

# Synchronizing the Linux System Time to a PTP Hardware Clock

Richard Cochran, Cristian Marinescu and Christian Riesch

OMICRON electronics GmbH

Oberes Ried 1, 6833 Klaus, Austria

[richard.cochran|cristian.marinescu|christian.riesch]@omicron.at

**Abstract**—As computer and embedded systems are becoming more complex and distributed, keeping accurate time throughout the whole system becomes a challenging task. The IEEE 1588 Precision Time Protocol was designed to achieve very accurate synchronization in distributed environments. Linux is becoming the leading operating system for embedded devices, but little attention has been paid to the issue of how to internally synchronize the Linux system clock with the PTP hardware clock. Our paper presents the current status in this area, highlights possible solutions for this problem, and describes our efforts to address this key issue.

**Index Terms**—IEEE 1588, kernel, Linux, PTP, synchronization

## I. INTRODUCTION

Computer systems have evolved over the last decades from fairly isolated units to highly complex and distributed systems interconnected through various types of communication media. This evolution can be observed in many areas, such as industrial automation and test and measurement. As systems are becoming more complex and more distributed every day, keeping *one* accurate time source for these systems is a challenging task. Applications running in distributed environments need to correlate events, data, or actions to a moment in time in order to act like one greater virtual system.

The Precision Time Protocol (PTP) was designed to achieve very accurate synchronization in distributed environments, for example between nodes that communicate over unreliable and non-deterministic networks. While Linux is becoming the leading operating system for embedded systems, support for IEEE 1588 is only slowly being introduced into the mainstream kernel. We argue that better PTP support, fully integrated into the operating system, is necessary so that all applications can profit from the more precise time keeping.

The IEEE 1588 community has been concentrating for some time on issues like accuracy of PTP clocks, the influences of the network infrastructure on the achievable precision, or improving synchronization accuracy between nodes. Most of the PTP solutions regard the problem purely from an IEEE 1588 perspective, with the focus on how to keep the PTP clock synchronized with high precision to the master. Until now, little attention has been paid to the applications running on top of the system, and issues such as how to synchronize the operating system clock with the PTP clock have been completely left aside. In our opinion, this is a key issue that

must be addressed in order to ensure a wider acceptance of the IEEE 1588 standard in the real world.

Most of today's applications are unaware of the PTP clock and get their time information from the system clock using standard APIs like *time* or *gettimeofday*. It is important to synchronize the Linux system clock to the PTP clock with acceptable accuracy, without imposing the need to rewrite or modify the applications running on top of the operating system.

The paper presents our efforts to synchronize the Linux system clock to the PTP clock. First, we review the previous work done in this area. Next, we propose a solution for synchronizing the system clock. We present the results achieved using our methods. Finally, we discuss conclusions and lay out the steps to be done in future.

## II. BACKGROUND AND PREVIOUS WORK

David Mills, the father of NTP, was perhaps one of the first to struggle with synchronizing the system clock to an external time source. Even though his proposal for a new *kernel model for precision timekeeping* is almost twenty years old [1] [2] [3], many of his methods and results remain quite relevant in the context of the new hardware that we have today. Since NTP does not require hardware timestamps, being a pure software solution, it directly synchronizes the kernel clock and disciplines the frequency of the system clock. Mills described using a Pulse Per Second (PPS) input or external clocks and oscillators to improve the system clock accuracy and stability, limited only by the jitter of the operating system and the stability of the synchronization source.

Introducing IEEE 1588 support into the GNU/Linux operating system required designing and integrating two important services in the kernel, namely control of the hardware clock and packet timestamping at the hardware layer. In February 2009 Patrick Ohly introduced support for hardware assisted timestamping into Linux version 2.6.30 and published a modified version of the *ptpd* program using the proposed API [4].

