The Case for Feed-Forward Clock Synchronization

Julien Ridoux, Darryl Veitch, Timothy Broomhead
Department of Electrical & Electronic Engineering, The University of Melbourne, Australia
{jridoux, dveitch}@unimelb.edu.au, t.broomhead@ugrad.unimelb.edu.au

***Abstract*—Variable latencies due to communication delays or system noise is the central challenge faced by time keeping algorithms when synchronizing over the network. Using extensive experiments, we explore the robustness of synchronization in the face of both normal and extreme latency variability and compare the feedback approaches of *ntpd* and *ptpd* (a software implementation of IEEE-1588) to the feed-forward approach of the *RADclock*, and advocate for the benefits of a feed-forward approach. Noting the current lack of kernel support, we present extensions to existing mechanisms in the Linux and FreeBSD kernels giving full access to all available raw counters, and then evaluate the TSC, HPET, and ACPI counters' suitability as hardware timing sources. We demonstrate how the *RADclock* achieves the same micro-second accuracy with each counter.**

***Index Terms*—RADclock, feed-forward, synchronization, clock source, TSC, HPET, ACPI, IEEE-1588, latency, robustness**

I. Introduction

For more than 20 years, the NTP protocol and *ntpd* daemon [1], [2] have been the standard solution for clock synchronization over IP networks, in particular the Internet, where timestamps are exchanged between a host client and a remote reference clock server. It has been generally successful in synchronizing hosts to the millisecond scale, but its limitations prevent its use in new applications requiring high accuracy.

For example, to reduce costs and increase network capacity, the telecommunication industry is gradually replacing its expensive synchronized E1/T1 mobile base station backhauls with asynchronous Ethernet lines. To execute handoffs, the base stations require accurate sub-microsecond synchronization, which they cannot derive from the signal carrier anymore. Additional GPS or CDMA receivers can be used instead, but their high cost (incompatible with femto-cells for example) and unsuitability for underground deployment motivates synchronization over asynchronous networks.

The IEEE-1588 Precision Time Protocol (PTP) [3] is seen as an alternative to *ntpd* for such synchronization needs. It has been developed to support sub-microsecond synchronization over Local Area Network (LAN) islands. This comes however at the cost of replacing all LAN hardware components with IEEE-1588 enabled devices. Interest in PTP-compatible software implementations such as *ptpd* [4], [5] is rising, motivated either by cost issues, or through a desire to allow inexpensive clients to immediately take advantage of IEEE-1588 enabled devices already installed on the network.

The *ntpd* and *ptpd* algorithms share some core features. Each incorporates a *feedback* based 'servo' or clock synchronization algorithm that outputs corrections to the local clock based on new timestamp inputs from the server, as well as previous correction values. Each then disciplines the system clock of the host computer (maintained in the kernel and with its own correction mechanisms) via existing kernel support from the operating system [6]. Another feedback loop is then created since the algorithms actually use system clock timestamps as an (indirect) way to access the previous clock corrections. There are two immediate disadvantages from this design: a coupling between the kernel mechanisms (effectively a second feedback controller) and the clock servo itself, and the inability to separate the notion of absolute time from that of clock rate. As we will show, the result is accuracy which can be erratic, a descent into instability under certain conditions, and slow convergence.

A *feed-forward* design, whereby past clock corrections are **not** fed back into the next round of clock corrections, the latter being based only on the filtering and processing of server timestamps and raw hardware counter timestamps (the counter forms the hardware basis of the host clock), inherently avoids these problems. For example it is a key principle of the *RADclock* algorithm (Robust Absolute and Difference clock) [7], [8], [9], [10], [11], [12], [13], which has been shown to be a highly accurate and robust client-side solution for Internet clock synchronization. Until now however there has been a serious practical issue inhibiting feed-forward approaches: a lack of kernel support. This has resulted in a need for one-off kernel modifications, and has restricted the choice of hardware counter to the TSC (Time Stamp Counter), rather than for example the HPET (High Precision Event Timer). The TSC counts CPU cycles and is a high resolution counter with many advantages, but it has important limitations in some hardware architectures with features such as power management, frequency stepping, and multi-cores, which the HPET does not share.

The first contribution of this paper is to build the case for feed-forward approaches, represented by *RADclock*, by using a set of careful timing experiments to highlight their advantages over the feedback incumbent, represented by *ptpd* and *ntpd*. The second contribution is to precisely identify the kernel support required to allow feed-forward timing algorithms to compete on an equal footing, and the third is their implementation (freely available at http://www.synclab.org/radclock) under both FreeBSD and Linux. Existing kernel mechanisms allow any counter available in the hardware to be selected, but only provide access to a transformed version of the chosen counter's value, tailored to the needs of feedback algorithms. Our kernel extensions provide direct access to the raw, full value of each counter for the first time, both in the kernel and from user space. The fourth contribution is to exploit this access to characterize and compare the counters as hardware timing sources. We provide for the first time precise measure-

ments of their stability (in the precise timing sense of Allan deviation), and their access latency both from kernel space and user space, under both nominal and stressed conditions. Finally, we compare the performance of the *RADclock* across the different counters. We find that the differences are very small, falling under the noise of the measurement methodology itself.

This paper combines, extends and synthesizes the work of the conference papers [14], [15]. The main extensions are the inclusion of counter characterization results for Linux as well as FreeBSD, and extensive performance and robustness results for *ntpd* as well as *ptpd*. The paper is structured as follows. Section II reports on the impact of variable latencies on *ptpd*, *ntpd* and the *RADclock*, and highlights the need for the robustness of a feed-forward based synchronization algorithm. In Section III, we present a minimal set of kernel extensions to support any feed-forward algorithm based on any available hardware counter. Section IV evaluates the most common counters as timing sources, comparing their stability and access latency under various scenarios. Finally, in Section V we discuss the latency of the new kernel support itself as seen from kernel and user space, before testing the impact of counter choice on *RADclock* performance.

II. Impact of latency variability

We begin in II-A with a brief description of the *ptpd*, *ntpd* and *RADclock* algorithms, and a review of prior performance studies. We describe our methodology in II-B, and use it to compare *RADclock* and *ptpd* over a LAN in II-C, focusing on the impact of network and system latency variability on clock variability. These latencies, caused primarily by hardware and software queueing and scheduling, are **the** problem that a synchronization algorithm has to overcome in trying to correct the drift of the local oscillator, as they effectively corrupt the raw data used by the algorithm, namely timestamps. Section II-D compares *RADclock* and *ptpd* in both LAN and WAN environments. Finally, Section II-E briefly discusses the measurement of time differences using all three clocks.

A. The Synchronization Algorithms, and Prior Evaluations

ptpd The IEEE-1588 standard does not specify *how* to compensate for local oscillator drift, but the common practice is to use a Proportional-Integral (PI) controller fed by the timestamps exchanged with the reference server, and *ptpd* uses a similar approach [5]. Using local timestamps taken in the kernel, *ptpd* tries to correct both clock rate and clock error (difference from true time) simultaneously.

Both [16] and [17] examined the impact of network latency jitter, quantization, and temperature on the performance of the PI controller. In [16], each is considered using very simple models, such as a constant rate error resulting from a temperature change. A Matlab based simulation study is presented for a set of clients in a line topology. In [17], the performance of a single client under cross-traffic congestion is simulated in the OMNET++ simulator. In both studies, clock errors are derived from the simple models used but there is no use of real data or more realistic non-linear drift and noise.

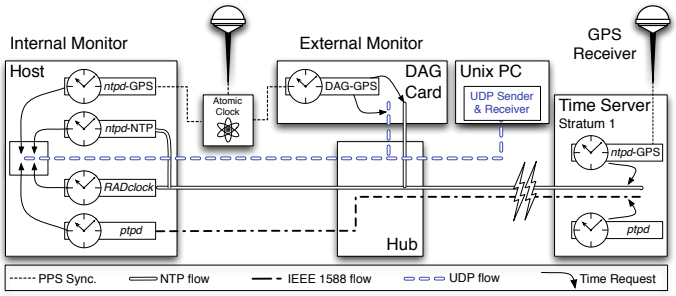

Fig. 1. Testbed. Timestamps, triggered by UDP packets, are taken internally by clients in the Host, and externally in the External Monitor.

ntpd Relying on local timestamps taken in user space, *ntpd*'s algorithm implements a sophisticated PLL/FLL (Phase/Frequency Lock Loop) [1], which having been refined over the past 20 years, is far more complex than the PI controller of *ptpd*. The design of *ntpd* in detailed in [18].

Most of the published performance results for *ntpd* can be found in [18], [19]. While providing a thorough account of the local oscillator characteristics for the purpose of the PLL/FLL design, [18] provides mainly performance results based on emulation. It does not evaluate *ntpd* against an independent reference in a real environment. Despite its age, we know of no independently validated performance results for *ntpd* approaching the scope and detail that we provide here.

