the wire, at approximately the same time. The DAG card timestamps are taken at the beginning of the packets, and the network card generates a hardware interrupt after the full packet has arrived. From this point there are several different possibilities, in two main categories, depending on the aim of the experiment. In the first category we study the characteristics of the existing clocks. To do so a user program, which is listening on the relevant socket and has the highest possible priority, reacts to the interupt by executing timestamping operations in user space. If two clocks are compared, the timestamping calls are executed immediately after each other. In the other category, we wish to compare the performance of different traffic monitors, which can be thought of as a combination of a clock and a timestamping operation. As detailed in section 4, these are the TSC-RT-Linux, the TSC-Linux, and the SW-NTP-tcpdump monitors. In the case of the TSC-RT-Linux monitor, the interrupt immediately results in the driver being invoked. The driver has been modified to immediately record the TSC register value, and then to store it for postprocessing. With the TSC-Linux monitor, after possibly some scheduling delay, again a modified driver records the TSC register. Finally, under the SW-NTP-tcpdump monitor, the Berkeley Packet Filter code in the kernel timestamps the packets using `gettimeofday()` as they are passed up from the driver. In all cases, the differences between the different software clocks and the DAG timestamps form the estimations of offset for that clock or monitor, on a close to periodic time grid.

The normalised difference in the offsets, the $y_\tau(t)$ above, can then be calculated over timescales of $\tau = \{T, 2T, 3T, \cdots\}$. The sequence of un-normalised offset differences at times $\tau = kT$ are also of interest as they can be naturally interpreted as errors in Inter-Arrival Time (IAT) measurements between every kth packet. Determination and control of these errors is important as they represent a key limitation to the use of IATs in active measurement. Accordingly, depending on context, results are presented for

offset : $\theta(t)$, used when examining how the actual error in timestamps increases over time (in all comparisons we set $\theta(0) = 0$),

offset error : $\theta(t+\tau) - \theta(t)$, when wishing to examine the offset at a particular (and by extension, over a range) of timescales. This is directly relevant to the issue of errors in IAT measurement as just mentioned,

rate error : $[\theta(t+\tau) - \theta(t)]/\tau$, or *normalised offset error*, when we want to examine the value of the skew γ, and the validity of the simple skew model, over different timescales.

The test machines were used in a lightly loaded configuration to minimize errors due to scheduling. From previous work, we know that in this mode only a few major events per 10,000 can be expected, which are readily recognised as they are isolated and in the millisecond range. Scheduling effects are pointed out at various points in the text. Several different 600Mhz Linux machines were used, each with Linux version 2.2.14, and when appropriate RT-Linux version 2.2. The RT-Linux networking software had to be extended, and some bugs fixed.

2.3 Clock Skew and Stability

In this section we measure the skew and stability of the two components of the software clock to illustrate the points described above, benchmark our measurement infrastructure, and prepare the ground for the new clock. In order to study the 'natural' behaviour of the underlying oscillators, NTP is not used in the experiments in this section. The results are important as the high stability we find provides the basis for our approach.

One can effectively focus on the interrupt–clock component by choosing timescales well over the interrupt period, here $\tau = 60000\delta =$ 600s was used. This meant that the effect of interpolation-skew over the measurement was at worst of the order of 2×50PPM of 10ms over 600 seconds, or 0.0017PPM. The linear increase in error in the lower curve in figure 1, a skew of 38PPM, is therefore essentially that of the interrupt–clock. To focus on the skew of the interpolation–clock it was necessary to modify the software clock in the kernel. A new `gettimeofday()` function was created based purely on a counting and conversion of the TSC value, using the usual error ridden period estimate as measured at boot time. The upper curve in the figure reveals that this estimate entailed a skew error of 55PPM. (Note that this modification of the software clock was used for this purpose only. It is not the same as the TSC based clock and monitors we present in this paper.) For ease of comparison we have set $\theta(0) = 0$.

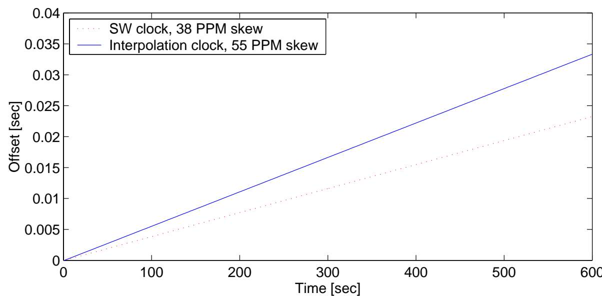

Figure 1: Offsets of the two component clocks used in the standard software clock (NTP off), showing significant skew.

As we move from small to large timescales, the overall skew of the normal software clock moves monotonically from 55PPM to that of the interrupt–clock, 38PPM, as the 20PPM discretisation error is averaged away. These skew levels are consistent with the typical figure of 50PPM reported in [3]. This is quite a large error, nonetheless we see from table 1 that in the range of a single interrupt it is still below 1μs, below the software clock resolution. It is clearly vastly greater however, than what would be possible using the TSC register with an accurate period estimate, a fact which we exploit in the next section.