In 2010 a Linux kernel framework for PTP Hardware Clock (PHC) support was proposed, as described by [5], who also provide experimental evidence demonstrating the soundness of that solution. The framework has gone through several rounds of review on the mailing list and received positive feedback. Background support [6] for the PHC patch series was merged into kernel version 2.6.39, and full PHC support is expected

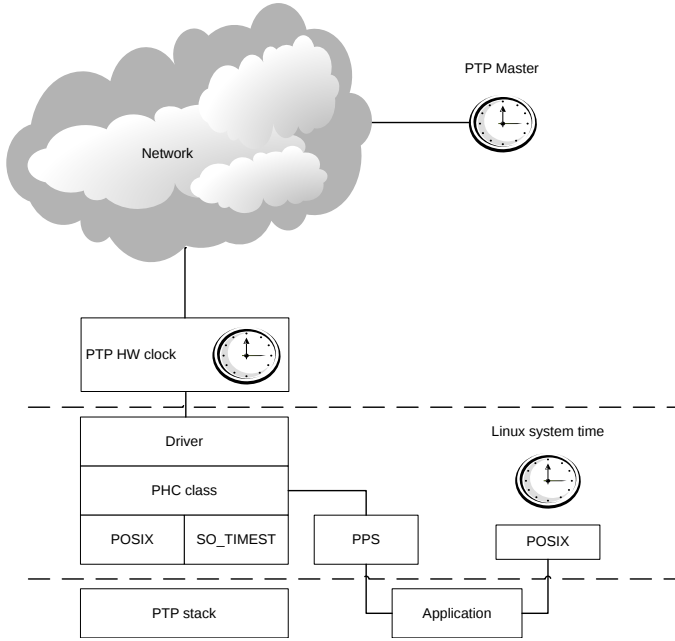


Fig. 1: Synchronizing the System Clock to the PTP Clock

to appear in version 3.0.0. Although timestamping and PHC support have been successfully introduced into the kernel, one issue still needs to be addressed, namely how to accurately synchronize the Linux kernel to the very precise PTP time source. Figure 1 gives an illustration of the Linux PTP hardware clock subsystem. We will return to this figure and to the questions surrounding the system clock in Section IV.

Patrick Ohly [7] proposed two methods to achieve this goal. The first he called “assisted system time,” and the second “two-level PTP.” This second method bears some resemblance to our proposed solution, and we discuss it in some detail, below.

Ohly’s first solution uses the PTP hardware clock for timestamping only, adding an offset to the timestamp which corrects the difference between the PHC and the system time. The basic idea is to directly control the system clock from the *ptpd* program, but to track the difference between the two clocks in the Ethernet device driver. Since the two clocks run with different frequencies, their offset wanders, and so must be measured periodically. Each measurement reads the PHC once and the system clock twice, once before and once after the PHC reading. To calculate the time offset, Ohly averages a sequence of ten measurements, first removing outliers by eliminating 25% of the data points. Ohly arrived at the number of averages based on trial and error on a particular hardware. An implementation of Ohly’s “assisted system time” was merged into the Linux kernel along with the hardware timestamping support, under the name *timecompare*.

One significant problem with this approach is that the number of repetitions is hard coded into each individual Ethernet driver, but the actual number needed to achieve good results depends on the system as a whole. The measurements are performed in kernel space to avoid disruption, and so tuning the number of repetitions requires recompiling the

TABLE I: *Cyclicttest* latency in microseconds

Arch	Load	Min	Avg	Max
x86	idle	6	94	3445
	CPU load	7	1855	1202288
	cache thrash	24	620892	5606967
ppc	idle	9	11	84
	CPU load	9	1118	29391
	cache thrash	9	962	4766

kernel. In general it is impossible to choose one set of driver parameters that will work on all different kinds of systems. The *timecompare* method is further flawed in the assumption that the PHC time reading occurs exactly midway between the two system time readings. While this may be true for PHCs integrated into a System-on-Chip or connected via a fast bus, this assumption does not hold in general.

We tried comparing the system time on an Intel ARM IXP425 with the PHC time from a National Semiconductor DP83640 PHY and found that the *timecompare* result was not stable. On this platform, reading the PHC goes over the MDIO bus clocked at 2.5 MHz. Each MDIO bus transaction takes at least 64 bus cycles. One read requires five transactions with the PHY registers. Our experiments demonstrate a delay of approximately 170-190  $\mu$ s for one read operation, yet the PHC time value is latched in the PHY on the first read.

