A User Friendly TSC Clock

Erik Corell[1], Philip Saxholm[1], and Darryl Veitch[2]

[1] Lund Institute of Technology, Lund University, Lund, Sweden
{cii99ec7,cii99ps7}@i.lth.se
[2] Australian Research Council Special Research Center for Ultra-Broadband Information Networks (CUBIN), an affiliated program of National ICT Australia Dept. of Electrical & Electronic Engineering, The University of Melbourne, Victoria 3010, Australia
d.veitch@ee.unimelb.edu.au

Abstract. Recently a new software clock and synchronisation algorithm based on the TSC register (clock cycle counter) was developed, with several advantages over the existing system clock. However, as it uses a modified kernel to support driver timestamping, installation is non-trivial, limiting its use. We present a modified TSC clock without the need for kernel modifications, using only user-level timestamps and existing system kernel timestamps exploited in a careful way. Using weeks of test data, we show how the system performance is virtually identical to that of a kernel implementation. Compared against a GPS synchronised DAG card reference, it performs very well under both BSD and Linux. We also show how the system can replace `ntpd` to improve the existing system clock. The software allows for significantly improved timestamping for both packet and internal system events, and is trivial to install. It is publicly available.

1 Introduction

In [1] (see also [2]) a new software clock was introduced, the *TSC-NTP* clock, whose hardware basis is the Time Stamp Counter (TSC) register, which counts cycles of the CPU oscillator. Like the existing system clock supported by most common kernels and the associated `ntpd` (Network Time Protocol Daemon), a combination we denote by *SW-NTP*, it makes use of NTP servers to synchronise itself inexpensively across a network, but uses new algorithms designed from scratch to take advantage of the high stability of modern hardware. Using months of data, the TSC-NTP clock was shown in [1] to be extremely accurate for the measurement of time differences, very accurate when measuring absolute time, and robust to many factors including loss of connectivity, network congestion, and even bogus server timestamps.

Part of the BSD implementation employed in [1] benefited from kernel modifications allowing the TSC to be read in the driver and made available to the user level. Driver or kernel level timestamping substantially reduces process scheduling and interrupt latency errors polluting timestamps, and is a significant advantage for packet timestamping, including the NTP packets involved in the synchronisation algorithm itself. The existing NTP synchronised software clock, the SW-NTP clock, benefits from kernel level timestamping, as it is built into existing kernels. Since this is not the case

for the TSC-NTP clock at present, using it involves the onerous task of installing a modified kernel and driver, which few users are willing to contemplate.

The goal of this work is to explore the feasibility of, and to provide, a 'user-level' version of the TSC-NTP clock which works with existing kernels and drivers, allowing it to be installed with little effort, but which offers comparable performance to the current kernel version (based on FreeBSD 5.3). This is a non-trivial task because the synchronisation algorithm uses filtering based on round-trip times (RTT), and in particular on the measurement of the minimum RTT. User level timestamping compromises this in a fundamental way as we explain below. We discuss possible solutions to this problem, and describe two effective techniques based on a judicious use of user level timestamps and existing SW timestamps from the kernel. Tests of the method in the laboratory and under operational conditions show that is very effective, reducing user-level related errors to below 1μs in most cases. The resulting user version, the USER-TSC-NTP or *USER-TSC* clock, is easy to install and is available for download at `http://www.cubinlab.ee.mu.oz.au/probing/software.shtml`. The software has been tested on FreeBSD 5.3 and Linux 2.6.6 only, however being based on the *PCAP* packet capture library and user-level C code, it is insensitive to OS type or version.

One of our techniques involves disabling `ntpd`, which normally disciplines the system clock, and instead synchronising it locally to the User-TSC clock. We also report on the resulting improvements to the system clock robustness.

Section 2 provides necessary background into the SW-NTP and TSC-NTP clocks. In Section 3 the core difficulty of user based synchronisation is explained, options discussed, and a solution architecture outlined and explored. Section 4 describes the performance of the solution measured against the reference DAG solution, and Section 5 expands on the option of disciplining the SW using the USER-TSC clock, in order to enhance the performance of each. We offer directions for ongoing work in Section 6.