RADclock The *RADclock* is feed-forward based. It does not rely on a feedback loop 'locking onto' the input signal, but instead post-processes server and local raw counter kernel timestamps to estimate the current clock error $E(t)$, which is removed when the clock is read. More concretely, the (absolute) *RADclock* is defined as $C_a(t) = N(t)\cdot\bar{p}+K-E(t)$, where $N(t)$ is the raw timestamp made at true time t, $\bar{p}$ is a stable estimate of average counter period, and K a constant which aligns the origin to the required timescale (such as UTC) (see [9], [11] for details, and [20] for a less formal description). The feed-forward design provides both an absolute clock and a *difference clock* (a clock uncorrected for drift used to measure time differences very accurately).

The *RADclock* (using the TSC as the counter on a single core with no power management) has been tested and independently validated on (collectively) years of live data [11], [21], [13], where in particular its robustness against disruptive events has been shown. In contrast, here we test it using several different counters, and compare its robustness directly to that of other approaches, which has only been done before to a limited extent for *ntpd* only [13].

The *RADclock* approach is fundamentally different because timing packets are timestamped using raw counter values which are independent of clock state, whereas *ntpd* uses the system clock to generate its input, creating a feedback loop.

B. Experimental Methodology

Figure 1 shows the key testbed components used in our comparison methodology. A Stratum-1 Time Server (right) is synchronized locally to a GPS signal by the *ntpd* daemon and serves time packets using both the NTP and IEEE-1588

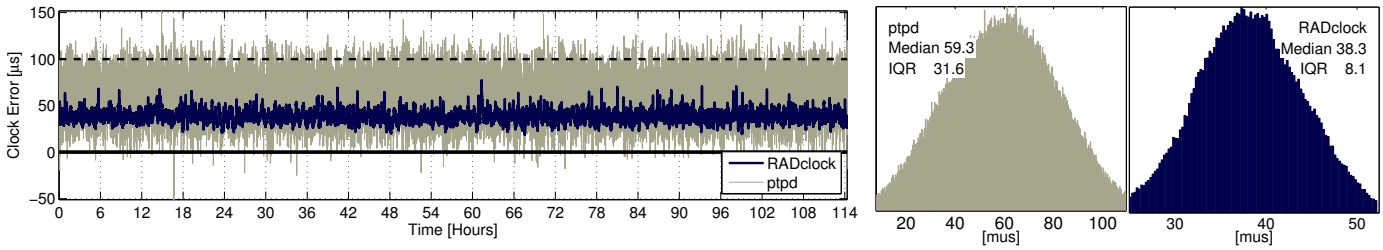

Fig. 2. Ideal environment: bettong by itself with minimal host and network load. Left: clock errors using external monitor, Right: histograms.

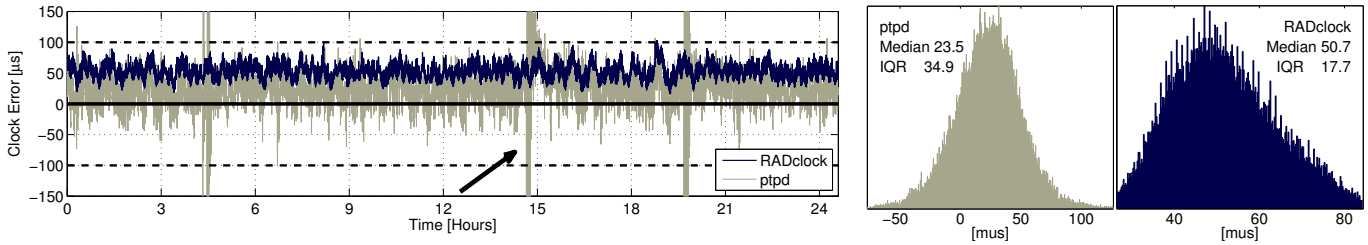

Fig. 3. Results for wallaby after it joins bettong on the hub. Spikes in host noise generate instability in *ptpd* (arrow marks event examined in Figure 4).

protocols. A PC host simultaneously runs several clocks to be evaluated/compared (left), including *ntpd* synchronized both over the network and via a local GPS signal. The external monitor consists of a GPS synchronized (with atomic clock PRS-10 corrected PPS) DAG capture card [22] using an 100 Mbps Ethernet hub as a tap.

The benchmarking methodology [21] relies on a flow of UDP packets sent and received by the PC host. Each UDP packet triggers an event which is timestamped by all clocks running in the host and the DAG card. The internal comparison of these timestamps gives a noise free view of their relative error, while the comparison of each clock's timestamps against the DAG provides an independent and absolute comparison. To provide a fair comparison, all clocks are configured with their default parameters and share a common constant polling period. For *ntpd* we found it necessary to restrict its configuration to a single server and disable adaptive polling rate, to avoid bad behavior.

We use the *rdtsc()* function, which accesses the TSC (using assembler instructions) from either kernel or user space, as a low latency low level timestamping operation for benchmarking purposes (the availability of this function is one of the advantages of using the TSC).

C. RADclock and ptpd over LAN

Our first experiment in a LAN environment uses two client hosts, wallaby (FreeBSD 5.3) and bettong (Linux 2.6.20), which run both the *RADclock* and *ptpd* daemons, each using the TSC as the underlying counter. Figure 2 shows the error over time (evaluated using the external monitor) of each client clock in bettong under ideal conditions: no other machine on the hub; negligible network traffic; very light host load. The time series show very consistent performance for each clock, however the *RADclock* is considerably less variable, with an Inter-Quartile Range (IQR) of 8.1 µs compared to 31.6 µs for *ptpd*. In this very undemanding, stationary LAN environment, the poorer performance of *ptpd* is a direct result of its inability to adequately filter the (nonetheless low) latency variability generated by both the network and the host.

To assess median performance we must compare it against the theoretical limit of $A/2$, where A is the path asymmetry between the client and server. We measure the network component of A as $A_n = 28\,\mu s$, yielding a net median error of $24.3\,\mu s$ for the *RADclock* and $45.3\,\mu s$ for *ptpd*. The asymmetry estimate can be further improved by including the host component A_h (see [11]) but for space reasons we omit this and focus exclusively on variability below. The findings for *ptpd* are worse than those from [5], namely median errors under 10 µs and an IQR of around 5-10 µs, but details of how these were obtained were not provided.

While keeping bettong running, we next add wallaby to the hub. Again the hosts are minimally loaded as is the network. The results in Figure 3 for wallaby are roughly similar to before although variability increases for each clock, since wallaby has a higher measured operating system noise [21]. It appears from Figure 3 that the *RADclock* has higher median error, however since asymmetry effects have not been removed here, no such conclusion can be drawn. As mentioned above we focus on variability in this paper.

Before leaving this benign environment we examine two

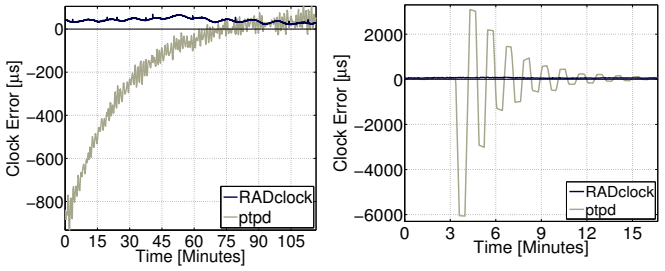

Fig. 4. Left: zoom on clock startup for bettong, Right: zoom on reaction to host noise event for wallaby (marked by arrow in Figure 3).

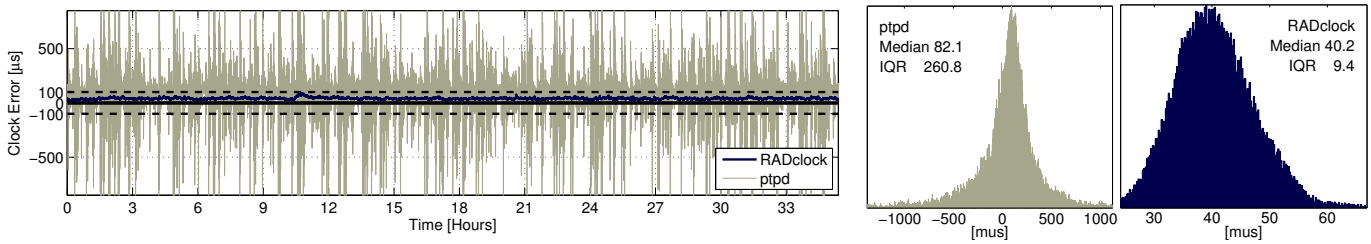

Fig. 5. Results for bettong with network congestion added (but still low host load). The *RADclock* is basically unaffected, *ptpd* is strongly affected.