Clearly, the assumption that the PHC timestamp occurs halfway between the system timestamps is only true for some particular kinds of hardware. In addition, hard coding important synchronization parameters into Ethernet drivers makes it much harder to correctly tune a system. Since the parameters are related not to the driver, but rather to the system as a whole, in our view hard coding them into the kernel is the wrong approach.

Ohly’s second idea, the “two-level PTP” method, is somewhat similar to using a PPS to synchronize the system clock to the PHC. This method proposes using two instances of the PTP stack. The first synchronizes the PTP hardware clock to the PTP master over the network, and the second one adjusts the system time to the PHC. Ohly postulated that the drawback of this solution would be increased system complexity. In our opinion, the idea is worth pursuing, but we agree that running two instances of the PTP software on the same node is rather impractical.

### III. LINUX SYSTEM LATENCY

The idea of offering the PTP hardware clock to the Linux kernel as a combined clock source and clock event device was considered but ultimately rejected, as discussed in [5]. Although this approach would obviate the need for internal PHC-system synchronization, it was quite obvious that certain kinds of hardware clocks cannot be used in this way, since they are far too slow, for example PHY chips addressed over the MDIO bus. Instead, the PHC subsystem depends on the PPS subsystem to synchronize the Linux system time to the PTP clock. We support this idea, arguing that the synchronization accuracy would be good enough for most applications without

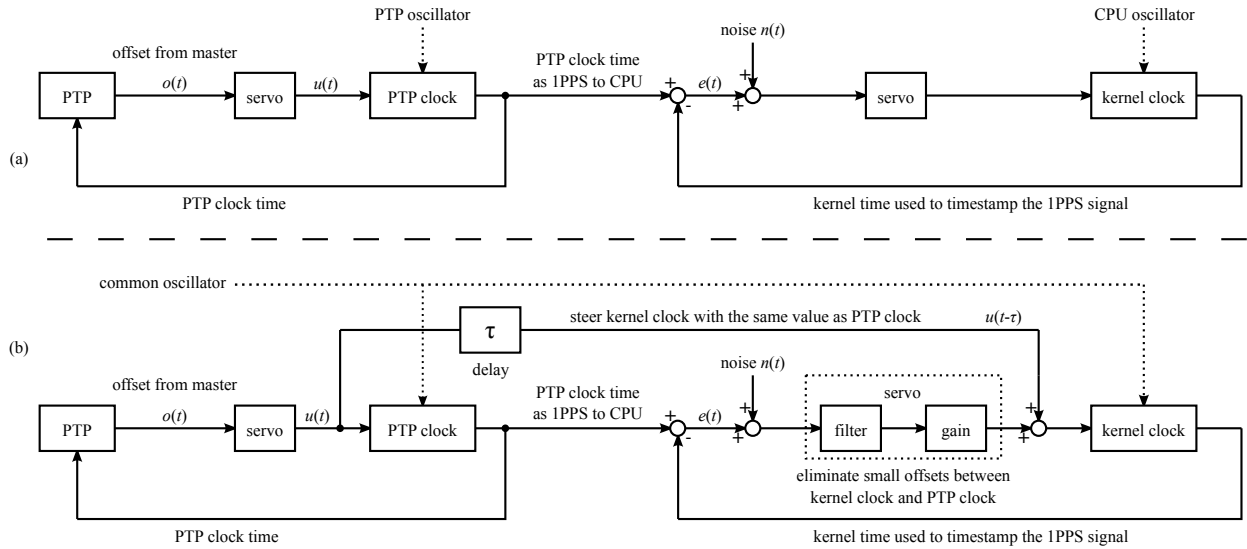


Fig. 2: Clock servo topologies (a) The PTP clock and the kernel clock are driven by separate oscillators. Synchronization of the clocks is established by two independent control loops. (b) Driving the clocks by a common oscillator reduces the influence of the noise  $n(t)$  caused by operating system latencies.

imposing the need to rewrite them to use the PTP clock. Applications with more demanding time requirements can always use the new PTP interfaces directly.