2 Software Clocks

In this section we provide an introduction tailored to our needs. For more details on the SW-NTP clock see [3], and for the TSC-NTP clock see [1].

2.1 Background

We denote true time, measured in seconds from some origin, by t. A real world clock $C(t)$ suffers an *offset* $\theta(t)$ from true time: $\theta(t) = C(t) - t$.

If we assume a simple pure frequency model for $C(t)$, we can define the *skew*, or error in clock rate, as the constant γ in $\theta(t) = \theta_0 + \gamma t$. It is shown in [1] that this *Simple Skew Model* describes oscillator behaviour well over timescales up to around $\tau^* = 1000$ [sec], however over longer periods clock drift is apparent, significantly influenced by temperature variations due in particular to diurnal and air-conditioning cycles.

The *oscillator stability* characterizes drift by examining the variance of the rate error $y_\tau(t) = \frac{\theta(t+\tau)-\theta(t)}{\tau}$ relative to a timescale τ, as a function of τ. Provided that the temperature environment is reasonable, rate errors do not exceed 1 part in 10^7 or 0.1 PPM (Parts Per Million) over a wide range of scales to either side of τ^*.

For clock synchronisation we exchange NTP packets with stratum 1 NTP servers, which are themselves locally synchronised via GPS or atomic clock, in client-server mode. As shown in Figure 1, we timestamp four key events: the times of departure and reception of each NTP packet from host to server, and server to host. The basic idea underlying **absolute** synchronisation is that since $t_a < t_b < t_e < t_f$, so should quality timestamps of these events. The *path asymmetry*, $\Delta \equiv d^{\rightarrow} - d^{\leftarrow}$, where $d^{\rightarrow} \equiv t_b - t_a$ and $d^{\leftarrow} \equiv t_f - t_e$ are the minimum forward and backward delays, dictates where the server timestamps lie within the RTT interval. However, since Δ cannot be measured in practice, $\Delta = 0$ is used by algorithms, resulting in an offset error of $\Delta/2$. This inherent limitation can be mitigated by selecting nearby servers, since $|\Delta| \leq t_f - t_a = r$, the minimum round-trip time.

2.2 The SW-NTP Clock

The system clock is based around the periodic interrupt cycle, of period typically 10[ms] or 1[ms], driving process scheduling. The TSC is used to interpolate times between interrupts, but is not the fundamental basis of the clock. The interrupt cycle period is obtained by counting a number of periods of another oscillator, of much lower frequency, on the motherboard.

The clock derives a nominal rate at boot time, and then adjusts itself through three mechanisms: *reset*, *skew* and *phase*, informed by filtering server timestamps obtained through `ntpd`, and system timestamps at the host, with the aim of driving $\theta(t)$ to zero.
Reset: discontinuous offsets added when extreme discrepancies in the range 128[ms] to 1000[sec] occur. They are not directly accessible at user level.
Skew: the average clock increase per interrupt cycle relative to nominal rate, spread out smoothly over the cycle. This is the main mechanism to adapt clock rate to reduce the perceived offset. It is accessible from the user level as the variable `freq` via the `ntpadjtime()` function, and is updated after each new NTP packet.
Phase: correction factors (`tickadj` variable measured in 'ticks' of 1μs) added to fine tune clock adjustments. They may or may not be applied at a given interrupt and this information is not accessible, although the maximum value is (typically a few μs).

The SW-NTP clock has access to timestamps made in the kernel, available at the user level in an operating system independent way through the *pcap* capture library. This applies to all packets filtered for, including NTP packets used in synchronisation.

