ISPCS 2009 International IEEE Symposium on Precision Clock Synchronization for Measurement, Control and Communication
Brescia, Italy, October 12-16, 2009

Counter Availability and Characteristics for Feed-forward Based Synchronization

Timothy Broomhead, Julien Ridoux, Darryl Veitch
ARC Special Research Center for Ultra-Broadband Information Networks (CUBIN)
An affiliated program of National ICT Australia
EEE Department, The University of Melbourne, Australia
t.broomhead@ugrad.unimelb.edu.au, {jridoux, dveitch}@unimelb.edu.au

***Abstract*— The availability of hardware counters in computers is essential both to the applications in charge of timekeeping, and those in need of accurate timestamping. Newer counters are now supported by open source operating systems, but the access interfaces are unnecessarily restricted, and in particular fail to satisfy the needs of feed-forward based synchronization algorithms. In this paper we present modifications to the Linux and FreeBSD kernels to enable any application to access all available counters in an unrestricted way, and then evaluate their stability, latency and robustness to stress. We demonstrate how the feed-forward based RADclock can, through this interface, make use of any of several counters, and achieve the same micro-second synchronization with each.**

***Index Terms*— RADclock, TSC, HPET, ACPI, time counters, clock source, feed-forward, synchronization**

I. Motivation

For more than 20 years, the NTP daemon [1], [2] has been the reference for designing computer clock synchronization algorithms over networks. The core of its algorithm for tracking the drift of a host computer's clock is based on feedback, more specifically an implementation of a PLL/FLL (Phase/Frequency Lock Loop). In the same spirit, open source operating system kernels provide support for feedback synchronization algorithms by implementing the IETF RFC1589 standard [3]. This support, embodied by the *gettimeofday()* and *adjtime()* system calls, inherently creates a feedback loop between the kernel time adjusted by the synchronization algorithm, and the timestamps fed to that same algorithm based on the system clock maintained in the kernel.

While this approach has been successful in synchronizing computers at the millisecond scale, the limits of a pure feedback algorithm are apparent at the 10 micro-second scale. The characteristic of the noise produced by network communication invalidates the default assumption that timestamps are dominated by Gaussian noise with few spikes, and can potentially seriously impair a feedback synchronization algorithm [4].

The TSCclock [5], [6], [7], [8], [9], [10] is a particular implementation of our Robust Absolute and Difference clock (RADclock) algorithm capable of overcoming the limitations of feedback algorithms in noisy environments such as networking. The RADclock provides an absolute time and a difference clock (a clock uncorrected for drift used to measure time differences very accurately), each synchronized via a unique feed-forward algorithm and a highly robust noise filter. The RADclock is based on the availability of a raw hardware counter which is continuously increasing and which does not roll over. In particular, this is essential for the difference clock and more generally for any feed-forward based processing of the raw counter data.

As of today however, no operating system provides generic support for such a counter. The timing support currently available in kernels is intrinsically tied to feedback applications. Up until now, the TSCclock has circumvented this problem by using the Time-Stamp Counter (TSC) that counts CPU cycles. The TSC counter is 64 bits wide (it would take almost 200 years to roll over at today's CPU frequencies), and it is readily accessible with the few lines of assembly code composing the *rdtsc()* function from either kernel or user space.

Unfortunately, in recent hardware architectures, features such as power management, frequency stepping, and unsynchronized multi-cores affect the stability and consistency of the values returned by the *rdtsc()* function. Until a robust solution is provided by CPU manufacturers, disabling these features (when possible) is a short term solution for the TSC to be used as a timekeeping and timestamping source, but it is not realistic in a production environment.

Fortunately, the TSCclock is only one possible implementation of our RADclock algorithm, whose principles are not bound to any particular hardware infrastructure and can be used with any counter satisfying basic stability criteria. In this paper we first present a set of kernel modifications that provide support for applications relying on a feed-forward paradigm. These modifications overcome the above limitations on the use of the TSC counter by providing an interface to access any hardware counter available on modern architectures. Using this interface we then study the characteristic of the most prominent modern counters present on computer motherboards and compare their stability, latency and robustness under realistic load and stress scenarios. Finally, we discuss the impact of counter choice on RADclock performance, as well as on the quality of RADclock timestamps accessible to applications.

II. Kernel support for feed-forward algorithms

Nowadays, personal computers embed the Time Stamp Counter (TSC), the Advanced Configuration and Power Interface (ACPI) timer, and often the High Precision Event

978-1-4244-4392-5/09/$25.00 ©2009 IEEE

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

TABLE I
COUNTER CHARACTERISTICS

Timer (HPET). These hardware counters are initialised to 0 at system boot and incremented at the period of their respective oscillators. Since each counter can be used for timekeeping, the operating system selects the one it considers the most reliable at boot time and provides an interface to access it. This interface is named *timecounter* [11] on FreeBSD and *clocksource* on Linux and is internal to the kernel.