Interrupt and scheduling latency determine on how quickly sleeping tasks can be switched to running mode. If the synchronization accuracy of the system clock to the PTP clock is much better than the scheduling latency of the system, we argue that the achieved synchronization is good enough for most applications. To support this claim, we briefly examine current Linux kernel interrupt and scheduling latency.

Since version 2.4, the Linux kernel scheduler has been greatly improved. In the past, the scheduler had one execution queue for all processors, resulting in  $O(n)$  scheduling complexity. The new scheduler was improved to have  $O(1)$  complexity, each processor having its own running queue [8]. Reducing kernel and user space latency is an ongoing effort. The PREEMPT\_RT branch has been partially merged into mainline Linux, resulting in improved real time performance. Still, depending on kernel version and hardware architecture, Linux exhibits typical scheduling latencies in the millisecond range.

In order to illustrate current Linux performance, we discuss measurements using the *cyclictest* program on different hardware platforms. Stemming from the Linux PREEMPT\_RT kernel development effort, this program measures overall system latency from the application point of view. By repeatedly yielding the CPU for a specific duration and comparing with the actual time blocked, *cyclictest* measures overall scheduling latency of a user space program.

Table I shows the results of our latency tests. We ran *cyclictest* with command line arguments `-i 10000 -l 100000` on an IBM Lenovo W510 with an Intel Core i7 CPU running Ubuntu kernel 2.6.32-29-generic-pae and on the Freescale P2020RDB PowerPC platform running kernel 2.6.39. The

tests ran both under heavy load and when the machine was otherwise idle. The average scheduling latency without CPU load did not exceed  $100 \mu\text{s}$ . While minimum values reach  $5\text{--}10 \mu\text{s}$  without load, absolute maximum values can exceed one second, especially when running cache thrashing programs.

Although latency measurements vary widely depending on the hardware architecture, kernel version, and system load, we find that our measurements are in broad agreement with results reported by others [9] [10] [11]. We can conclude that, if the synchronization accuracy between the PTP and system clocks is kept within  $100 \mu\text{s}$ , this should be good enough to have accurate time information for most of the applications running on top of the system.

#### IV. SYNCHRONIZING THE LINUX SYSTEM CLOCK

A block diagram of the Linux PTP hardware clock subsystem is shown in Figure 1. Starting at the top, timestamped packets from the PHC are made available to the PTP stack via the SO\_TIMESTAMPING socket option. The PTP stack calculates an appropriate correction and tunes the PHC using standard POSIX clock functions. At the same time, a PPS signal from the PHC is timestamped by the kernel in an Interrupt Service Routine (ISR), and this is made available to a user space program via the standard NTP interface for PPS. The main focus of our study is on the detailed operation of this program.

Figure 2a depicts a typical clock servo topology used to synchronize both a PHC and the Linux kernel clock to the PTP master of the network. This topology represents a system where PHC and kernel clock reside on two distinct hardware components, for example a network card and a CPU, driven by two separate oscillators. Such a constellation is typically found on a standard PC.

A PTP implementation running on the system determines the time offset  $o(t)$  between the master and the slave clock. The first clock servo aims to minimize this clock offset by tuning the PTP clock using the control signal  $u(t)$ . Once the PHC is properly synchronized to the master clock, the task is to use the precise time information to discipline the Linux system clock.

We discussed two basic approaches to the problem of estimating the value of the clock offset  $e(t)$  in Section II. One can either read the clock value of both clocks and calculate the clock offset or use the PPS timestamps of the signal generated by the PHC. In both cases, since these measurements are essentially software timestamps, they are perturbed by the system latencies discussed in Section III. The quantity  $n(t)$  represents the measurement noise introduced by these latencies. Our experiments have indicated that this noise  $n(t)$  may be an order of magnitude higher than the actual clock offset  $e(t)$ , negatively affecting the performance of the control loop. As an example, Figure 3 shows a histogram of PPS timestamps taken on the Freescale P2020RDB while under heavy load. These data were collected by setting the system time to the PHC time and then observing the PHC's PPS offset as seen by the system clock over a few minutes. Depending on system activity, the ISR can be delayed a dozen microseconds or more. Even though these particular data were acquired with an unrealistically heavy load (see the script in Section V), still we observed similar latencies when the system was idle, albeit with much lower frequency of occurrence.