2.3 The TSC-NTP Clock

The TSC-NTP clock is based on the TSC oscillator only, and exploits its high stability as described above. In sharp contrast to the SW-NTP clock, the TSC-NTP clock is built around obtaining stable long-term rate estimates, rather than sacrificing rate to track offset. It is in fact two clocks:
$\mathbf{C}_d(t)$ **for time differences below** τ^*: here rate is essential, offset irrelevant, extremely robust, very high accuracy possible.
$\mathbf{C}_a(t)$ **for absolute time (or differences above** τ^***)**: here drift must be tracked, robustness lower, lower accuracy with Δ a limitation.

The synchronisation algorithm makes use of the two server timestamps per NTP packet, but ignores the corresponding system timestamps, instead taking raw TSC timestamps in the driver. Currently this is implemented by modifying the Berkely Packet Filter (BPF) code in the FreeBSD 5.3 kernel. These *stamps* (4-tuple of timestamps) are first used to provide an estimate $\hat{p}$ of the average oscillator period p. The underlying clock is given by $C(t) = \hat{p} * TSC + \theta_0$, where θ_0 is an estimate of the offset which is not updated. An estimate $\hat{\theta}$ of the error in this clock is updated at each new NTP packet. The two clocks are then:
$\mathbf{C}_d(t)$: $\quad \Delta(t) = C(t_2) - C(t_1) = \Delta(TSC) * \hat{p}$,
$\mathbf{C}_a(t)$: $\quad C_a(t) = \hat{p} * TSC + \theta_0 - \hat{\theta}$.
By not changing θ_0 and not applying $\hat{\theta}$, the difference clock benefits from the underlying rate stability of the TSC over small to medium timescales without being perturbed by estimates of drift, which are irrelevant over those scales. By not changing θ_0, instead applying a correction only when reading, the absolute clock avoids varying its rate at the whim of imprecise offset estimates, which greatly enhances stability and robustness.

The estimates of $\hat{p}$ and $\hat{\theta}$ are based on filtering NTP packets according to their RTTs. The excess of RTT above an estimate $\hat{r}$ of the minimum RTT r is used as a basis of rejection of distorted timestamps when measuring $\hat{p}$, and as a weight when averaging estimates made over several packets in the case of $\hat{\theta}$.

3 A Solution: Kernel Timestamp Recreation

In this section we discuss the negative impacts of taking timestamps at user level, and describe our approach for circumventing them.

3.1 Solution Design

Any timestamp of a packet arrival is clearly taken after it actually arrives! however on the sending side the order is implementation dependent. Figure 1 shows the kernel TSC timestamp T^k_{tsc} being made at time t^k_{tsc}[1], **before** the departure at $t = t_a$. In our kernel implementation this is in fact guaranteed, so the kernel TSC timestamps enclose the true times. This is also generally true of system timestamping in the kernel under FreeBSD and Linux including the versions we use here.

This enclosure property is extremely important for the RTT based filtering methodology, since it guarantees that the RTT seen by timestamps lies **above** the true value r. Without this, one can have $\hat{r} < r$, resulting in poor quality timestamps being perceived as good, and good quality as poor. The central problem of user timestamping is that this requirement is not only broken on the sending side, that is $t^u_{tsc} > t_a$ can occur as illustrated in Figure 1, but transgressions are both common and severe. Another very serious problem is that process scheduling significantly delays timestamping, resulting in far fewer timestamps of good quality surviving filtering.
We examined three solutions approaches:

1 Re-engineer the synchronisation algorithm for user timestamping:
- non-trivial, prevents efficient reuse of existing synchronisation algorithm.

[1] In general, we use t for event times and T for timestamps taken at those times. We use a superscript k (resp. u) to indicate timestamps made at kernel (resp. user) level.