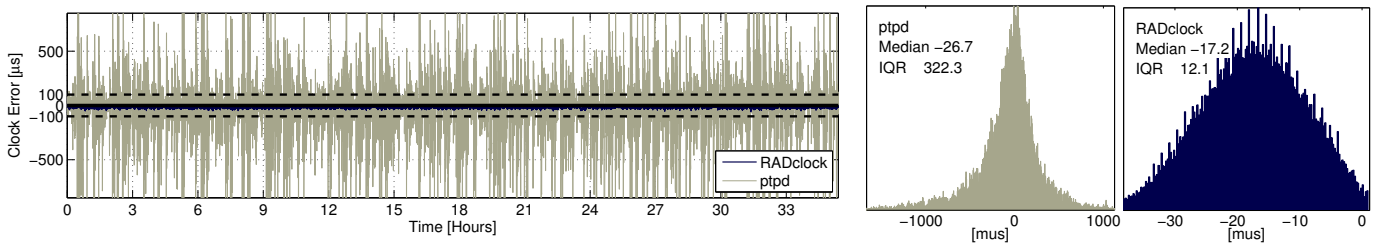

Fig. 6. Results for wallaby with both network congestion and high host load. Both clocks are affected, but *RADclock* much less so.

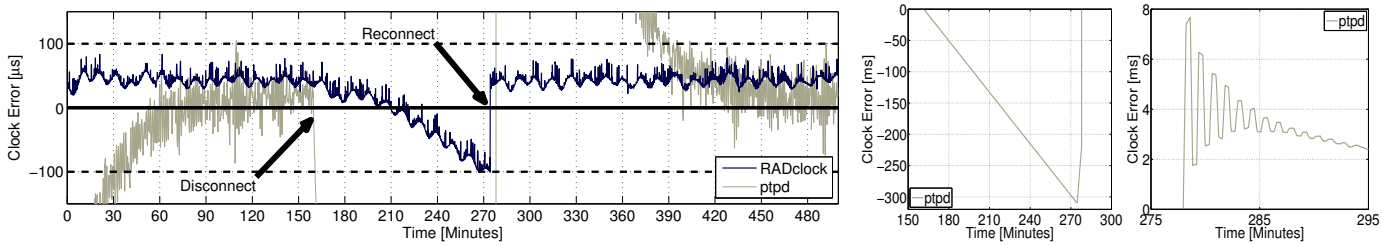

Fig. 7. Robustness test: disconnection from server. The *RADclock* drifts gracefully, *ptpd* does not. Right: zooms on *ptpd* disconnection and reconnection.

interesting points with the help of Figure 4. The left plot shows a zoom on the startup phase of both clocks for bettong. It takes an hour and a half for *ptpd* to converge whereas the *RADclock* achieves nominal performance almost immediately. The right plot is a zoom-out (vertically) and in (horizontally) on the spike pointed to by the arrow in Figure 3. The triggering event for this spike is caused by operating system effects such as interrupt scheduling and is typical of FreeBSD – no such spikes were seen on Linux. What is of interest here is how the clocks react to such an unusually 'large' system 'noise' input. In *ptpd* the event resulted in a very large jump in error and a slow, oscillating return to synchronization. In contrast, the *RADclock* is unaffected.

We next produce much more variable communication latency and increase network congestion with traffic exchanged between wallaby and a third host placed on the monitoring hub. We repeatedly transfer a 750MB file simultaneously to and from wallaby via the UNIX *scp* command, separated by 10 s pauses. Each transfer is capped to 15000 Kbit/s, an average load of around 30Mbit/s, over around 7 minutes.

Figure 5 shows the performance of the clocks on bettong under this scenario of high network congestion, but low host load. Comparing against Figure 2, we see that the *RADclock* is barely affected, but *ptpd* suffers significantly. Figure 6 shows the performance on wallaby during the same experiment. Note that wallaby experiences not only significant network congestion but also significant system noise because it is the origin and destination of the cross-traffic. We again find that the IQR for the *RADclock* is barely affected when comparing either against Figure 5, or against the same machine under very low load (in fact it is lower than that 17.7 µs found in Figure 3!), whereas the IQR performance of *ptpd* degrades by a further 61.5 µs from the already poor result of Figure 5. In fact the performance of the *RADclock* under heavy load is far better than the performance found for *ptpd* even under the ideal conditions of Figure 2.

Note that under these high load conditions, the noise polluting the external comparison also increases significantly, and in fact dominates the actual clock error in the case of the *RADclock*. This noise is more severe on the incoming side (*i.e.* received UDP packets). Thus, whereas in the other plots we showed errors as evaluated using the incoming direction, in Figure 6 we have instead given performance using the outgoing direction (even the outgoing direction suffers from increased noise however in this more difficult environment, which means that actual clock performance is better than the results quoted above). The change from incoming to outgoing brings with it a shift in median related both to asymmetry which we have not attempted to correct for here.

Finally, we examine the robustness with respect to a loss of connection to the time server. We return to the light load scenario, and first allow each clock to converge. We then

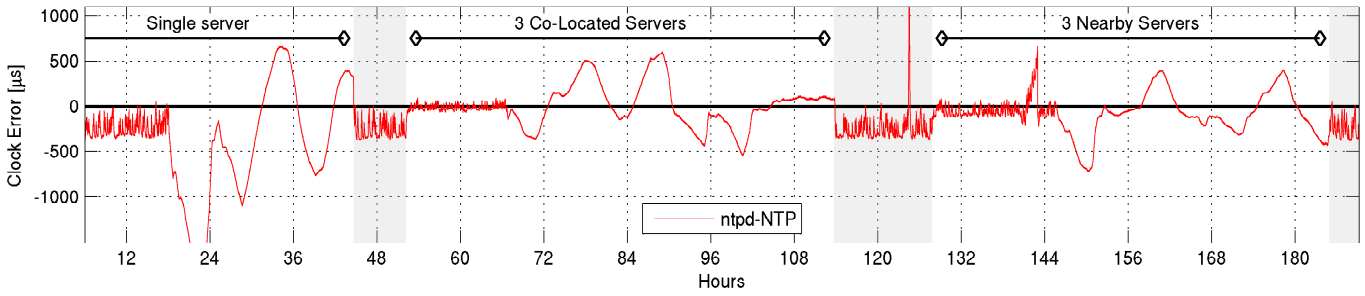

Fig. 8. Impact of multiple servers and polling period adaptivity on *ntpd*. The gray areas indicate forced configuration (single server and 16 s polling period).

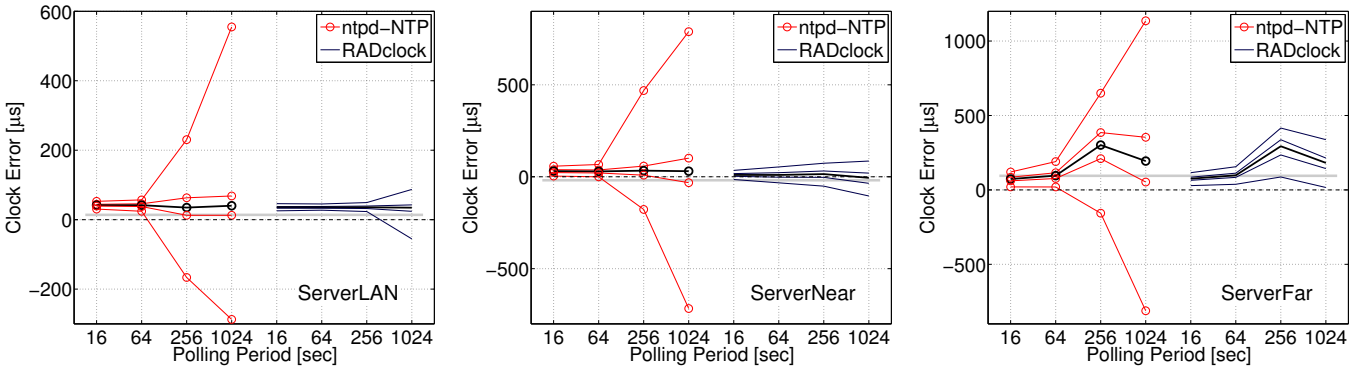

Fig. 9. Distribution of *RADclock* and *ntpd* error as a function of polling period and distance to time servers (LAN; same campus; across the continent). The *ntpd* and *RADclock* results are separated horizontally (note repeated horizontal axis) for clarity.

disconnect the monitoring hub from the network for about 2 hours, then reconnect it. Figure 7 shows the impact on wallaby. The *RADclock* shows a gradual drift indicative of the uncorrected local oscillator, and immediate recovery upon reconnection. In contrast the reaction of *ptpd* is extreme. As the middle plot in Figure 7 shows, following the disconnection at 150 minutes *ptpd* dives to reach an error of -300 ms before reconnection, after which its error remains in the ms range (rightmost plot) for most of the hour required for convergence.