We now consider the stability of the two component clocks. The aim is not to characterise the detailed statistical behaviour of their stabilities, but to identify the timescales at which a pure or 'simple-skew' model is meaningful. Recall from equation 2 that stability is essentially concerned with the variance of error in rate. In figure 2 the top plot shows the rate varying over a 24 hour period for both the software clock (again, this is essentially the interrupt–clock over large timescales) and the interpolation–clock. From the raw data taken every $T = 1$s a value of $\tau = 1000s$ was used in the plot. The range of variation is seen to be very small for both, of the order of 0.1PPM over the period, that is skew varying in a ± 0.1PPM range. The lower plots show the corresponding Allan deviation curves (the root of the Allan variance) in logarithmic co-

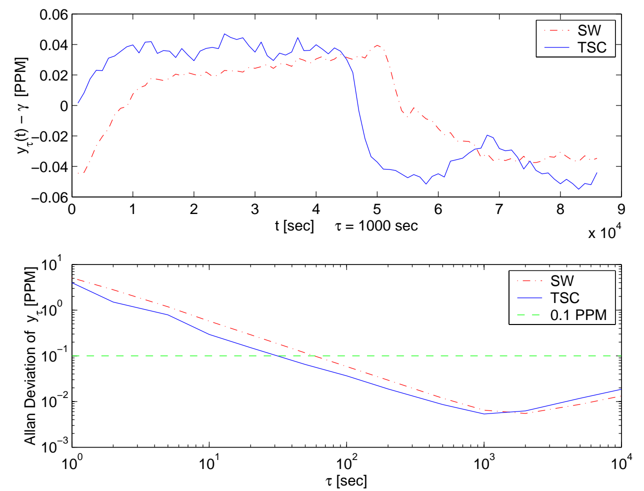

Figure 2: Stability of the software clock (SW, estimating the interrupt–clock) and the interpolation–clock (TSC based) over 24 hours. Top: rate variations with $\tau = 1000s$. Bottom: Allan deviation plots showing the validity of a simple skew plus noise model up to 1000s.

ordinates. The plots have a similar form and show a strong resemblance to the measurement results published in [3]. We found similar results for other machines running Linux and BSD over three very different networks.

In each Allan plot two regimes are clearly visible. Over measurement time scales τ up to some 1000 seconds, the slope of -1 indicates that the Allan deviation falls rapidly as $\frac{1}{\tau}$. This is consistent with $\omega(t)$ being an additive stationary short-range dependent 'noise'. The interrupt latency noise, which is hardware dependent and takes values up to 5μs but which is more typically around 1μs, is one effect which falls into this category The second region is determined by the long term oscillator stability (the deviation seems to move up but in fact for large τ there are few data points left and the confidence intervals are large. It is therefore not inconsistent with the horizontal behaviour found in [3]). The important fact is that the change point marks the end of the simple skew plus noise regime. For the clocks in the test machine it occurs at around 1000s and at these scales the error is around 0.01 PPM. Values taken from many computers [3] indicate that 0.1PPM is a typical figure, and one which we use in the sequel. Attempts will only be made to correct rate to this level of accuracy, as beyond it the simple skew model ceases to be valid in any case.

Note that improved estimates of the stability of TSC based clocks can easily be obtained by using a RT-Linux receiver, however we do not do so here in order to provide a fair comparison against the standard software clock, which does not exist under RT-Linux. The results in any case are similar.

2.4 Synchronization and NTP

The Network Time Protocol (NTP) is used in the Internet and elsewhere to synchronize computer clocks to an external reference source. The servers providing the synchronization service are hierarchically organized, with primary servers synchronized directly to external reference clocks such as GPS, and secondary servers synchronized to primary servers and others in the synchronization subnet. The synchronization is based on the exchange of timestamps between the client and the server, and possibly some other peer.

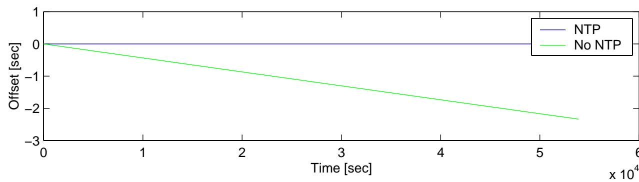

Figure 3: Offset comparison: the positive effect of NTP on large time scales.

The round-trip delay and the clock offset is then estimated from them. An upper bound in the offset error estimation is half of the round-trip time [1].

NTP is essentially concerned with controlling *offset* within reasonable bounds. This is normally done by a series of rate adjustments which are controlled by sophisticated phase locking algorithms in the kernel, and passed down to the software clock (in fact the 'interrupt clock') in a careful way [4]. It is important to understand however that its objective is not constant, smooth rate, which is of key importance for high precision measurement, but bounded offset. Because of the high skew of the standard oscillator, the range of the kernel parameter (`tickadj` under Linux), which implements the change in rate, needs to be large, and rate changes can be as high as 500PPM when variable network conditions result in a large error in the offset estimate. In fact, most of the corrections performed by the NTP clock adjustment algorithm do not compensate for changes in oscillator rate (stability), which as we have seen are very small, but for the much larger errors due to the skew of the standard oscillator, and errors in the NTP offset estimates due to changing network delay and system noise (the skew of the interpolation clock, and integer arithmetic used in the clock implementation, also play a role). Not only do these adjustments create irregularities in rate, they also imply that NTP synchronized hosts, even those synchronized to nearby primary servers, can only provide time stamping accuracy in the range of milliseconds. Indeed, a performance analysis of NTP [2] has shown that the accuracy of the primary servers is in the range of 10μs, and a client synchronizing to a primary server via Ethernet can achieve clock accuracy of around 1ms at best. In the results presented here the host was connected to a primary NTP server which was only 1.5ms away, *near optimal* conditions for the NTP + software clock combination.

In figure 3 we compare the software clock offset of our test host with and without running NTP. For this particular experiment probe packets were sent every second to the host which captured the arrival times using `tcpdump`. As always the DAG card provided the timing reference. The synchronizing effect of NTP on time scales well above a millisecond is evident from the figure. As we look more closely however, we will see that the rate is highly variable on short to medium timescales due to the effects noted above.