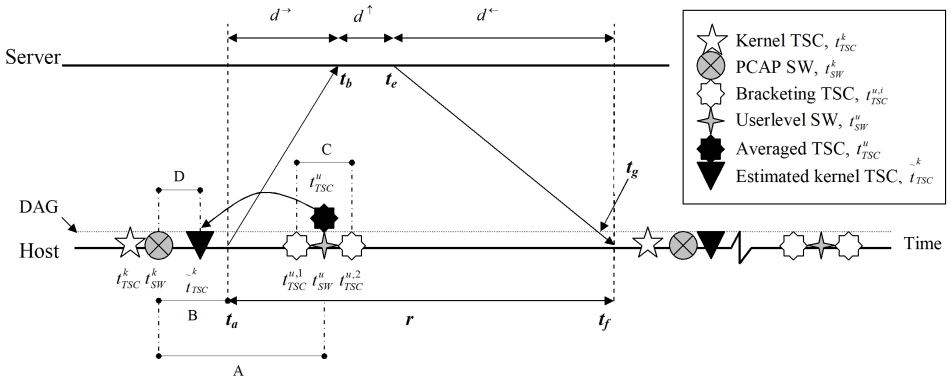


Fig. 1. Timeline of NTP packet events, and kernel and user timestamping events relevant to either the user or kernel TSC-NTP clock. Important intervals: A: delay d_{sw}, B: SW interrupt latency, C: bracket width B^u, D: backing error.

2 Control NTP packet generation at user level to ensure $t^u_{tsc} < t_a$:
- runs counter to stated aims of minimal installation effort and system impact.

***3** *Recreate* an estimate $\tilde{T}^k_{tsc}$ of the missing kernel level timestamp:
- avoids above drawbacks, allows modular approach: preprocessing phase passing compensated timestamps to existing synchronisation algorithm.

We opt for option 3. It is feasible because in fact kernel timestamps **are** available: the system timestamps obtained via `pcap` (see t^k_{sw} in Figure 1). The idea is:
Subtracting the `pcap` timestamp from a second SW timestamp at user level yields the correspondence between the kernel and user timestamps. A user TSC timestamp can then be backdated to learn what it would have been at time $t = t^k_{sw}$.

This approach remains challenging because of the varying nature of the SW-NTP clock, and the difficulty of knowing the correspondence between its rate and that of the TSC-NTP. Errors can occur, and it is essential to prevent them from resulting in $\hat{r} < r$. We apply two levels of quality assessment of our estimated kernel timestamp $\tilde{T}^k_{tsc}$:

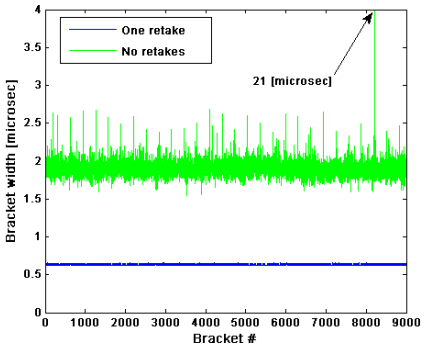
Dangerous : evidence that an estimate is of danger to $\hat{r} \rightarrow$ **discard packet**
- minimal impact as synchronisation algorithm highly robust to loss.

Unreliable : evidence that the estimate is not excellent $\rightarrow$ **signal warning**
- existing API to synchronisation algorithm allows for quality warnings.

3.2 Timestamp Backing Algorithm

We first describe one component of the algorithm, timestamp *bracketing*.

We wish to simultaneously read the TSC and SW clocks at the user level, but as this is impossible, we read the TSC, then the SW, and the TSC again, obtaining timestamps $T^{u,1}_{tsc}$, T^u_{sw}, $T^{u,2}_{tsc}$, and set $T^u_{tsc} = (T^{u,2}_{tsc} + T^{u,1}_{tsc})/2$. If a scheduling timeout occurs, the *bracket-width* $B^u = \hat{p} * (T^{u,2}_{tsc} - T^{u,1}_{tsc}) > 0$ will be orders of magnitude larger than the time needed, $\approx 1\mu$s, to call the user clock function `gettimeofday`. If a delay in context switching occurs, it can be a small factor larger. In either case, by retaking the




```
Blim = 0
do{                          //
   TSC1 = rdtsc()            // T^{u,1}_{tsc}
   SW = gettimeofday()       //   T^u_{sw}
   TSC2 = rdtsc()            //     T^{u,2}_{tsc}
   B = TSC2 – TSC1           // B^u
   Blim += Binc              // ensure exit
} while (B>Blim)             //
Blim –= (Binc+Bdec)          // keep B tight
TSC1 += B/2                  // T^u_{tsc}
```