D. RADclock and ntpd over LAN and WAN

We now compare the *RADclock* to *ntpd*, which is much more refined than a simple PI controller. It is well known however, that the complexity of *ntpd* makes its configuration, where accuracy is traded-off against robustness, a non-trivial task. We demonstrate this by observing the performance of *ntpd* running on a Linux host synchronizing to the Stratum-1 servers at our disposal.

The experiment shown in Figure 8 starts with *ntpd* synchronizing to one of our 3 ServerNear servers, which are co-located 5 hops away from the client with a minimum Round-Trip Time (RTT) measured at 0.89ms. We fix *ntpd*'s polling period to 16 s and let it converge. A noisy but stable performance is then exhibited in a [-350,-50] µs band. After 18 hours, we update *ntpd*'s configuration to allow *ntpd* to adaptively select its polling period. The result is erratic performance in a [-1.8,+0.6] ms band until we reimpose the constant 16 s value at hour 46. At hour 52 we add the two remaining ServerNear servers to *ntpd*'s configuration. These 3 servers are reached with the same route and *ntpd* synchronizes to the best of them exhibiting good performance in a [-50,+50] µs band. At hour 66 we relax the fixed poll period and *ntpd* shows again an erratic performance, only slightly better behaved thanks to the better ServerNear selected, despite the redundancy introduced by the additional servers. At hour 112 we reinstate the original configuration and again let *ntpd* converge. This phase is not free of errors as shown by the short but large spike error of 1.2 ms at hour 124. Finally, at hour 128 we add two new servers to the configuration. ServerLAN is a Stratum-1 on the same LAN with a minimum RTT of 0.28 ms. ServerLoc is located two hops away with minimum RTT 0.38 ms. Good overall performance is now seen, but again, the use of multiple nearby high quality servers does not prevent errors (hour 140). At hour 146 we again relax the poll period constraint and find erratic performance despite the richer network path diversity provided by the 3 servers. Because the network congestion is very low here, it is likely that the addition of cross-traffic will translate into even worse performance.

This experiment shows the sensitivity of *ntpd* to its configuration. Letting *ntpd* choose its own polling period leads for example to unstable behavior and performance degradation by two orders of magnitude.

We now compare *ntpd* and the *RADclock* when *ntpd* has been configured carefully to give its best possible performance (single nearby statically allocated Stratum-1 server, static and small polling period, avoidance of disruptive events). Figure 9 presents the clock error distributions of the *RADclock* and *ntpd* under FreeBSD as a function of the polling period, and three Stratum-1 NTP servers at different distances: ServerLAN,

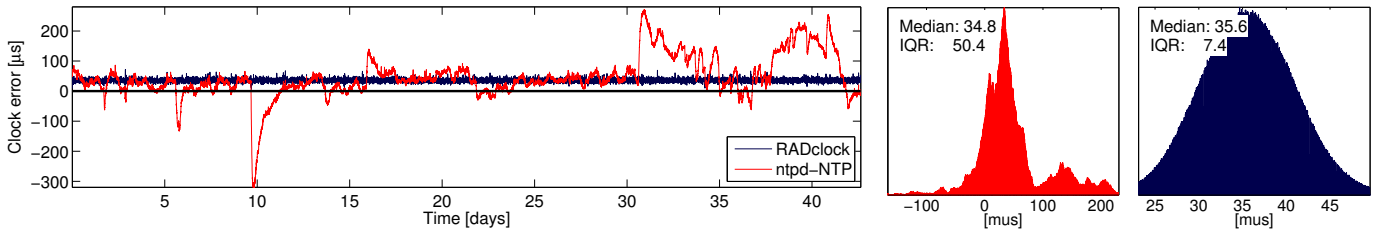

Fig. 10. Performance of *RADclock* and *ntpd* synchronizing to LAN Stratum-1 server and 256 s polling period; time series (left) and histograms (right).

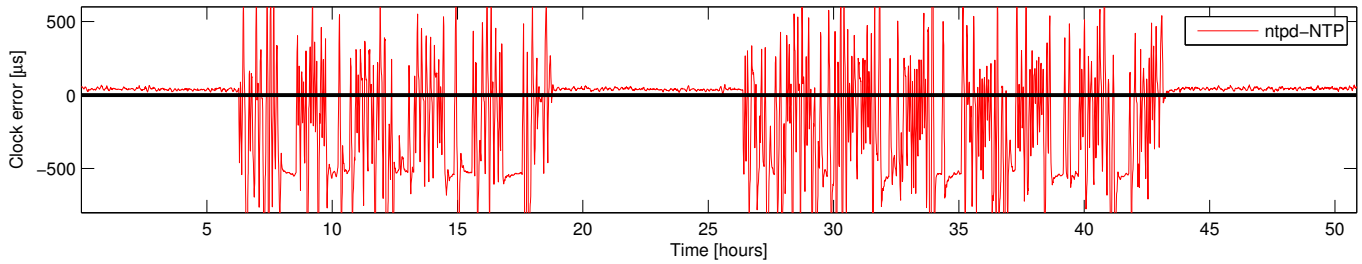

Fig. 11. Performance of *ntpd* synchronizing to Stratum-1 LAN server with disruption periods due to in-kernel random delays added to NTP packets.

the best ServerNear, and ServerFar (3500km away, 14 hops, minimum RTT 37.7 ms). In each plot, the thick black lines show the median clock errors, and the surrounding lines give the [1, 25, 75, 99] error percentiles. Each polling period corresponds to a data capture of at least 3 weeks where *ntpd* and the *RADclock* share the exact same flow of NTP packets.

As expected, the spread of errors increases both with increased distance to the server (greater network latency variability), and polling period (less raw data from the server). For each point in the (server, polling period) space, *RADclock* performs clearly better than *ntpd* both in terms of general variability shown by the IQR, and outliers. The clocks exhibit a similar median error (compare not to zero but to the horizontal gray line, which corrects for the total route asymmetry).

Results similar to Figure 9 were presented in [13]. The present figure is however entirely new, based on a more recent version of FreeBSD (6.1 instead of 5.3) and associated *RADclock* kernel timestamping, a different *ntpd* version and configuration, and uses different (and better) servers. Under these conditions each clock, in particular *ntpd*, performs better compared to the results in [13]. The new results also use a wider range of polling period (16 to 1024 s rather than 16 to 256), new and much longer data sets, and include fewer routing disruptions (further improving *ntpd*'s performance). This data, unique in the literature, took over 2 years to collect, and represents over 9 months of fully validated continuous operation of the two clocks running in parallel.

While *ntpd* is clearly more robust than *ptpd*, periods of stability loss occur for polling periods larger than 64 s even in the LAN environment where congestion is low. Figure 10 shows the entire time series and associated histograms corresponding to ServerLAN with a polling period of 256 s. This time series is characteristic of *ntpd*'s behavior observed in all data sets. For long periods, *ntpd* performs decently but exhibits phases of significant instability which sometimes last as long as a week! Again, no particular environmental triggers are present, and all points to instability in the *ntpd* feedback algorithm. Using the exact same NTP packet input, the *RADclock* exhibits an error IQR of 7.4 µs compared to *ntpd*'s 50.4 µs.

We conclude this section by actively disrupting *ntpd*. Here *ntpd* is running on a Linux client and synchronizes to ServerLAN with a fixed 16 s polling period. For the first 6 hours, *ntpd*'s clock error is small, as seen in Figure 11. From hour 6 to 17 and from hour 26 to 43, we run a low priority user space process, which uses the LibPCAP library to capture the flow of NTP packets used by *ntpd*. This process forces each packet captured in the kernel to be exported to user space without any buffering, resulting in small delays before *ntpd* can access the packets. Surprisingly, this makes *ntpd* extremely unstable. Its performance degrades by two orders of magnitude and exhibits patterns that remind one of Figure 6 for *ptpd*.

E. RADclock difference clock versus ptpd and ntpd.

Given any absolute clock $C_a(t)$, the error incurred by using it to measure a time difference: $C_a(t_2)-C_a(t_1)$, is just the sum of the clock errors at true times t_1 and t_2. The performance for time difference measurement for each of *RADclock*, *ptpd* and *ntpd* therefore follow from the results above. However, the *RADclock* also provides a *difference clock*, which corresponds to using a clock uncorrected for drift, and therefore unpolluted by the inherent errors of drift estimation. As detailed in [21], this results in far more accurate, and far more robust, measurement of time differences below a critical time scale, than if an absolute clock were used. For example, when measuring the time taken to execute a code block, the number of CPU cycles expended is often used. The difference *RADclock* using the TSC counter corresponds to this same idea, except it also converts the cycle count to seconds using an extremely robust and accurate (say to 0.1 µs over a 1 s interval) estimate of long term counter period. Due to their feedback natures, *ptpd* and *ntpd* cannot provide a difference clock, but can only measure time differences by subtracting absolute timestamps.