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 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*.

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 driven by the information passed by the synchronization algorithm to the kernel via the *adjtime()* system call.

The kernel's notion of time and the synchronization algorithm are locked together. A feed-forward algorithm cannot take advantage of the existing mechanism since the tracking of the oscillator drift is already coupled to the timestamping mechanism. A new mechanism is therefore required, and 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 (run at very close to constant frequency) 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, is guaranteed not to roll over. To allow user space programs to access this counter, we also implemented a new *getcounter()* system call.

[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.

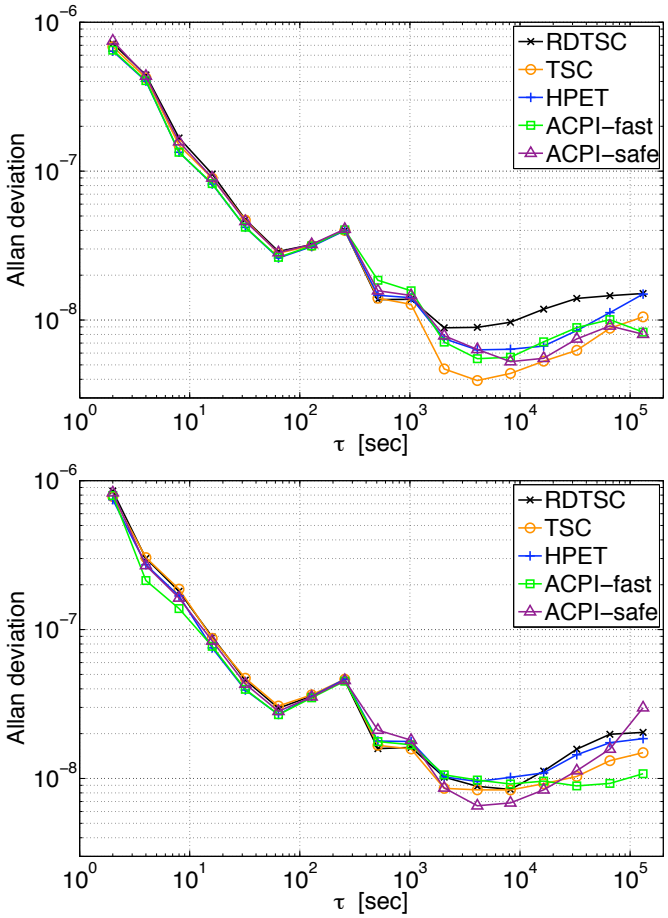

Fig. 1. Stability of 4 counters based on PPS based polling in a: Top – desktop computer (*tastiger*); Bottom – rack server (*platypus*).

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 in an elegant manner while taking advantage of the existing implementation, a feature that should ease its adoption by the open source community. It is also generic enough to give access to future hardware counters as soon as they are supported by the *timecounter* or *clocksource* interfaces [12]. 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.

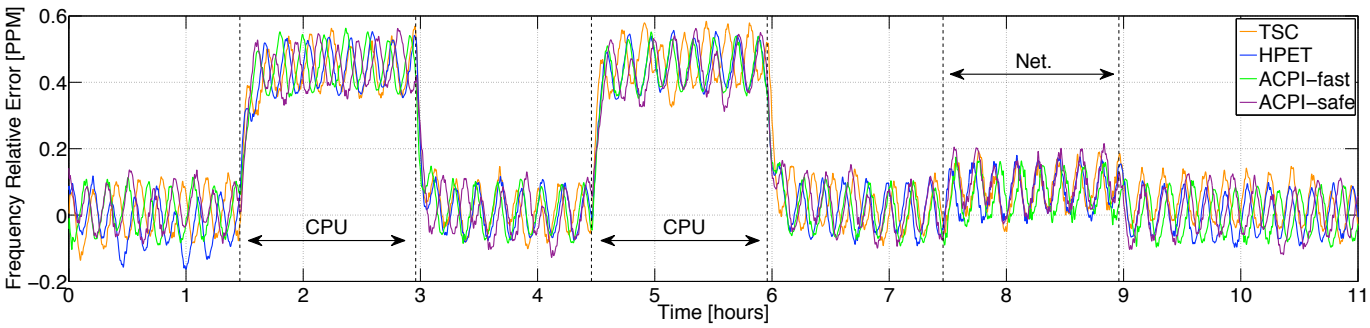

Fig. 2. Instantaneous relative frequency error of each counter under CPU load and network interrupt stress scenario on the rack server *possum*.