Fig. 2. Bracketing. Left: A single retry nearly always produces a tight bracket. Right: The algorithm ensures we always do (we use Binc$= 64$ and Bdec$= 1$ cycles (raw TSC units)).

triplet a stable value is quickly and reliably found. Due to the high stability of the CPU oscillator, the error in measuring B^u is well under 1[ns] even for very poor $\hat{p}$.

We adaptively set thresholds for acceptance of a bracket to make the algorithm self-calibrating across different systems. The details of the algorithm, which is guaranteed to terminate, are given in Figure 2. The results show the difference in B^u when retakes are taken once, compared to not at all: a single retake provides an impressive performance gain. Over lengthy trials (at low system load) the algorithm produced $\{0, 1, > 1\}$ retries with proportions $\{0.984, 0.016, < 0.00001\}$. In what follows, bracketing is always used to produce the user comparison pair (T^u_{tsc}, T^u_{sw}). At high loads, the proportion of retries would increase, however we do not expect this to be significant. We describe below how to handle excessive delays between the user and kernel timestamps. These should remain rare unless the system is permanently at very high load.

The objective of the *backing algorithm* is to estimate what the TSC would have read at time t^k_{tsc} based on a user TSC timestamp T^u_{tsc} and the SW timestamps T^k_{sw} and T^u_{sw}:

$$\tilde{T}^k_{tsc} = T^u_{tsc} - \frac{c}{\hat{p}} * (T^u_{sw} - T^k_{sw} - j) \tag{1}$$

where $D_{sw} = T^u_{sw} - T^k_{sw}$ estimates the delay $d_{sw} = t^u_{sw} - t^k_{sw}$ between the system timestamps. The variables c and j correct the correspondence between the two clocks. Even if d_{sw} were a huge 1[sec], and $\hat{p}$ was out by a very large 1PPM, the resulting error due to the USER-TSC clock would be only 1μs. For simplicity therefore we ignore errors in $\hat{p}$ until the next section.

The factor c is the number of TSC seconds per one SW second at the time of measurement. It corrects for *skew* errors in the SW clock. Unfortunately, c, although strongly related to the accessible skew parameter `freq`, is not simply equivalent to it for several reasons. Estimating it well would necessitate independently measuring the nominal rate of the underlying oscillator and add considerable complexity to the algorithm. As a result, we use the approximate value of $c = 1$. Since d_{sw} is typically small, so will the error be in most cases. It not, we act as follows:
If $D_{sw} > 1$[ms], which typically occurs in 0.1% of packets, we signal *warning*, and

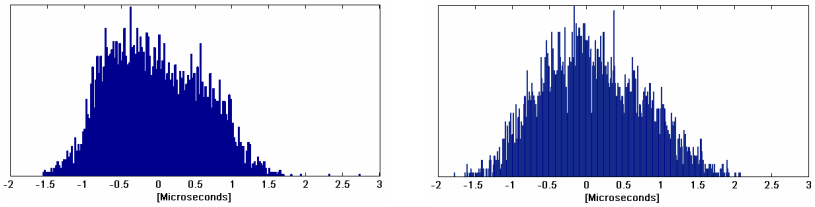


Fig. 3. Linux Backing results. Emulation using SW-NTP (left), SW-TSC (right)

if it exceeds 10[ms], we signal *dangerous* and discard the packet (about 1 packet per 10000 (FreeBSD) or 5000 (Linux)).