III. Kernel support for feed-forward algorithms

Although the above section compared three specific clocks, the key conclusions are based on fundamentals. Feedback controllers suffer the fundamental tradeoff whereby parameter values which improve short term tracking degrade rate stability over the time-scales at which tracking operates, and in more extreme cases threaten global stability. The latter is paramount, resulting in parameter setting with slow initial convergence, and controller-induced errors in clock rate, but still with no guarantee of stability if conditions worsen. In contrast, a feed-forward approach **cannot** be unstable, and the absence of the above tradeoff means that high accuracy can coexist with fast convergence, both initially and following periods of high noise.

While the results of Section II were obtained using the TSC, the *RADclock* algorithm itself will work with any counter of constant nominal frequency that does not roll over. Currently, PC architectures typically embed the Time Stamp Counter (TSC), but also the Advanced Configuration and Power Interface (ACPI) timer, and often the High Precision Event Timer (HPET). In a number of architectures power management makes use of the TSC problematic (here we disable it). The ACPI and HPET are immune to this problem (refer to [15], [23] for more details).

Because any of the above counters can be used for timekeeping, the operating system selects the one it considers the most reliable and provides a generic interface to access it. This interface is called *timecounter* [24] on FreeBSD and *clocksource* on Linux, and is internal to the kernel. While the *rdtsc()* function allows the interface to be bypassed in the case of the TSC, the other counters can only be accessed via this interface. Unfortunately, it is not suitable for feed-forward synchronization algorithms. To understand why, we first describe briefly the principle underlying these interfaces.

All counters are initialized to 0 at system boot and incremented at the period of their respective oscillators. With the exception of the TSC, the available counters roll over several times per minute (see table I). The kernel mechanism that tracks roll-over events, thereby maintaining a consistent notion of time, works as follows. On every system, the "*hardware clock*"[1] generates interrupts that are captured by the kernel (typically every $1\,\mathrm{ms}$). On every interrupt, the kernel creates two timestamps. One timestamp is the reading of the current hardware counter value (*counter record*) and the other is derived from the system clock (*time record*).

When a program issues a *gettimeofday()* system call, or when an interrupt is raised by the *hardware clock*, the kernel needs to create a system clock timestamp. To this end, the kernel reads the current counter value and computes δ, the number of cycles elapsed between the last *counter record* and the current value. The kernel converts δ into seconds, adds it to the last *time record* and returns the result. If the timestamp creation was triggered by *gettimeofday()*, the timestamp is passed back to user space. If it was triggered by a *hardware clock* interrupt, the timestamp becomes the new *time record* and the current counter reading the new *counter record*.

[1]The hardware clock is usually based on the legacy 8254 Programmable Interval Timer (PIT), the Real-Time Clock (RTC) or the HPET counter itself.

	TSC	HPET	ACPI
Frequency	CPU freq.	14.3 MHz	3.57 MHz
Size (bits)	64	32 / 64	24 / 32

TABLE I

Because a monotonically increasing *time record* is associated to every *counter record*, this mechanism implicitly tracks the counter's roll-over events. It is also robust to *hardware clock* interrupts being missed, since their frequency is far higher than that of any counter roll-over. It is an intrinsically feedback mechanism however, since the conversion of δ into seconds is performed by the kernel's system clock, and is driven by the information passed by the synchronization algorithm to the kernel (via the *adjtime()* system call).

The kernel's notion of time, and any synchronization algorithm making use of the interface, are locked together. The algorithm never sees a raw counter value, but only values which have already been coupled to both its own state and the system clock timestamping mechanism. It follows that a feed-forward algorithm, being based on direct access to the raw oscillator (via a counter), cannot use this interface. Based on our previous experience with the TSC counter, we know that the ideal counter has to be wide enough not to roll over, have high stability and be accessible quickly and atomically.

Our implementation synthesizes such a counter. It consists of a 64 bit *cumulative counter record* added to the *timecounter* and *clocksource* interfaces. This 64 bit field is used to record a snapshot of a cumulative count of the active counter and because of its size, will not roll over. To allow access from user space we implemented a new *getcounter()* system call.

When a program issues a *getcounter()* system call, or when an interrupt is raised by the *hardware clock*, the kernel has to determine the current cumulative counter value. The kernel reads the current value of the counter and computes δ as in the feedback case. The current cumulative counter value is then created as the sum of δ and the last *cumulative counter record*. In the case of triggering by *getcounter()*, the current cumulative count is passed back to user space. If it was triggered by a *hardware clock* interrupt, it is stored as the new *cumulative counter record*.

This mechanism implements the simple yet crucial requirements for kernel support for feed-forward applications. It decouples the timestamping and timekeeping mechanisms in the kernel while taking advantage of the existing implementation. It is also generic enough to give access to future hardware counters as soon as they are supported by the interfaces [25]. Finally, it gives access to consistent raw counter timestamps via a new data structure available both from within the kernel and to user space applications.

Note that allowing raw counters to be accessed has another advantage, it allows the timestamping of timing packets to be low level, and yet synchronization intelligence to be located in a single place: user space, rather than being split between user space and the kernel.

IV. Counter Characteristics

Whereas all available counters are now fully accessible to any synchronization algorithm, their characteristics may

differ. In this section we examine how a number of commonly available counters compare as hardware timing sources.

We use recent computers that embed the TSC[2], HPET and ACPI counters. We focus mainly on a FreeBSD 7.0 system that provides two access methods to the ACPI counter [26]. The *fast* method simply reads the counter as quickly as possible. The *safe* method is intended for ACPI architectures that may not correctly latch the counter, and compensates by reading it several times until a correct value is read, making it slower.

The characteristics we focus on are the key ones for feed-forward synchronization algorithms, but the findings should be useful for many applications requiring raw timestamps. As far as we are aware this is the first study which systematically characterizes, and compares, these counters.

A. Stability

In our testbed the FreeBSD kernel captures the PRS-10 PPS signal through a serial port using the standard PPS API [27]. On each pulse, a raw 64 bit cumulative counter timestamp is created within the kernel and exported using the PPS API. In post-processing, the inter-pulse times measured in counter units are calculated, and are then expressed in seconds using the average pulse period computed over the trace using GPS receiver timestamps. The resulting inter-pulse time series is then analyzed using the Allan deviation plot to measure the counter stability on different timescales. Allan deviation is a time-scale dependent measure of variability that is commonly used to analyze clock errors [11].

The above methodology enables us to observe counter stability when accessed via the *timecounter* interface. Figure 12 shows the Allan deviation of each counter measured over consecutive 1 week periods on a desktop computer (tastiger) and a rack server (platypus) both located in the same temperature controlled server room. In addition to the 4 counters which we access via the *timecounter* interface, we also show the TSC counter when accessed via the *rdtsc()* function both as a point of reference, and to compare to our previous work [11].

Figure 12 shows that all the counters we observe exhibit a stability below 1PPM (Parts Per Million) at any timescale above 10 s. It also clearly indicates that on both test machines, all counters have very similar characteristics. At small time scales, the variability of the system noise due to the serial port dominates but quickly falls below 0.1PPM at large scales. The counters exhibit a familiar minimum a little past 1000 seconds, before flattening out to a low level at daily time scales and beyond, consistent with a tightly controlled temperature environment. The only differences between the counters are weak and appear at large time scales, due to the weekly temperature variations unavoidable in consecutive captures.

All counters on both test machines exhibit a "bump" in the Allan deviation at around 200 s, which is related to the period of the server room air conditioning system. This is of interest for production environments that are equipped with similar systems. Although the impact is noticeable it remains well below 0.1PPM. A similar bump was noted in [9].

[2]All functions affecting the stability of the TSC have been disabled for the purpose of comparison and consistency.

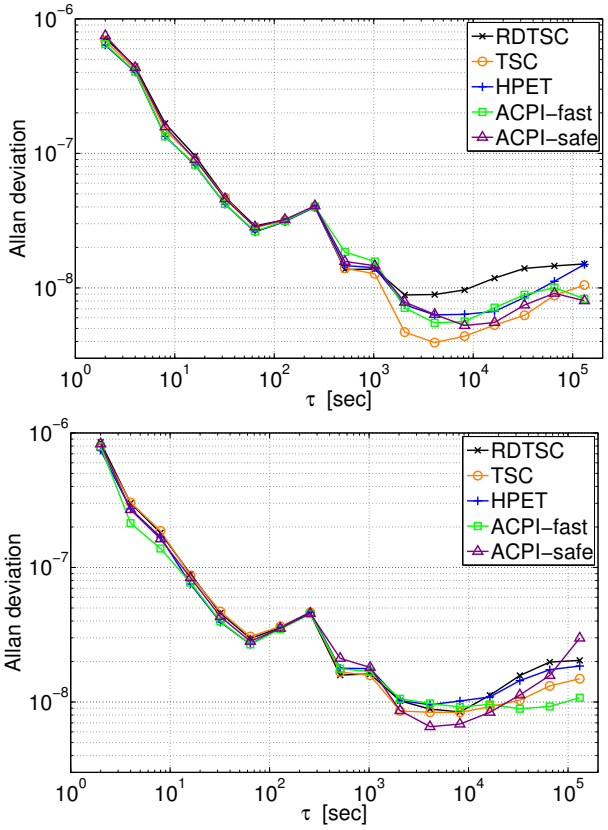

Fig. 12. Stability of 4 counters based on PPS based polling in a: Top – desktop computer (tastiger); Bottom – rack server (platypus).