We first describe the two main NTP applications, *NTP–Date* and *NTP–daemon* (see [1] for a detailed description). The purpose of NTP–Date is to provide a one-off ability to bring an uncontrolled clock back into synchronization quickly. In contrast, NTP–daemon is designed to track offset in a reasonably gentle way, and is adaptive. The default parameters allow adaptivity with update intervals logarithmically spaced between 64 seconds and 17.1 minutes.

In figure 4 we illustrate two fundamentally different sources of irregularity in rate on small scales: (1) system 'noise', that is delays due to scheduling and software and hardware processing and queueing, and (2) network based NTP in the worst case (ie. using NTP–Date, however still under optimal server conditions). We show both offset (left) and its difference $\theta(t + \tau) - \theta(t)$ with $\tau = 1$ms (right). For this purpose we use a TSC based clock called by a user program (like the interpolation clock described in section 2.3 but with an accurate cycle period estimate), because it is unaffected by NTP, but shares the same system noise as the system clock. To avoid possible confusion we call it the TSC-User clock, and use it only in this section. The top plots in figure 4 concern the TSC-User clock, and the middle plots the software clock with NTP–Date running. Both show complex variations, and we cannot tell which are due to operating systems effects, and which to clock rate effects. In the bottom plots however the difference of the two is given. Since the TSC-User and software clock timestamps are requested immediately after each other, any operating system delays between the reference timestamping of the packets by the DAG and the clock timestamps will be experienced equally by both. The difference therefore shows the true difference in performance between the clocks (although it is not impossible that a scheduling event could occur inbetween or during the two timestamping calls, it is extremely unlikely as the total duration of the two: TSC-User+SW $< (0.1+1)\mu$s is small. We observed that this occurs no more often than once per 100000 timestamps). A clear skew difference is seen in the offset in the bottom left plot, which is revealed in the bottom right plot as being due to rate change recommendations of 5μs being handed down at **each** periodic interrupt. The size of the system noise can also be evaluated by comparing the plots and is seen to be of the order of 10μs.

Figure 5 compares the offset of the TSC-User clock with that of the software clock under the direction of NTP–daemon with default parameters. Although under NTP the offset has been bounded to 1ms as one might expect, the difference in rate stability is dramatic, even though the worst rate change in this interval corresponds to only 10PPM, far less than the maximum of 500PPM. These differences are visible here because we are at timescales for which the system noise has become negligible.

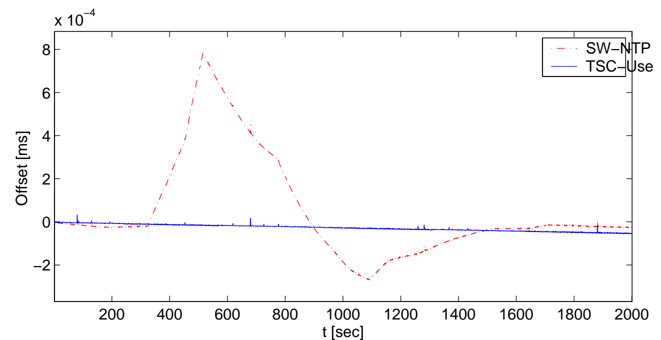

Figure 5: Offset comparison of the TSC-User clock and NTP–daemon + the software clock. The rate stabilities are evident. (spikes are due to shared system noise.)

Figure 5 demonstrates two things. First, that NTP induced rate changes can be large in response to system noise and/or variability in network delays. The resulting false offset estimates then trigger subsequent clock adjustments. Second, it graphically illustrates the inherently smooth nature of the CPU cycle rate, which can be exploited via the TSC register. Note that the spikes shared by both clocks are due to system noise on the time the clocks are *read*, and do not relate to the dynamics of the clocks themselves.