Offset errors should be corrected by j, however since both *Reset* and *Phase* jumps are not directly observable, j cannot be calculated, so we set $j = 0$. Instead, we detect serious errors via sanity checking procedures which signal *dangerous* and discard the packet. Note that resets can be of either sign, but are huge ($>128\mu$s) and rare, and so allow very reliable detection, whereas phase adjustments are common but their accumulation between NTP packets could be small or very large (many [ms]). In addition to the above checks, which are also relevant for offset if $j > 0$, we add:
Discard if $D_{sw} < 0$, or if there is a discrepancy in RTTs calculated with TSC and SW timestamps. There are three scenarios according to where the offset jump occurred:
RTT packet in flight: backing unaffected, so TSC timestamps and RTT good. SW is affected, it's RTT will be different $\implies$ can detect $\rightarrow$ discard.
Between SW stamps at receiver: backing unaffected at sender but not at receiver $\implies$ bad RTT from TSC, but SW RTT good $\implies$ can detect $\rightarrow$ discard.
Between SW stamps at sender: backing unaffected at receiver but not at sender $\implies$ TSC RTT bad. SW also affected at receiver, RTT also bad, but with same direction and size, $\implies$ can't detect.

Since SW-NTP makes decisions after NTP packets return, only the second scenario should actually occur, and even then very rarely. We encountered no examples of undetected offset errors in several weeks of testing on different machines and systems, using a detection threshold of 1[ms], however there is no guarantee errors cannot occur.

3.3 Validation

In a user level implementation there are no kernel TSC timestamps against which to validate our estimate. We address this in two ways: *emulation*, and *comparison kernel*.

We emulate the kernel timestamp at user level simply by reading the SW and using the system `usleep()` function to create a loosely controlled scheduling delay and context switch, after which we perform bracketing on a second SW timestamp. We also bracket the first 'kernel' SW timestamp and compare it against the backing algorithm. Although at user level we used `gettimeofday` rather than the kernel function `microtime`, by insisting on small 'kernel' bracket widths we avoid any potential bias.

We performed emulation under Linux and BSD on five different computers over a period of 2 weeks and the results were very consistent, with errors bounded by around ± 2 μs for BSD and ± 4 μs for Linux. A histogram of representative results under Linux is shown in Figure 3, and for BSD in Figure 4. Two version of the USER-TSC are given in each, depending on how the SW was synchronised, and the results are all comparable.

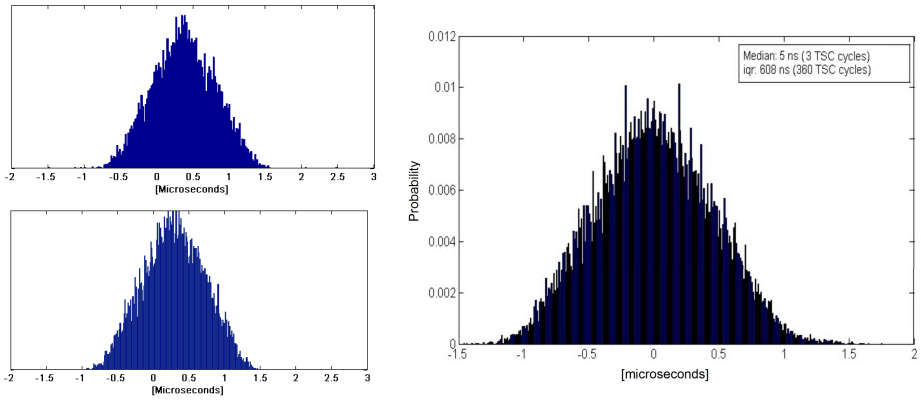


Fig. 4. BSD Backing results. Left: Emulation using SW-NTP (top), SW-TSC (bottom). Right: comparison kernel using SW-NTP (all but one value shown). Errors are of the order of 1μs.

Details on the the 'SW-TSC' alternative are explained in Section 5. We conclude that the backing algorithm performs very well with high probability on both systems.