In conclusion, the counters exhibit virtually identical stability characteristics, which makes them equivalently useful for the purpose of timekeeping from this point of view.

B. Stability Under Stress

We examine the response of each counter to a predefined scenario. Starting in a stable environment, the computer undergoes alternating 90 minute long periods of stressed and normal conditions. In the first two stress periods, a user space infinite loop continuously maintains the CPU activity over 95%. During the last stress period, the computer network card is set in promiscuous mode and a tcpdump process captures all packets transmitted over the network. By generating heavy cross traffic (the 100Mbit/s hub the computer is connected to is loaded at maximum capacity), the network card generates many interrupts on the system as packets are captured.

Using the PPS signal capture in the kernel, we compute the number of cycles elapsed between consecutive pulses, giving us a direct estimate of the "instantaneous" frequency for each counter. Using the "normal conditions" period of the test, we also compute an average reference frequency for each counter. Figure 13 shows, for each counter, the error of the instantaneous frequency relative to its reference frequency expressed in PPM. The captures are each 11 hours long and are taken sequentially on the rack server machine **possum**.

Our expectation was that the CPU load stress would affect the TSC counter more than the others since it is located on the

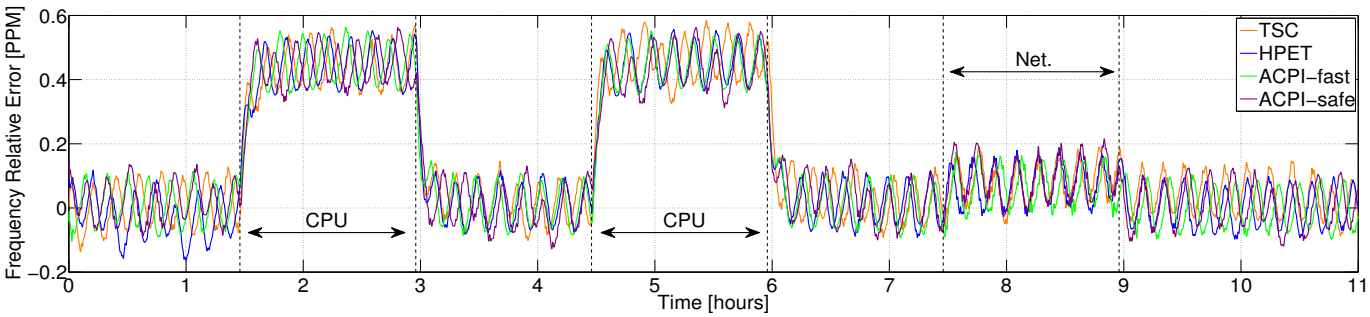

Fig. 13. Instantaneous relative frequency error of each counter under CPU load and network interrupt stress scenario on the rack server *possum*.

CPU chip. Furthermore, it was supposed that the generation of many interrupts would create unequal contention for the counters' access methods due to their differing hardware design. Figure 13 indicates that our preconceptions were wrong: the frequencies of all counters change in a comparable manner when under stress. Moreover, the results point to temperature as a main factor for frequency change. Under CPU load, the temperature of the computer increases notably, while the creation of a multitude of network interrupts induces a much milder temperature change (the high frequency oscillations are due to the air conditioning system). The most probable explanation for the observations is that the crystal and/or the clock synthesizer chip on the motherboard is affected by the temperature changes and that this is reflected by all the counters that derive from it.

We have seen that all counters are equivalently stable even under high stress, and that even under heavy load the frequency variation remains below 1PPM, a significant result for server machines which are likely to encounter such situations. Finally, we see that even very high network activity has little impact, an important feature for network intensive applications such as network monitoring and anomaly detection that are dependent on packet capture and timestamping.

C. Access Latency Under Stress

Each counter is read via a different mechanism. The TSC value can be returned extremely fast and accessed in few assembly instructions[3]. The HPET and ACPI counters are accessed via reading on a data bus, which is slower. The HPET counter however, is memory mapped and should therefore be faster than ACPI.

Here, we study the access latency of each counter (including the TSC) accessed via our extended interface, measured in CPU cycles from within the kernel. Specifically, each time a given counter is to be read, we first read the TSC using *rdtsc()*, access the counter under study, then use *rdtsc()* again. These two TSC values are exported to user space and subtracted from each other to measure the counter access latency, comprising the sum of the latencies of the *timecounter* or *clocksource* interface, and that of the counter itself. As we discuss in the next section, the former is approximately constant.

Figure 14 shows the latency of the counters expressed in CPU cycles, over the same 11 hour stress scenario as described in Section IV-B. The distributions of the latency for each phase of the stress scenario are presented in a compact format where whiskers show the minimum and 95th percentile values. The box lower and upper sides show the 25th and 75th percentiles values, while the internal horizontal line marks the median.

We first look at the normal periods when the system is not under stress. As expected, the TSC is the fastest counter with a median access latency of 420 CPU cycles (less than 140 ns on a 3 GHz processor). The HPET counter comes second with a median value of 1935 cycles (0.65 µs), and the ACPI is last at about 3690 cycles in its fast mode (1.23 µs), and 10440 cycles in its safe mode (3.48 µs). These values are far from negligible since they reach the micro-second level even for a fast modern computer. Thus, whereas the counters were equivalent from a stability perspective, their differences in access latency are appreciable and can impact the quality of timestamping.

Although the size of the counters' access latencies can have an impact, their variability is typically quite small. The Inter-Quartile Range (IQR) for the TSC is essentially null because of the high level of discretization of the TSC reading that is architecture dependent. HPET exhibits an IQR of 135 cycles, ACPI-fast 285 cycles and ACPI-safe 390 cycles. Such small values are encouraging as they translate to a lighter burden on the synchronization algorithms' delay variability filtering, and may even have no effect at all if in addition the access latency has symmetric calling and returning components.

The periods when the computer is under stress show an interesting pattern. As shown by their distributions in Figure 14 the counter latencies take lower values during the CPU test. The median values all drop by about 100 cycles, a counter-intuitive result: performance improvement under stress! The explanation is not that the counters are accessed faster, but that our measurement methodology has one flaw. While robust to frequency changes, reading the number of CPU cycles is prone to pipelining and caching optimizations. Under CPU load, the caches are "hot" and the apparent performance improvement is an artifact of this. For the same reason, the slightly lower number of instructions to measure the latency of the counters reduces the probability of the corresponding execution being scheduled out, which then reduces the number of outliers.

In the case of the network stress periods, the results obtained are extremely close to the normal ones, even to the values of the 95th percentiles. The only noticeable difference concerns the extreme outliers, which do not appear in the plots, but which take much higher values. In other words, in the presence

[3]2 instructions on 32 bit CPU systems and 1 instruction on 64 bit systems.

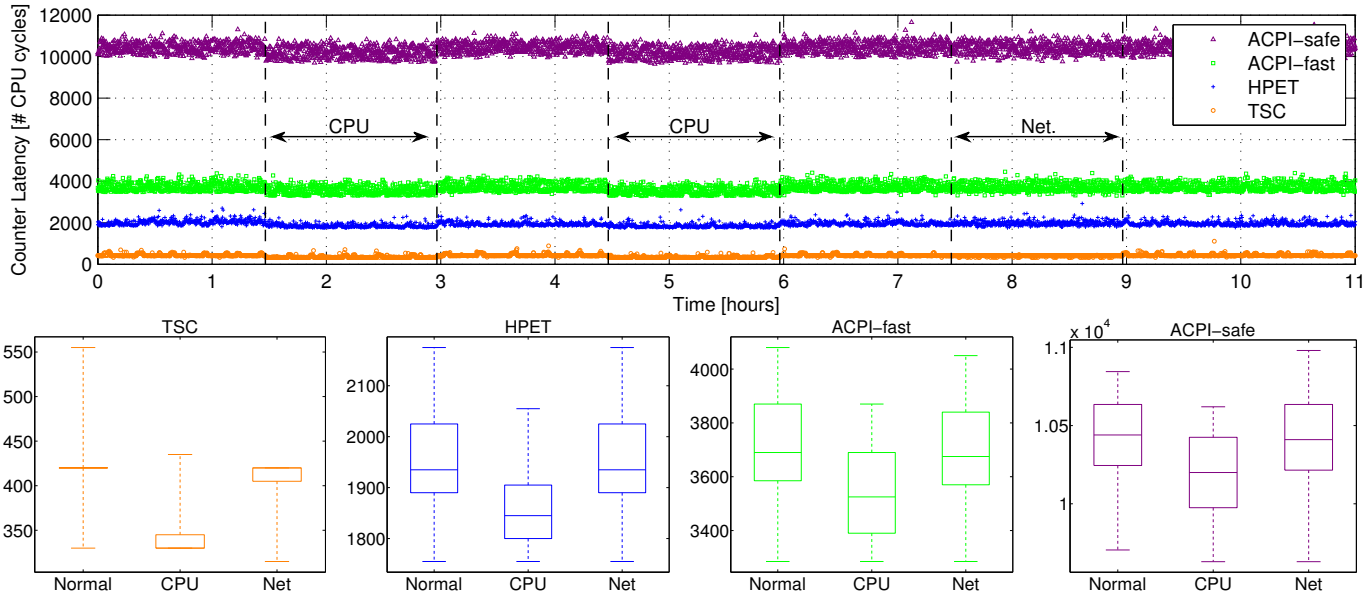

Fig. 14. Access latency for the 4 counters available (FreeBSD), under the stress scenario. Time series (top) and distributions from 0 to 95th percentile.