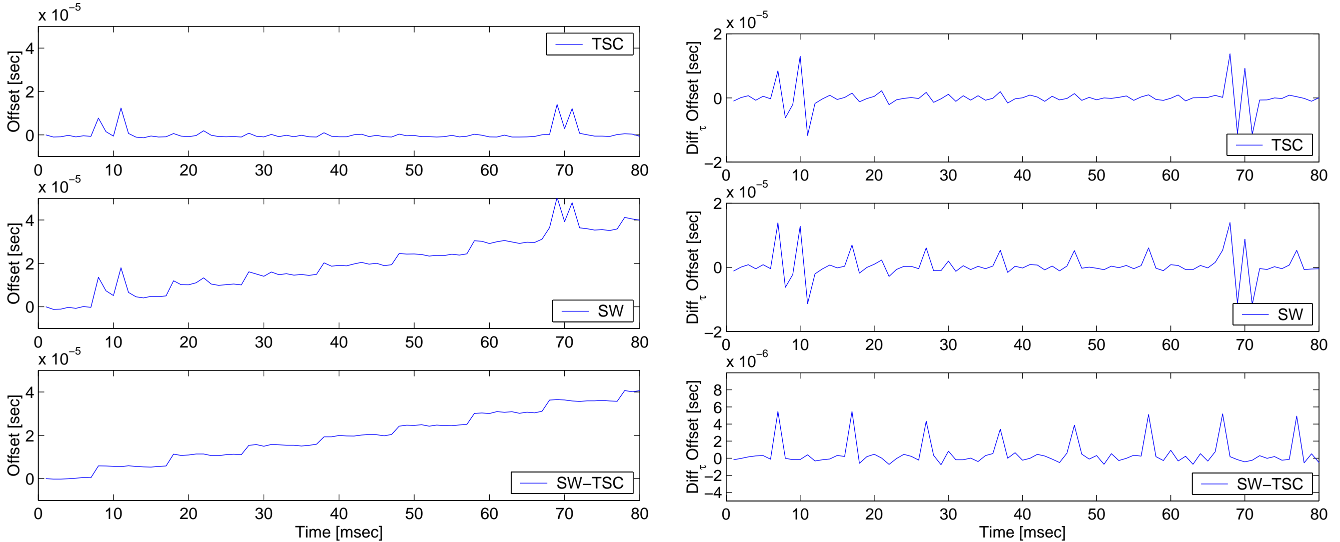

Figure 4: Two different sources of irregularity in rate: NTP and system noise. Top plots: TSC-User clock, Middle: software clock, Bottom: difference. The effect of NTP–Date is easily seen in both the offsets (left) and rates (right). Note scale changes.

Offset errors over small scales have been examined in figure 4. To explore and compare them over a wide range of medium timescales, in figure 6 we plot mean absolute value of the growth of offset error of the TSC-User clock (the same in both plots) and NTP–daemon under two parameter settings, the highest possible rate of polling (top), and the default parameters (bottom). For NTP–daemon, not surprisingly more frequent polling of the server results in a smaller offset (note the scale change), which is superior to that of the TSC-User clock for periods over 2000s. The crossover point moves to the right as the polling rate decreases and is around 6 hours in the lower plot (see below). Note however that even 2000s is well over the time-scales of importance in traffic measurement!

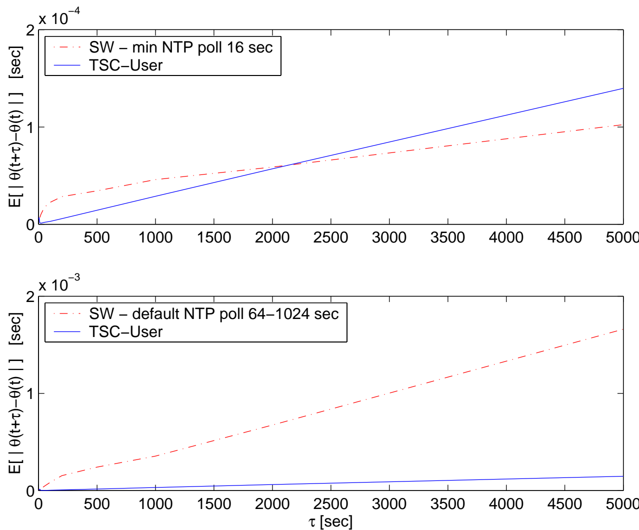

Figure 6: Comparison of mean absolute value of offset error ($\theta(0) = 0$)) over time. Top: TSC-User and NTP–daemon with polling set to the minimum 16s, Bottom: TSC-User and NTP–daemon with default parameters.

Increasing the polling rate to the NTP server also increases network traffic, and so the polling is often set low, another reason why the synchronization performance can be much poorer than the optimal value of 1ms.

2.5 GPS

In cases when the accuracy of network based clock synchronization is not sufficient, GPS receivers are often used instead as high precision external references, both to improve offset synchronization, and to avoid the network delay dependence of NTP. Large scale measurement projects collecting statistics of one-way metrics such as Surveyor [6] and RIPE [13] all use GPS synchronized measurement hosts.

The accuracy of modern GPS receivers' output signals is well below 1μs. Typically the output signal is fed to the serial port of the synchronized computer. A special serial port driver is needed, and essentially using the same clock adjustment algorithm as NTP, the software clock can be synchronized to an accuracy of 10μs. The main source of error is not the jitter on the received external synchronization signal, as in the case of NTP over a network, but the delay due to hardware and software processing and the jitter due to system noise and clock reading precision. These limiting factors are the same as those of any clock running under the noise inherent in a `Unix`-like multi-tasking operating system.

For monitoring high bandwidth links an accuracy of $10\mu s$ is not sufficient. In these cases special measurement cards like the DAG series [9], capable of synchronizing to a GPS signal with much greater accuracy, can be used, or alternatively, a purely software solution can be retained, as we now describe, provided the traffic rate is not too high.

3. A NEW SOFTWARE CLOCK

An overview of the nature of the new clock we propose has already been given in the introduction. In this section we define it more precisely and give a thorough account of its features.

3.1 The Clock

In section 2.3 we saw that over time periods up to and beyond 1000 seconds, the stability of both the standard oscillator and the CPU oscillator was good enough to make a simple skew model perfectly adequate. More precisely, over these timescales we found that the following model can be applied

$$\theta(t) = \theta(0) + \gamma * t + \eta(t), \tag{3}$$

where $\eta(t)$ is a stationary noise of zero mean and amplitude (standard deviation) around 5μs, corresponding to operating system and hardware noise, the latter being essentially interrupt latency. Over these timescales, errors in offset due to failure of the model, ie where oscillator instability is such that we can no longer take γ to be constant, are below 0.1PPM. From table 1 we see for example that for an IAT measurement over an interval as large as a second, this is well under 1μs. Errors in measurement are of course still made because of the additive noise $\eta(t)$, and are relatively worse the smaller the measurement interval. Such errors will be suffered in common with any other clock, including a GPS synchronized standard software clock.

Of the two oscillators, that of the CPU is clearly preferable because of its higher resolution. Furthermore, in future PC architectures the presence of a CPU is a reasonable assumption, as is an ongoing increase in resolution. Another clear advantage is the hardware updating of the TSC, and that reading a register is likely to remain one of the fastest operations available.