Designers of embedded systems are not generally bound to the architecture of Figure 2a. For these systems, we suggest a different approach that may result in better synchronization between the PHC and the system clock. System-on-Chip devices like the Freescale P2020 integrate CPU and PTP hardware clock on the same chip, operating the CPU clock and the PHC from the same oscillator. In other designs where the PHC is external to the CPU, a common oscillator can be used to drive both the PHC and the CPU (and thus the kernel) clock.

Figure 2b shows a configuration where both the PHC and the kernel clock are driven by the same oscillator. As in Figure 2a, the PTP stack derives  $o(t)$ , the offset between the slave and the master clock. A clock servo again is used to tune the PHC. After the control loop has settled and the PHC is aligned to the master clock, the control signal  $u(t)$  is only adjusted to correct for the wander of the local clock.

The variations in oscillator frequency now affect both the PHC and the kernel clock in the same way, and the control signal required to maintain synchronization is the same for both clocks. We therefore directly feed  $u(t)$  not only to the PHC but use it also to tune the kernel clock. Since we cannot set the new tuning value for both clocks simultaneously, a time delay  $\tau$  is introduced, but seeing as this time delay is typically small compared to the sampling time of the control loop (the PTP sync rate), it may be neglected.

Since the two clocks are driven by the same oscillator and are tuned by the same signal  $u(t)$ , they are syntonized. The

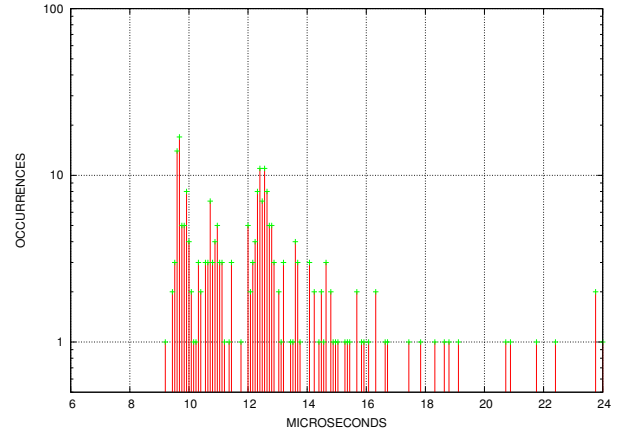


Fig. 3: PPS Timestamping Under Heavy Load

second control loop is only required to eliminate the constant phase offsets between the clocks. Although the controller input  $e(t) + n(t)$  of this servo is affected by measurement noise due to system latencies, the controller must only correct the small influence of  $\tau$ . Thus it can be tuned slowly by using appropriate filters to remove the measurement noise.

## V. PROOF OF CONCEPT

We conducted a pair of experiments to determine just how well the system clock may be synchronized with the PHC by using a PPS. For these tests we used the Freescale P2020RDB and a Meinberg Lantime M600 master clock. To test the case where the PHC and system clocks are operated as in Figure 2a, we used a version of the *ptpd* [12] that has been extended to use the PHC interface. We ran this program as a slave-only ordinary clock and synchronized with a PTP master clock over the network. Synchronizing to the PTP master naturally forces the PHC to diverge from the system clock.

To discipline the system clock, we wrote a simple program implementing a proportional-integral (PI) controller. The clock servo's transfer function is given by

$$\frac{U(z)}{E(z)} = k_p + \frac{k_i z}{z - 1}. \quad (1)$$

Here,  $U(z)$  and  $E(z)$  are the Z-transforms of the frequency adjustment that is applied to the clock and the offset between the PHC and the system clock, respectively. The coefficients  $k_p = 0.0784$  and  $k_i = 0.0016$  were found by pole placement and showed good controller performance on our system. The sampling time was one second.

In addition to reading the PPS timestamps and tuning the system clock, this program also checks the actual clock offset using the *timecompare* method discussed in Section II.