We also modified the FreeBSD 5.3 kernel to recover TSC timestamps made in the instruction immediately following the system `microtime` call, allowing a direct validation of the algorithm in the BSD case. Although bracketing was not used, it is most unlikely that an interrupt was served between the back to back timestamps in kernel context. Figure 4 shows a median error of just a few [ns] and a narrow interquartile range under 1μs, consistent with the emulation results. These results were convincingly confirmed in a continuous two month testing run, which yielded backing error quantiles $[\min, 0.01, 25, 50, 75, 99.999, \max]$ equal to:
Sending side: $[-9.7, -1.7, -0.32,$ **-0.002**$, 0.31, 1.54, 1.6]$ μs
Receiving side: $[-9.0, -1.0, -0.10,$ **0.200**$, 0.50, 1.42, 19.]$ μs
These give us high confidence in the algorithm performance under BSD, and the utility of the emulation, and therefore of Linux performance also.

4 Performance of USER-TSC

With the backing algorithm correcting TSC based timestamps down to the accuracy of server timestamp resolution, we expect to see the USER-TSC clock perform very similarly to the kernel version. Nonetheless we directly test the clock here in case rare failures interact poorly with the synchronisation. For space reasons we focus on the absolute clock C_a only. The difference clock C_d is far more accurate and robust than C_a, and we could detect no difference between the kernel and user versions.

Figure 5 shows the comparison for absolute time. In parallel, each clock is tested against reference timestamps made by a GPS synchronised DAG card just prior to entering the host, under identical conditions. The resulting histograms are essentially identical: we have succeeding in making the impact of user level timestamping negligible.

The algorithmic performance of either absolute clock is revealed by removing the systematic bias due to path asymmetry. Using the DAG reference and $\hat{r}$, we estimate

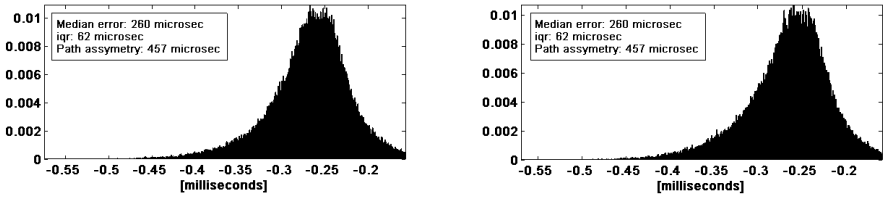


Fig. 5. Clock error of TSC-NTP using DAG: Left: kernel version, Right: user version.

Δ and subtract $\Delta/2 \approx 228\mu$s from the histograms, to obtain a medium error of $\approx 32\mu$s, consistent with results from [1]. The inter-quartile range of 62μs suffers from interrupt latency effects of the order of $\pm 20\mu$s due to this imperfect 'external' validation methodology, where the reference DAG and host timestamps are separated in time (see Figure 1). Particularly large interrupts can be detected and have been filtered out, but for space reasons we omit these details here.

5 The Next Step: controlling SW

Although the backing algorithm performed extremely well, it may be possible in rare cases for errors to corrupt timestamps, and hence $\hat{r}$. The algorithm could in principle be made more robust by actively taking control of the SW, thereby completely eliminating the problem of unobservable resets, phase adjustments, and poorly known true skew. In this section we investigate this idea, which, although involving changes to system configuration, is still at the user level. The potential benefits are twofold:

1. a more robust (and accurate) locally controlled *SW-TSC* system clock
2. a more robust *USER-TSCc* clock using SW-TSC timestamps for backing.

Using a standard configuration option, we stop `ntpd` from disciplining SW, but allow it to continue to send packets to the NTP server. We then perform a local slaving of SW to USER-TSC through varying `freq`. The algorithm, which is a simple proof of concept implementation, can be outlined as:

Local Synchronisation Algorithm

Initialisation: after USER-TSC warm-up, set $SW(t) = C_a(t)$, activate USER-TSCc.

For each NTP packet, after USER-TSCc processing, set $\theta_{SW} = SW(t) - C_a(t)$.