From the structural advantages of the TSC outlined above we must extract a practical software clock and determine its properties. In the next subsection we do this under the assumption that we already have an estimate of the average TSC period p with an accuracy equivalent to a skew of γ. The following subsection will deal with the estimation of p.

3.1.1 Performance with 'Best' Rate

Through measurements using our reference source on several test machines, we have measured values for the cycle rates r accurate to a skew better than 0.1PPM. Such values are the best possible in the sense that the offset error they generate only becomes significant above timescales where the simple skew model above fails to hold in any case. It is remarkable that the values found have remained constant over many weeks, and through many reboots of the test systems. Indeed figure 7 exhibits this in a continuous measurement over 100 days. We have already displayed these small skews, indirectly, via the rate stability plots for the TSC based interpolation–clock in figure 2. It is this 'best' rate which is on display in the top plots in figure 4, the simple skew in figure 5, and indeed in all plots involving TSC based clocks in the paper.

Assume then that r, or equivalently, a value of the CPU cycle period $p = 1/r$, is given. The basic timestamping operation is simply the reading of the TSC register and writing it to memory. From this point two main approaches can be taken:

Timestamps in Postprocessing : The aim here is maximum accuracy, and to allow multiple timestamps to be obtained very rapidly if required. Thus only the raw information of the key events to be timestamped is kept. The actual timestamps in seconds are calculated separately either as a carefully controlled low priority process in parallel, or after the measurements are complete. This approach is appropriate for an active measurement receiver, where the volume of received packets is low, and one can afford to wait.

Alternative System Clock : Here the aim is to have a multipurpose alternative system clock available at all times. The basic timestamping operation then needs to be followed by a rapid and accurate conversion phase to a timestamp in time units. Both of these require care in implementation issues, principally integer arithmetic which preserves the required level of precision.

Of the two approaches above, the first may be performed in arbitrary precision arithmetic and is straightforward. For completeness, and to aid those who may wish to implement the clock, we quickly outline an implementation of the second. For simplicity we assume a nominal CPU speed of 1 gigahertz, and hence a cycle period of approximately 1 nanosecond. In either case, the 'conversion constant' p, the CPU crystal period which converts the TSC register 'timestamp' to time units, is stored not as a single real number but as the ratio of two integers: $p = d/c$, where c is a number of cycles, and d the duration of them. The TSC register is 64 bits long, which at 1ns per bit corresponds to 585 years. Whilst this covers a sufficient range for the moment, it is more scalable to periodically update a counter of some larger time unit – we use microseconds, and to fit the remainder into a smaller integer.

The advantages of the TSC counter and the principles of the above two ways of using it are naturally scalable to higher processor rates, and can be implemented in parallel to the normal software clock. In the case of replacing the latter however the TSC clock cannot be given a standard timestamp interface, for example by replacing the internals of `gettimeofday`, without limiting its resolution to 1μs.

The performance of this clock is very good, a rate precision of at least 0.1PPM, a resolution of 1ns, a rate stability equal to that of the underlying hardware, and an exceptionally low processing overhead for timestamping (less than 50ns for a 600Mhz test machine).

3.1.2 Approaches for Rate Estimation

The accurate estimation of the cycle period p is an essential component of our approach, and of course requires the use of a reference time source in some form. One approach could be to measure p very accurately once by attaching a reference source, and thereafter to use that value. This is not satisfactory as it is not clear that obtaining such a reference source for a one-off measurement, or to support a very infrequent (re)calibration regime, is not just as inconvenient and expensive as the permanent reference source solution we are seeking to avoid. Furthermore, oscillator rates do change over time for a variety of reasons. Moving from large to small timescales, these include:

months/years ageing of the oscillator hardware

weeks/months changes in the environment, power supply, airconditioning

days/week cycles of activity, including systematic temperature variations

hours/day local temperature variations (activity, weather..).

Thus, although observations of our test system show very little variation over a period of months, one could never be certain that the skew had not changed significantly for some reason. It is therefore desirable to be able to re-measure p on a regular and automatic basis.

Accessing a reference source via a network is clearly a very convenient solution as it eliminates the need for special hardware and allows for automated, regular recalibration. In the next two sections we describe two quite different ways of doing so.

3.2 An NTP Based Rate Calibration

The existing network of GPS synchronized NTP primary servers is an obvious candidate for gaining 'second hand' access to a reference time source. We exploit the fact that the software clock uses NTP to track offset to within some bound $\Delta\theta$.

The idea is that a fixed bound can be made to correspond to as small a relative error in rate estimation as desired, simply by measuring over a large interval. From table 1, it is seen that $2\Delta\theta = 2$ms (an optimal 1ms error at each end of the measurement interval) corresponds to a target skew of 0.1PPM by measuring over 5.55 hours. A simple way to gain access to the NTP time estimates is to read the software clock. Thus, *we can obtain a period estimate by simply choosing a time interval D_T in excess of $2\Delta\theta * 10^7$, setting d to the difference in the software clock timestamps, and c to the difference in the TSC.* The 1μs resolution of the software clock is not a limitation here as the difference required is in the millisecond range.

To the basic period estimation it is desirable to add an ability to follow its possible evolution. This can be done through periodic windowed estimates, taken with a large degree of overlap to ensure that the 'tracking' does not introduce its own dynamics. The intuition and aim of this heavily damped algorithm is that a quasi-static skew parameter γ of a simple constant skew model be measured.