We ran our tests under two different kinds of system load. In order to determine the absolute worst case synchronization behavior, we first ran the tests under an extreme system load. Then, in order to provide a comparison with various other published results, we ran the tests under a much lighter load.

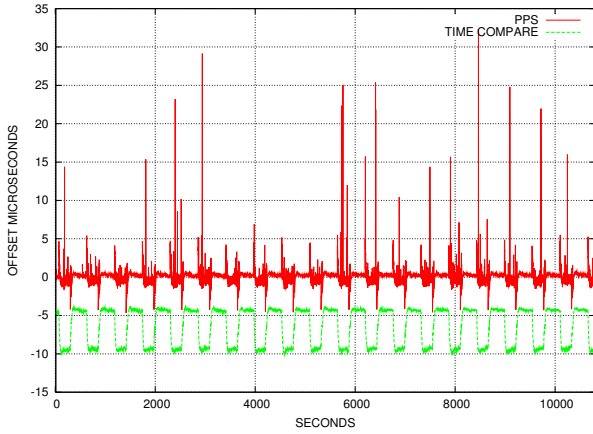


Fig. 4: Three Hour Synchronization Performance

In order to study the effect of system load, we let the following script run during the test, which produces an endless sequence of about five minutes of extreme load followed by five minutes idle time.

```
while [ 1 ] ; do
  find /
  dd if=/dev/zero of=/dev/null count=1000000
  ./hackbench -g 2 -T 8 -s 10000 -l 10000
  ./calibrator 400 2M /tmp/out
  sleep 300
done
```

The result of the test over a three hour period is shown in Figure 4. The top line shows the received PPS timestamps, and the bottom line shows the result of the *timecompare* measurements.<sup>1</sup> The periodic disturbances from the load script are clearly visible, as are occasionally ISR latencies of up to about 30 microseconds. The apparent effect of the heavy system load is a time error of about six microseconds.

<sup>1</sup>The *timecompare* offset of about four microseconds merely reflects the average ISR latency. Normally the PPS timestamps would be corrected by this offset, but we did not do this.

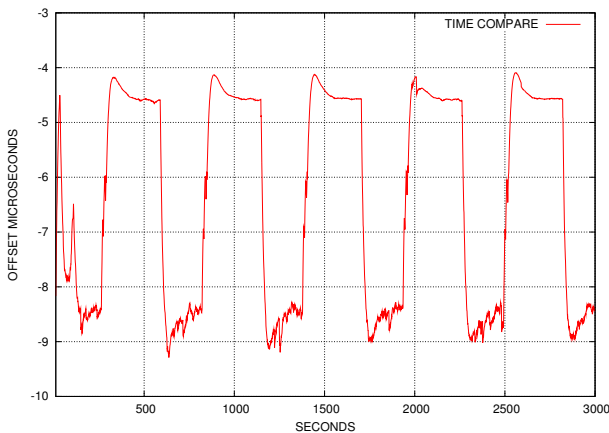


Fig. 5: PHC and System Clock Synchronized

In order to evaluate the servo scheme of Figure 2b, we repeated the test with a hybrid *ptpd* version, altered to adjust the PHC and system clock at the same time. In this way, a single program, the *ptpd*, implements both the PTP stack and the PPS servo.

At first glance, the performance of the syntonized servo scheme appears to be about the same as with the first scheme. However, a closer look uncovers some performance differences. Figure 5 shows detail of the syntonized test compared with Figure 6 from the first test. Note that Figure 6 shows a detailed view of the lower trace from Figure 4. When the clocks are syntonized, the overall peak-to-peak time error due to system load is reduced by about one microsecond. In addition, in the absence of the system load (during the excursions in the upper half of Figure 5) the curve appears much smoother.

Considering the typical Linux scheduling latencies discussed in Section III, the synchronization performance achieved here seems quite acceptable. From these experiments, it appears that using the PPS from the PTP hardware clock to the system clock can provide good results. Even under an extreme system load, the induced drift did not exceed six microseconds.