Adjust `freq` such that θ_{SW} would reduce to zero over 512 [sec].

If $\theta_{SW} > 256$ [ms], reset $SW(t)$.

In a nutshell, we do see evidence of increased robustness as expected, however accuracy, in this implementation, actually drops. This is best seen in Figure 6 (top left) where USER-TSCc and SW-TSC display a median difference of 0.85[ms]. This unacceptably large discrepancy can be explained in terms of the heavily damped nature of the algorithm, which was chosen for simplicity and to be conservative in terms of stability. Damping (like a moving average filter) is well known to result in a lag in tracking. We are confident that with further work, a more reactive and sophisticated algorithm could reduce this by at least an order of magnitude without sacrificing stability.

The top right plot in Figure 6 gives the absolute error (using DAG) in USER-TSCc. The distribution body is consistent with the results from Section 4, spread out by the 'lag' effect. The bottom plot is the error for USER-TSC. Here the distribution body

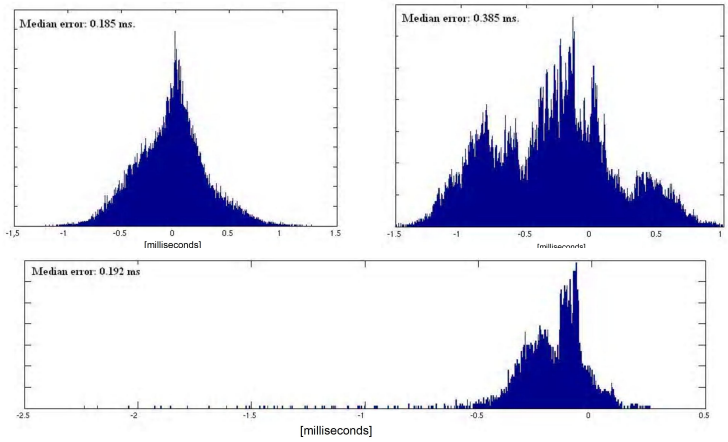


Fig. 6. Effect of local control: Top Left: SW-TSC compared to USER-TSCc, Top Right: error of USER-TSCc, Bottom: outliers present in error of USER-TSC which uses SW-NTP.

is not spread, however outliers are present which are absent under the smoother local control, demonstrating the lower robustness when using SW-NTP for backing.

Naturally, the feedback between the SW and TSC clocks in this configuration must be managed carefully to ensure stability. This is achieved through the initialisation step, a highly damped local control, and because coupling in the reverse direction is inherently weak, as backing errors are mostly proportional to d_{sw} which is small.

Finally, in terms of the effect of backing when controlling the SW, results for Linux appear in Figure 3, and for BSD in Figure 4, and are very comparable.

6 Conclusion

We have provided methodologies and software to reliably reproduce kernel level TSC raw timestamps based only on information available at user level. We have used this to create a user version of the TSC-NTP clock, USER-TSC, which does not require any kernel or driver modifications, allowing the advantages of this new clock to be used for a wide variety of applications requiring accurate and robust timestamping. We showed how robustness could be enhanced for the USER-TSC clock and the SW clock by synchronising the latter locally off the former. Results are very promising. Future work will focus on improving the local synchronisation and benchmarking the USER-TSC clock fully in extensive trials.

References

1. Veitch, D., Babu, S., Pásztor, A.: Robust synchronization of software clocks across the internet. In: Proc. 2004 ACM SIGCOMM Internet Measurement Conference, Taormina, Italy (2004) 219–232
2. Pásztor, A., Veitch, D.: PC based precision timing without GPS. In: Proceeding of ACM Sigmetrics 2002 Conference on the Measurement and Modeling of Computer Systems, Del Rey, California (2002) 1–10
3. Mills, D.: Internet time synchronization: the network time protocol. IEEE Trans. Communications **39** (1991) 1482–1493 Condensed from RFC-1129.