More concretely, we continue with the example with $\Delta\theta = 1$ms. New duration $d'(i)$ and counter $c'(i)$ measurements are made every $D_I = 96$s. From figure 2 we see that this is a timescale over which the simple skew model is valid. Up to 256 past values are stored, covering a total estimation duration of $D_T = 96 * 256 = 24576$s or 6.8 hours (6.8 is thus a convenient figure which exceeds 5.55h). From this database estimates based on measurements D_T apart are used to form the final estimates: $d(i)$ and $c(i)$, $i = 1, 2, \cdots$, at times $t = i * D_I$ seconds, corresponding to the rate estimates. The window size is steadily increased until it reaches D_T where it remains constant. In this way an initial estimate becomes available after only D_I seconds.

The time series of period estimates, $p(i) = d(i)/c(i)$, is shown in figure 7 over a period of 100 days. On this scale the startup phase is hidden in the vertical axis. The variation of the period

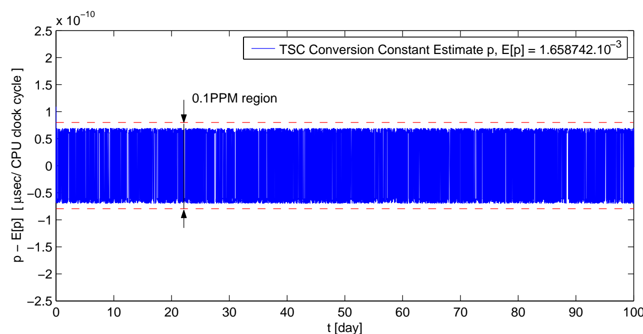

Figure 7: The (centered) CPU period series $p(i)$ over a 100 day period, showing a 0.1PPM stability.

estimates is very low. Indeed, the visible discretization is due to the fact that the only variations seen were in the last bit in d, with very rare exceptions due to exceptional NTP events. Such events could be filtered out with a more sophisticated algorithm. The error in the period estimate, measured as its variation over its value, is seen to be around $1.5 * 10^{-10}\mu$s per clock cycle over $1.82 * 10^{-3}$, or ≈ 0.1PPM, as designed for. It must be emphasized that these results are for the optimal conditions of $\Delta\theta = 1$ms, which can be very optimistic. Provided $\Delta\theta$ is known however, D_T can be set appropriately and the method will work.

3.3 A Remote Rate Calibration Technique

A disadvantage of the NTP based rate synchronisation is that the offset bound $\Delta\theta$ is not really known, making it difficult to know and control the accuracy of the corresponding estimate. Our 'remote calibration' technique applies to situations where we have more control: a reference source at one measurement point. The aim is to use it to measure the cycle period p for machines at other locations.

The idea is that by sending out a stream of accurately timestamped packets to be received by an accurate software receiver monitor, p can be measured by comparing the reference departure timestamps (in seconds) with the TSC register values recorded by the receiver monitor, after filtering out the network induced variability.

We used one local machine to remotely calibrate another machine in our local network, via a node in Hungary which reflected the packets. This provided a difficult test for the method (40 hops in total), whilst allowing us to use our reference infrastructure in a convenient way. We used a TSC-RT-Linux sender with a DAG card monitor, and the receiver ran a TSC-RT-Linux monitor (see below) timestamping arrivals with the TSC value, also with a DAG monitor for verification. A 12 hour experiment was conducted consisting of a periodic test stream of 4320 packets with an inter-departure time of 10s. The raw data, TSC values against sender timestamps, exhibits a striking linear behaviour whose slope corresponds to p, and was measured as 602.54537 Mhz. Using this value to convert the TSC values to seconds and comparing with the *receiving DAG* timestamps for verification, we see in the lower plot in figure 8 that the residual variation in rate is well under 0.1PPM. A precise value

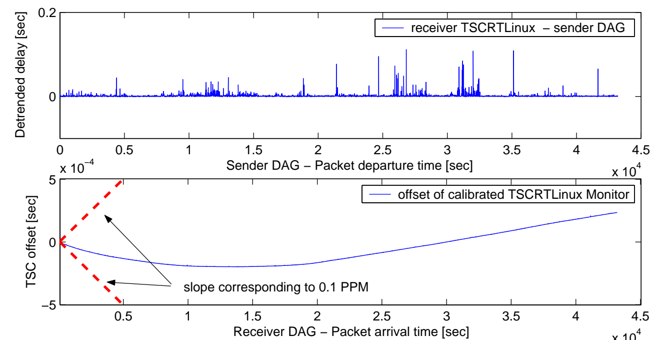

Figure 8: Calibration over 40 hops. Top: the receiver TSC readings as a function of measurement packet departure times. Bottom: The residual rate variation, using the measured p value, is in the 0.1PPM range.

for p allows us to return to the raw data, convert the TSC values to arrival timestamps in seconds, and subtract the DAG sender times to obtain the end-to-end delay for each packet. This 'detrended' delay series is shown in the top plot, where the magnitude of the network delay variations are easily appreciated.

Intuitively, prior to the estimation these variations were filtered out by following the densely populated part of the slope. To achieve this, we formed a histogram of the inter-arrival TSE values and selected those packets which were in a quartile range Δq about the

median. The slope was then obtained via least squares using only these values. The estimate was found to be very insensitive to the value of Δq in a wide range.

3.4 Offset Synchronization

The question of the synchronization of offset is quite different to that of the synchronization of rate which we have discussed so far. Nonetheless, it is relevant to consider this question also, if only briefly.