Figures 7 and 8 show the synchronization performance under light system load with our syntonization scheme and without it, respectively. The effect of the load is a one microsecond time error, visible at the 600 second mark. It is instructive to compare these with Figures 5 and 6. Once again, when the clocks are syntonized, the peak-to-peak time error is less and the trace is smoother overall. These results compare favorably with various other methods using software time stamping. Correll [12] reports synchronization within 10  $\mu$ s on a “fairly busy” m68k CPU. Similar results are reported for the RADclock [13] method under “very light system load.” Our experiments clearly demonstrate how synchronization based on software time stamping methods is significantly affected by system load. In our view, measurements reported without exactly specifying the system load are probably optimistic and should be viewed with healthy skepticism.

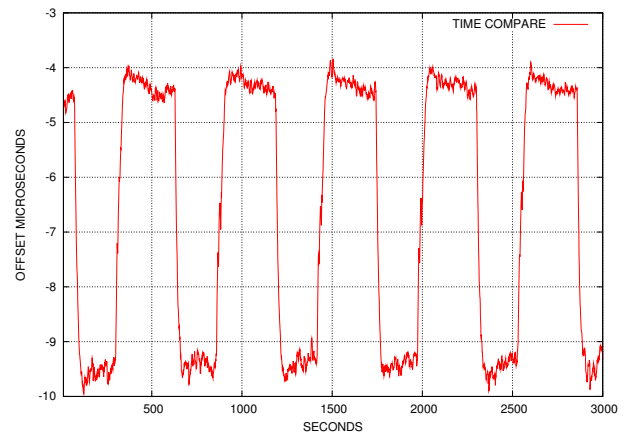


Fig. 6: PHC and System Clock Separate



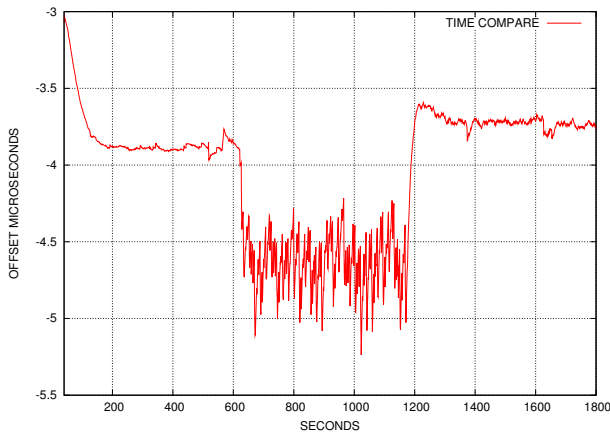


Fig. 7: PHC and System Clock Syntonized

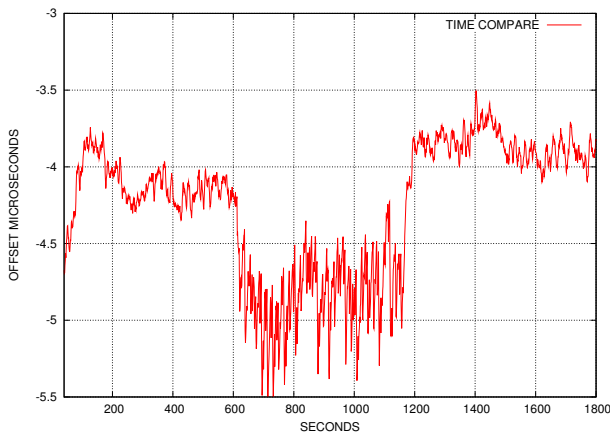


Fig. 8: PHC and System Clock Separate

## VI. CONCLUSIONS AND FUTURE WORK

Synchronization of the Linux system clock to the PTP hardware clock is of great importance, since it will enable all applications to benefit from the more precise PTP time source. The solution presented here eliminates the need to rewrite applications to directly use the PTP clock. Our efforts have been focused on synchronizing the clocks, and we presented two clock servo topologies to achieve this. The servo architecture using two different oscillators can be used in any device running Linux. Embedded systems and System-On-Chip designs can take advantage of the improved architecture, which features a common frequency source. In this case, the clocks are syntonized, and the second control loop is only required to eliminate small offsets between the PTP hardware clock and the system clock.