of network interrupts the counter access code is not interrupted more often, but is it interrupted for longer. For example, the maximum latency for HPET in the normal case is 38265 cycles but 11874705 cycles under network load. The TSC counter is an exception however. Because the network stress is based on tcpdump, all incoming packets are timestamped, leading to another "hot cache" optimization as the TSC counter is continuously accessed, resulting in almost no outliers.

In summary, the reaction to the stress scenario is mild, and almost identical, for each counter.

V. Raw Timestamping and Synchronization

In this section we examine the impact of counter choice on *RADclock* performance. Before doing so, we complete the picture on the impact of the enhanced counter interface itself by measuring its access latency from both kernel space and user space. This is important for any clock application, since clocks must be readable by any process.

A. Timecounter/Clocksource Timestamping Latency

First from a kernel perspective, we are interested in quantifying the overhead produced by the use of the generic counter interface, independently of the counter being accessed. For this purpose we implemented a kernel module that performs two calls to the *rdtsc()* function, then reads the TSC counter via the interface, and then calls the *rdtsc()* again. This provides us with the latency of the *rdtsc()* function itself and the latency of the interface when reading the TSC counter. By subtracting one from the other, we obtain the latency of the interface alone.

Figure 15 shows that this latency has an extremely compact distribution on FreeBSD and Linux. The spread is only 240 cycles (80 ns on a 3 GHz processor) on FreeBSD and 152 cycles (50 ns) on Linux. For the purpose of software timestamping and synchronization, the kernel interface latency can then be approximated by its median value, namely 240 cycles on FreeBSD and 208 on Linux.

In summary, using the new interface for kernel timestamping adds an extremely small penalty compared to that of the *rdtsc()* function. Compared to the advantages it provides, it is therefore a very attractive tool for this purpose.

B. User Space Timestamping Latency

A program running in user space that needs to timestamp normally issues a *gettimeofday()* system call to access the system clock. In the case of a feed-forward clock using the TSC, the same program only need use the *rdtsc()* function to create a raw TSC timestamp, which the clock can convert to an absolute time in seconds asynchronously. The assembly code composing the *rdtsc()* function bypasses any usual kernel/user space channel and provides comparable performance whether from the kernel or user space.

The ACPI, HPET or other counters do not offer this option, and the use of the *timecounter* or *clocksource* interface forces a user program to issue our new *getcounter()* system call to retrieve a raw timestamp from the kernel. We now measure the latency of this call, using a technique similar to before.

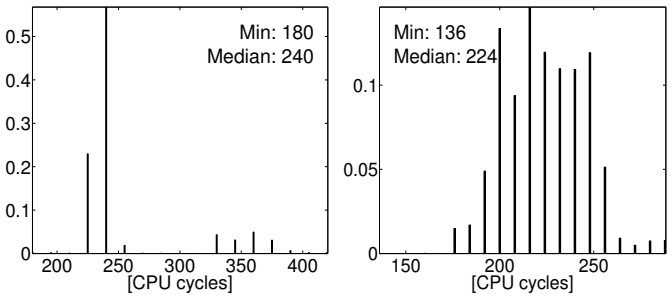

Fig. 15. Kernel latency distribution from minimum to 99th percentile: *timecounter* interface (FreeBSD, left) and *clocksource* interface (Linux, right).

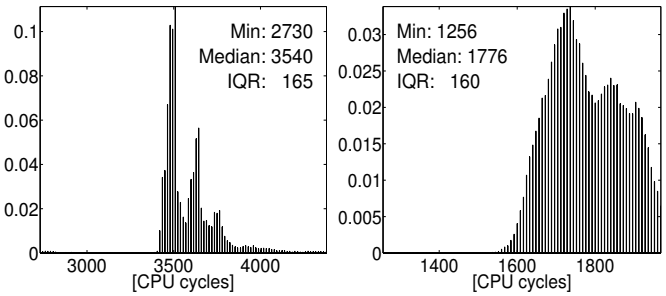

Fig. 16. Histograms of user space *getcounter()* system call latency for FreeBSD on **possum** (left) and Linux on **kultarr** (right).

Figure 16 shows the distribution of the system call latency on FreeBSD and Linux, and this time, the penalty induced by our solution becomes apparent. Whereas creating a raw timestamp based on *rdtsc()* cost about 150 cycles from user space, it now costs over 3500 cycles (almost 1.2 µs) on FreeBSD and over 1770 cycles (0.5 µs) on Linux. Furthermore, on both systems the variance of the latency distribution can no longer be ignored. Even if the clock were perfect, the timestamps would often exhibit errors close or above 1 µs on modern systems. This is even more important for synchronization algorithms that themselves rely on such timestamps. This result advocates the use of timestamps taken from within the kernel for the time keeping application itself (*RADclock* does this), which also reduces host latency variability.

C. Synchronization Algorithm Performance

We now examine the impact of counter choice on the *RADclock*. Figure 17 shows the distribution of *RADclock* errors against the DAG card reference on FreeBSD. The performance of the *RADclock* using the TSC via the *rdtsc()* function is shown, as well as each counter through the *timecounter* interface. Each distribution corresponds to one entire week of captured data where the *RADclock* synchronizes to a Stratum-1 server on the LAN using the NTP protocol. Note that the clock errors have each been corrected by a common estimate of the (in general unavoidable) network and host asymmetry bias as described in [21], in order to focus on the error variability.

As expected from the results above on counter stability, and the benefits of kernel based timestamping, *RADclock* performs similarly irrespective of the counter used. The IQR of all clock errors are all at about 7 µs, a value dominated by the impact of the air conditioning on the counters, as also observed on our Stratum-1 NTP servers fed with an atomic clock PPS signal.

The median error values vary in a band only a couple of micro-seconds wide. It is tempting to explain these variations through the differences between counter latencies which are of the same order of magnitude, however the slightly different network noise characteristics of each capture, or the counters' stability at weekly timescales, are also possible causes.

Additionally, we observed the impact of the stress scenario. While *RADclock*'s performance is slightly affected by the extreme temperature change and network activity, the performance is similar for all counters used. The median clock errors of all counters are at most 1.8 µs apart in the case of CPU load

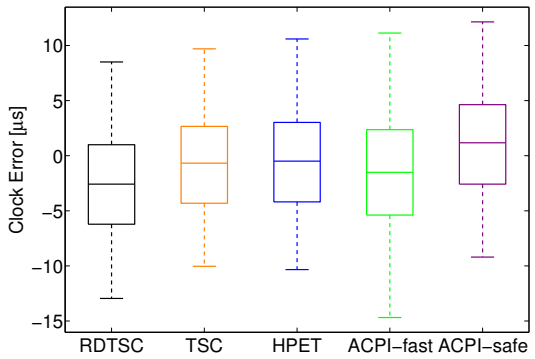

Fig. 17. *RADclock* final clock error for tastiger after asymmetry compensation for each counter. Distribution summary is from 1st to 99th percentile.

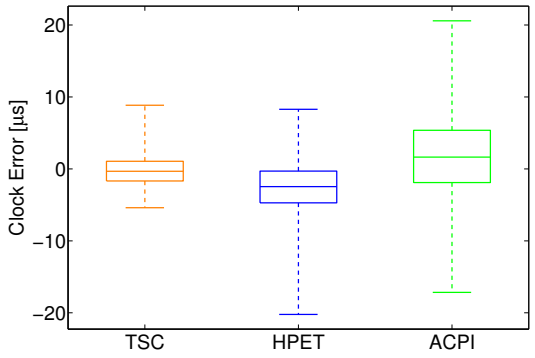

Fig. 18. *RADclock* final clock error after asymmetry compensation on Linux. Distribution summary (1st to 99th percentile) of TSC in server room, HPET on desktop computer, and ACPI on laptop in office environment.

and 1.1 µs for network load. This result reinforces the previous analysis and confirms that any counter under study is a valid candidate for the purpose of timekeeping.