Let us assume that TSC-RT-Linux monitors are installed in the origin and destination of a one-way active probing experiment. Over timescales of importance to the measurement of IAT's and delay variation: 10's to 100's of milliseconds up to a second, the error in offset due to the 0.1PPM skew is well under 1μs (table 1). Thus the rates are so well synchronized that we can assume that they are both correct and equal. The difference in offset between the clocks however will not be zero. This is important to know in a few cases, for example an absolute delay or latency figure is needed to predict the performance of time critical applications such as voice over IP and distributed gaming. However, in such applications an estimate of the delay with millisecond accuracy is adequate. This is because any technique which analyses delay series in detail inevitably considers delay variation (and IATs), the absolute delay appearing as a simple constant.

For offset synchronization NTP is an obvious solution, as its aim is exactly that, to control offset. Nonetheless, we have seen that the usual NTP + software clock combination suffers from a variety of sources of timing error, which the tracking algorithm must content with, which can lead to rapid and dramatic changes in rate (and even offset), at the whim of unfortunate network conditions [12]. In contrast, the TSC clock is gently poised around a skew value of zero, and so offset errors can only accumulate very slowly. By interfacing NTP with the TSC-RT-Linux monitor to use the latter's inherent stability to filter the sometimes excessive recommendations of the former, it should therefore be possible to reduce the sources of variability to the network only. Estimation based on minima of one-way and round-trip delays should then be far more successful than usual in filtering out network delays in this more reliable environment. By using a scheme of this type, combined with an effort to connect to nearby, primary NTP servers, offset errors for delay estimates can be kept below 1ms. The feasibility of this approach has been verified in our recent work.

4. A HIGH PERFORMANCE MONITORING SOLUTION

We have shown that a TSC based clock with an accurate calibration can easily equal the performance of the traditional GPS + software clock solution. Aside from the clock itself however, the other dimension of timestamping is being able to read it as close as possible to the events being timestamped. Optimising this step is clearly application dependent. A general purpose system call suffers from system noise which lowers the inherent accuracy of a TSC clock to little better than a simple NTP + software clock solution. Only when NTP loses synchronization, when networking conditions are highly variable, or when NTP–Date is used, is the TSC clock clearly superior under such circumstances. In other words, the inherent rate stability and accuracy of the TSC clock is masked by noise under Linux, leaving only its reliability as its main advantage. This was clearly seen in figure 4 where the offset differences due to rate changes in the bottom plot were on the order of microseconds, whereas the system noise was ten times this. However, any drop in noise level immediately allows the other benefits of the clock to be seen.

To see and benefit from this, we optimise for network measurement by placing the timestamping operation (TSC register reading) early in the driver code, a solution we call the *TSC-Linux monitor*. Under Linux (and BSD) however, operating system noise, chiefly processor scheduling, still plays a role. This is greatly reduced under RT-Linux, and we refer to the combination as the *TSC-RT-Linux monitor*.

To illustrate these consider figure 9 where offset and rate error (over $\tau = 1$ms) is displayed for the traditional *SW-NTP-tcpdump* monitor, the TSC-Linux monitor, and the TSC-RT-Linux monitor (both of the TSC monitors use the 'best' rate). For the SW-NTP-tcpdump and TSC-Linux monitors spikes due to scheduling can be seen both in the offset and its difference, though it is much reduced in the TSC-Linux monitor. In contrast, in the bottom plot the TSC-RT-Linux monitor shows no large spikes. The increase in performance from top to bottom is remarkable (note the scales). The TSC-RT-Linux monitor has offset variations of the order of only 1μs, and rate variations in the 0.1PPM range. This is much better than the common GPS and software clock solution, but is achieved entirely with software. The periodicity visible in the TSC-RT-Linux offset plot is of the order of the offset error of the DAG clock itself, and indeed may be caused by the DAG synchronizing to its own GPS pulse per second input.

5. CONCLUSION

In this paper it was explained why rate performance, rather than absolute offset synchronization, is the key requirement of a software clock for most purposes of network measurement. From this point of view the drawbacks of existing solutions were explored in detail, including unsynchronized standard software clocks, the network based NTP controlled software clock, and the increasingly common GPS synchronized software clock. The disadvantages include being subject to excessive system noise, highly variable rate on small timescales, and/or excessive cost and inconvenience. As network measurement using inexpensive PC infrastructure becomes increasingly common and demanding, it is important to provide a clock which has a high rate accuracy, but without the overheads of special hardware.

A software clock was presented with excellent rate performance, based on the TSC register which counts CPU cycles. It offers resolution of the order of nanoseconds, and very smooth rate: over timescales up to 1000 seconds a simple skew model of rate is valid, with a skew value which can be determined to within 0.1PPM. On longer timescales the variation in the skew parameter is also of this order. Estimates obtained for several different machines, running Linux or BSD, were found (using our local reference clock, a GPS synchronized DAG3.2e card) to stay in this range on a timescale of weeks or even months. Using these estimates, rate accuracy was limited by that of the system noise, and is equal to the GPS + software clock solution, but without the disadvantages of using GPS.