The tests also demonstrate the soundness of the concept. The synchronization accuracy is, in our opinion, good enough to be used in a wide range of applications. The proposed solution has also the advantage of using standard kernel components (the PHC and PPS subsystems), without imposing the need to modify or recompile the kernel.

Although the existing Linux *timecompare* method remains

useful for evaluating synchronization experiments, when contrasted with well established PPS servo methods it appears to be an unattractive, ad hoc solution. One great advantage of our proposal is that it keeps the clock servo programs in user space. In this way, system administrators and end users can easily adapt the servo algorithms to their particular needs.

However, there is more work yet to be done. Support for feeding the PPS timestamp directly into the NTP servo in the kernel (the so called “hard PPS” method) was added into kernel 2.6.38. It would be interesting to compare the synchronization performance of this kernel space method to our user space approach. Although the results of our first experiments are quite promising, still we should like to test a wider variety of hardware and software combinations.

We expect that our solution will have a positive impact on the acceptance of PTP within the open source community, enabling IEEE 1588 to become a standard synchronization option for the GNU/Linux operating system. Only when PTP solutions are fully integrated into the operating system will we reap the full benefit of IEEE 1588 for the great majority of Linux devices and applications.

## REFERENCES

- [1] D. Mills, “Modelling and analysis of computer network clocks,” University of Delaware Electrical Engineering Department, Tech. Rep., May 1992.
- [2] —, “Precision synchronization of computer network clocks,” University of Delaware Electrical Engineering Department, Tech. Rep., November 1993.
- [3] —, “RFC 1589. a kernel model for precision timekeeping,” University of Delaware, Tech. Rep., March 1994.
- [4] P. Ohly. (2009) Precision time protocol - temporary fork with support for hardware timestamping. [Online]. Available: <http://github.com/pohly/ptpd>
- [5] R. Cochran and C. Marinescu, “Design and implementation of a ptp clock infrastructure for the linux kernel,” in *Precision Clock Synchronization for Measurement Control and Communication (ISPCS)*, 2010 International IEEE Symposium on, Sep. 27–Oct. 1, 2010, pp. 116–121.
- [6] T. Gleixner. (2011) Rework of the ptp support series core code. [Online]. Available: <http://lkml.org/lkml/2011/2/1/137>
- [7] 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, 2008. [Online]. Available: [http://www.linuxclustersinstitute.org/conferences/archive/2008/PDF/Ohly\\_92221.pdf](http://www.linuxclustersinstitute.org/conferences/archive/2008/PDF/Ohly_92221.pdf)
- [8] T. M. Le. (2011) A study on linux kernel scheduler version 2.6.32. [Online]. Available: <http://www.scribd.com/doc/24111564/Project-Linux-Scheduler-2-6-32>
- [9] S. Kai and Y. Liping, “Improvement of real-time performance of linux 2.6 kernel for embedded application,” in *Computer Science-Technology and Applications, 2009. IFCSTA '09. International Forum on*, vol. 2, Dec. 2009, pp. 71–74.
- [10] C. Williams. (2011) Cyclicttest. [Online]. Available: <https://rt.wiki.kernel.org/index.php/Cyclicttest>
- [11] W. Grandegger. (2011) gpiorqbench: measuring external interrupt latencies. [Online]. Available: [http://www.denx.de/wiki/DULG/AN2008\\_03\\_Xenomai\\_gpiorqbench](http://www.denx.de/wiki/DULG/AN2008_03_Xenomai_gpiorqbench)
- [12] K. Correll, N. Barendt, and M. Branicky, “Design considerations for software only implementations of the IEEE 1588 precision time protocol,” in *Proceedings of the IEEE 1588 Conference*, Zurich, October 2005. [Online]. Available: [http://ptpd.sourceforge.net/ptpd\\_2005\\_1588\\_conference\\_paper.pdf](http://ptpd.sourceforge.net/ptpd_2005_1588_conference_paper.pdf)
- [13] J. Ridoux. (2010) Radclock accuracy and robustness. [Online]. Available: <http://www.synclab.org/radclock/performance.php>