Finally, Figure 18 shows the performance of the *RADclock* on a LAN but running on Linux in different environments. The TSC counter is used on a uni-processor computer without power management in the air conditioned server room. The HPET and ACPI counters are used respectively by a multi-core desktop computer, and a laptop with active power management and CPU frequency stepping, both in an office environment. The performance is extremely similar: even the laptop's IQR is below 8 µs, demonstrating the robustness of the approach across a wide spectrum of configurations.

VI. Conclusion

An extensive set of experimental results were presented demonstrating the sensitivity of two important synchronization approaches: one based on the IEEE-1588 protocol (*ptpd*) and the other (*ntpd*) the solution currently used by billions of networked computers, to latency variability typical of network based synchronization. In parallel, we showed the insensitivity to these same variabilities of our alternative solution, the *RADclock*. We argued that in large part it is the inherent advantages of a feed-forward paradigm adopted by the *RADclock*, rather than the feedback approach of *ptpd* and *ntpd*, which is responsible for this striking difference. Noting that existing kernel mechanisms are built around *ntpd* and do not satisfy

the requirements of feed-forward approaches, we identified changes which allow the raw counter value for any available hardware counter to be accessed for the first time, and made minimal extensions to existing kernel mechanisms on FreeBSD and Linux to achieve this. We immediately profited from this access to provide the first detailed characterization of the common counters TSC, HPET and ACPI as hardware timing sources on each of these operating systems, under both nominal and stressed conditions. We found that they have identical stability properties, and a small access latency from kernel space, but that there are small but non-negligible differences in access latency from user space. We demonstrated that the *RADclock* can use any of these counters with no measurable difference in performance, which is important since the TSC, although enjoying several advantages, cannot always be used on systems with features such as power management.

ACKNOWLEDGMENTS

We acknowledge Cameron Chambers' work on the implementation of the proposed kernel modifications. This research was supported under Australian Research Council's Discovery Projects funding scheme (project number DP0985673). This project has been made possible in part by a grant from the Cisco University Research Program Fund at Silicon Valley Community Foundation and by a Google Research Award.

REFERENCES

[1] D. L. Mills, "Network Time Protocol (Version 3) specification, implementation and analysis," IEFT, Network Working Group, RFC-1305, March 1992, 113 pages.

[2] ——, "The Network Computer as Precision Timekeeper," in *Proc. Precision Time and Time Interval (PTTI) Applications and Planning Meeting*, Reston VA, December 1996, pp. 96–108.

[3] J. C. Eidson, *Measurement, Control and Communication Using IEEE 1588*. Springer, April 2006.

[4] "The Precision Time protocol (PTP), ptpd," http://ptpd.sourceforge.net/.

[5] K. Correll, N. Barendt, and M. Branicky, "Design Considerations for Software Only Implementations of the IEEE 1588 Precision Time Protocol," in *Proc. ISPCS*. Zurich, Switzerland, October 10-12 2005.

[6] D. L. Mills, "A Kernel Model for Precision Timekeeping," IEFT, Network Working Group, RFC-1589, 1994.

[7] A. Pásztor and D. Veitch, "A Precision Infrastructure for Active Probing," in *Passive and Active Measurement Workshop (PAM2001)*, Amsterdam, The Netherlands, April 23-24 2001, pp. 33–44.

[8] ——, "PC based precision timing without GPS," in *Proc. ACM Sigmetrics Conf. Measurement and Modeling of Computer Systems*, Del Rey, California, June 15-19 2002, pp. 1–10.

[9] D. Veitch, S. Babu, and A. Pásztor, "Robust Synchronization of Software Clocks Across the Internet," in *Proc. ACM SIGCOMM Internet Measurement Conf.*, Taormina, Italy, Oct. 2004, pp. 219–232.

[10] E. Corell, P. Saxholm, and D. Veitch, "A User Friendly TSC Clock," in *Passive and Active Measurement Conference (PAM2006)*, Adelaide Australia, Mar. 30-31 2006, pp. 141–150.

[11] D. Veitch, J. Ridoux, and S. B. Korada, "Robust Synchronization of Absolute and Difference Clocks over Networks," *IEEE/ACM Transactions on Networking*, vol. 17, no. 2, pp. 417–430, April 2009.

[12] J. Ridoux and D. Veitch, "Ten Microseconds Over LAN, for Free," in *Proc. ISPCS'07*, Vienna, Austria, Oct.1-3 2007, pp. 105–109.

[13] ——, "Ten Microseconds Over LAN, for Free (Extended)," *IEEE Trans. Instrumentation and Measurement (TIM)*, vol. 58, no. 6, pp. 1841–1848, June 2009.

[14] ——, "The Cost of Variability," in *Proc. ISPCS'08*, Ann Arbor, Michigan, USA, Sep. 24-26 2008, pp. 29–32.

[15] T. Broomhead, J. Ridoux, and D. Veitch, "Counter Availability and Characteristics for Feed-forward Based Synchronization," in *Proc. ISPCS'09*. Brescia, Italy, Oct. 12-16 2009, pp. 29–34.

[16] N. Chongning, D. Obradovic, R. Scheiterer, G. Steindl, and F.-J. Goetz, "Synchronization Performance of the Precision Time Protocol," in *Proc. ISPCS 2007*, Vienna, Austria, Oct. 1-3 2007.

[17] G. Giorgi and C. Narduzzi, "Modeling and Simulation Analysis of PTP Clock Servo," in *Proc. ISPCS 2007*, Vienna, Austria, Oct. 1-3 2007.

[18] D. L. Mills, *Computer Network Time Synchronization: The Network Time Protocol*. Boca Raton, FL, USA: CRC Press, Inc., 2006.

[19] D. L. Mills and P.-H. Kamp, "The Nanokernel," in *32nd Annual Precision Time and Time Interval (PTTI) Meeting*, Reston VA, November 2000, pp. 423–430.

[20] J. Ridoux and D. Veitch, "Principles of Robust Timing Over the Internet," *ACM Queue, Communications of the ACM*, vol. 53, no. 5, pp. 54–61, May 2010.

[21] ——, "A Methodology for Clock Benchmarking," in *Tridentcom*. Orlando, FL, USA: IEEE Comp. Soc., May 21-23 2007.

[22] "Endace Measurement Systems," http://www.endace.com/.

[23] T. Broomhead, L. Cremean, J. Ridoux, and D. Veitch, "Virtualize Everything But Time," in *Proc. OSDI 2010*, Vancouver, Canada, Oct. 4-6 2010.

[24] P. H. Kamp, "Timecounters: Efficient and precise timekeeping in SMP kernels," in *Proceedings of the BSDCon Europe 2002*, Amsterdam, The Netherlands, 15-17 November 2002.

[25] P. Ohly, D. N. Lombard, and K. B. Stanton, "Hardware Assisted Precision Time Protocol. Design and case study." in *Proceedings of LCI International Conference on High-Performance Clustered Computing*. Urbana, IL, USA: Linux Cluster Institute, April 2008, pp. 121–131.

[26] T. Watanabe, "ACPI Implementation on FreeBSD," in *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2002, pp. 121–131.

[27] J. Mogul, D. Mills, J. Brittenson, J. Stone, and U. Windl, "Pulse-Per-Second API for UNIX-like Operating Systems, Version 1.0," IETF, Tech. Rep., 2000.

Julien Ridoux (S'01-M'06) received the M.Eng. degree in Computer Science (2001) and M.Sc. degree in Telecommunication and Networks (2002) respectively from the Ecole Polytechnique de l'Université de Nantes (France) and University Paris 6 (France). In 2005 he received the Ph.D. degree in Computer Science from the University Paris 6. Since 2006 he is a Research Fellow at The University of Melbourne where his main research interests are distributed clock synchronization and Internet traffic modeling.

Darryl Veitch (SM'02-F'10) completed a BSc. Hons. at Monash University, Australia (1985) and a mathematics Ph.D. from Cambridge, England (1990). He worked at TRL (Telstra, Melbourne), CNET (France Telecom, Paris), KTH (Stockholm), INRIA (Sophia Antipolis, France), Bellcore (New Jersey), RMIT (Melbourne) and EMUlab and CUBIN at The University of Melbourne, where he is a Professorial Research Fellow. His research interests include traffic modelling, parameter estimation, traffic sampling, and clock synchronization.

Timothy Broomhead began his degrees in mechatronic engineering and computer science at the University of Melbourne (2007). Since 2008 he has worked as an intern at the University of Melbourne over summer breaks at the CUBIN laboratory, working on the *RADclock* project. He also has an in depth interest in electronics and amateur radio, becoming Australia's youngest licensed operator (2001). He is also an IT consultant since July 05 and led several Web developments projects.