To profit from the inherent high stability of the CPU crystal its period p must be measured to an accuracy of 0.1PPM without the need of a reference clock (such as a GPS receiver) at the point of calibration. We discussed in detail two methods for achieving this. The first exploits the network of NTP servers. By measuring the period over a sufficiently long time interval, the bound in offset

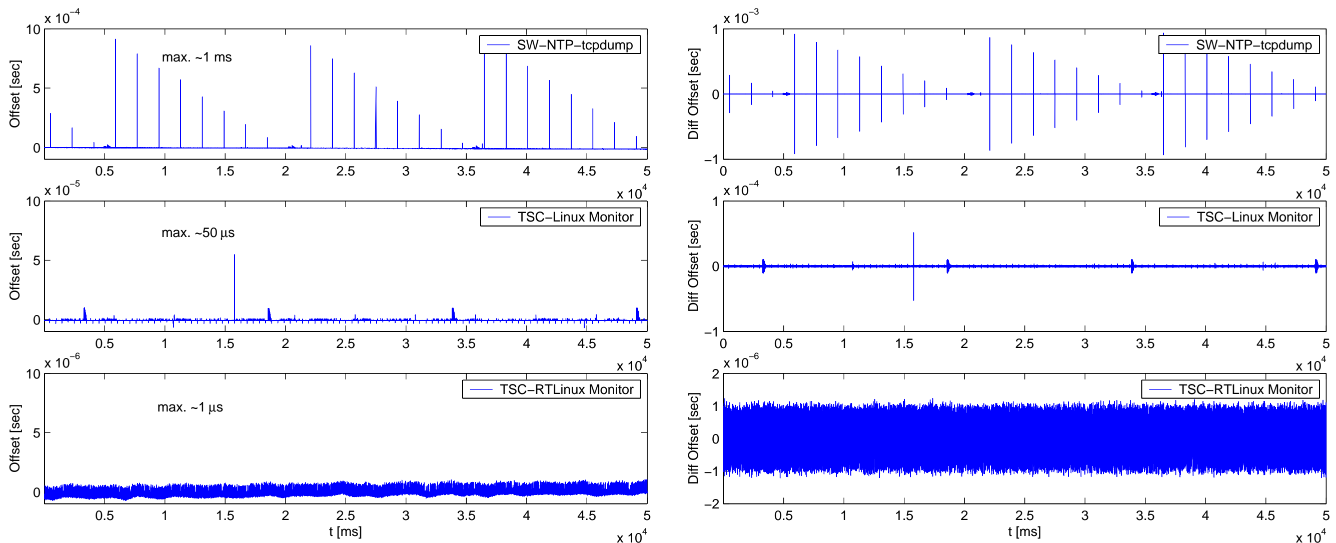

Figure 9: Offset and rate error with $\tau = 1$ms for three network measurment monitors. Top: SW-NTP-tcpdump, Middle: TSC-Linux, Bottom: TSC-RT-Linux. Note scale changes.

provided by the NTP synchronised software clock can be made as relatively small as desired. The second assumes that one controls a reference clock at one point, and shows how it can be used to accurately measure p for remote machines, via packets sent out across the network from the reference source. We demonstrate how this was done to an accuracy of 0.1PPM, despite the presence of very strong network noise over a calibration route 40 hops long.

Apart from an accurate clock, accurate timestamping requires that the clock can be accessed quickly. In the context of network measurement, we show how this can be optimised to produce very accurate TSC based traffic monitors. The main optimisations are lowering system noise by placing the timestamping operation in the driver code very close to the network interface, and avoiding scheduling issues by using a real-time operating system. Linux and RT-Linux where the systems used. The resulting monitors: the TSC-Linux monitor and the TSC-RT-Linux monitor, have rate performance exceeding that of the equally inexpensive NTP + software clock + tcpdump solution. The TSC-RT-Linux monitor has an offset variability of only 1μs and a rate stability of 0.1PPM.

6. ACKNOWLEDGMENTS

This work was supported by Ericsson. The authors thank the WAND group for measurement access.

7. REFERENCES

[1] D.L.Mills. Internet time synchronisation: the network time protocol. *IEEE Trans. Communications*, 39(10):1482–1493, October 1991.

[2] D.L.Mills. Precision synchronisation of computer network clocks. *Tech.Report 93-11-1, Electrical Engineering Department, University of Deleware*, November 1993.

[3] D.L.Mills. The network computer as precision timekeeper. In *Proc. Precision Time and Time Interval (PTTI) Applications and Planning Meeting*, pages 96–108, December 1996.

[4] D.L.Mills. The nanokernel. In *Proc. Precision Time and Time Interval (PTTI) Applications and Planning Meeting*, pages 423–430, November 2000.

[5] C. Dovrolis, P.Ramanathan, and D. Moore. What do packet dispersion techniques measure? In *Proceedings of IEEE Infocom'01*, April 2001.

[6] S. Kalindidi and M. J.Zekauskas. Surveyor: An infrastructure for internet performance measurements. In *Proc. of INET'99*, June 1999.

[7] K. Lai and M. Baker. Measuring link bandwidths using a deterministic model of packet delay. In *Proc. of ACM SIGCOMM'01*, pages 283–294, August 2001.

[8] C. Liao, M.Martonosi, and D.W.Clark. Experience with an adaptive globally synchronizing clock algorithm. In *Proc. of 11th ACM Symp. on Parallel Algorithms and Architectures*, June 1999.

[9] J. Micheel, I. Graham, and S. Donnelly. Precision timestamping of network packets. In *Proc. of the SIGCOMM IMW*, November 2001.

[10] A. Pásztor and D. Veitch. A precision infrastructure for active probing. In *Proc. of PAM2001, Workshop and Passive and Active Measurements*, pages 33–44, April 2001.

[11] A. Pásztor and D. Veitch. The packet size dependence of packet pair like methods. In *Proc. of IWQoS'02*, May 2002.

[12] V. Paxson. On calibrating measurements of packet transit times. In *Proceedings of ACM SIGMETRICS*, June 1998.

[13] H. Uijterwaal and O. Kolkman. Internet delay measurements using test traffic: Design note. *Tech.Report RIPE-158, RIPE NCC*, June 1997.