III. Counter Characteristics

In the following section, we compare the different counters available and examine some of their key characteristics. For this purpose we use recent computers that embed the TSC, HPET and ACPI counters. We use a FreeBSD 7.0 system that provides two access methods to the ACPI counter [13]. The *fast* method simply reads the counter as quickly as possible. The *safe* one is intended for ACPI architectures that may not correctly latch the counter, and compensates for this defect by reading the counter several times until a correct value is read, making this method 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 compares these counters.

A. Stability

Our testbed at the University of Melbourne is equipped with a PRS-10 rubidium oscillator whose Pulse-Per-Second (PPS) signal is locked onto a Trimble Acutime Gold GPS receiver. Using the FreeBSD PPS API [14], the kernel captures the PRS-10 PPS signal through a serial port. On each pulse, a raw 64 bit cumulative counter timestamp is created from within the kernel and exported and converted to seconds based on a long term period estimate computed from the GPS receiver time.

This set-up enables us to observe the stability of each counter when accessed via the *timecounter* interface. Figure 1 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 allow comparison to our previous work [9].

Figure 1 shows that all the counters we observe exhibit a stability below 1PPM at any timescale. 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. The counters exhibit then 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.

Interestingly, all counters on both test machines exhibit a "bump" in the Allan deviation. The time scale of this bump corresponds to the period of the air conditioning system controlling the temperature in the server room. This periodic change of temperature affects all counters in the same manner. This is of interest for production environments that are equipped with similar systems, and even if the impact of the air conditioning system is noticeable it remains well below 0.1PPM. A similar bump was noted in [7].

In conclusion, the counters exhibit virtually identical stability characteristics which are far better than their specification and equivalently useful for the purpose of timekeeping.

B. Stability Under Stress

We now 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% of its capacity, allowing us to observe the counter's behaviour under heavy load. 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 100Mb/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.

Before running these tests the expectation was that the CPU load stress would affect the TSC counter more than the others since it is located on the 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.

Using the PPS signal capture in the kernel, we compute the number of cycles elapsed between two consecutive pulses, giving us a direct estimate of the "instantaneous" frequency of each counter. Using the "normal conditions" period of the test, we compute a reference frequency for each counter. Figure 2 shows the error of the instantaneous frequency relative to the respective reference frequency expressed in PPM. Each data

set has been captured over a period of 11 hours on the rack server machine *possum*. The captures occur sequentially but are aligned in the plot relative to the beginning of each run of the stress scenario.

Figure 2 indicates that our preconceptions were wrong: the frequencies of all counters change in a comparable manner when under stress. However, it points at 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 most probable explanation is that the crystal and/or the clock synthesizer chip present on the computer motherboard are affected by the temperature changes and as a consequence all counters are affected in a similar manner. The high frequency oscillations shown in Fig. 2 correspond to the impact of the air conditioning system seen in the Allan deviation earlier.

This result shows that all counters are equivalently stable under heavy load. This experiment also reveals that even under heavy load, the frequency variation remains below 1PPM, a crucial result for server machines susceptible to carrying out heavy computing tasks. Finally, we see that an extremely heavy network activity has a very small impact on the counters, an important feature for network intensive applications such as network monitoring and network anomaly detection that are dependent on packet capture and timestamping.

C. Latency Under Stress

While the counters have similar stability characteristics, they have different access methods. The value of the TSC can be returned extremely fast and accessing it requires few assembly instructions[2]. The HPET and ACPI counters are accessed via reading on a data bus and are regarded as providing slower access. The HPET counter however, is memory mapped and should therefore be faster than ACPI.

To unambiguously measure the latency of each counter, we count the number of CPU cycles required to access a given counter via the kernel interface. Using the fast *rdtsc()* function, and memory barriers to prevent instruction reordering, consistent values of the TSC counter are read before and after accessing the selected counter via the interface.

Figure 3 shows the latency of the counters expressed in CPU cycles, a metric that is independent of CPU frequency variations. As presented in section III-B, the computer has been subject to the same stress scenario over periods of 11 hours, and the counter latency has been measured every 0.5 s.

The corresponding distributions of the latency for each phase of the stress scenario are also presented in figure 3. Each distribution is 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.

It is worth noting that the values shown here are effectively the sum of the latency of the *timecounter* interface and the latency of the counter itself. A we will discuss in the next section, the former can accurately be modelled by a constant.

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

We first look at the normal periods when the system is not under stress. As expected, the TSC is the fastest counter and the median access latency is 420 CPU cycles (less than 140 ns on a 3GHz 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 initial results highlight the first difference between the different counters. While showing similar stability characteristics, the latency in accessing them can impact the performance of the kernel timestamping mechanism. These values are far from negligible since they reach the micro-second level even when assuming a fast modern computer.

The variability of the latency to access the counters is small however under normal condition. The Inter-Quartile Range (IQR) for the TSC is essentially null because of the high level of discretisation 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 a small variability for all counters is an encouraging fact since it essentially translates to a quasi-constant offset error for a clock synchronization algorithm.

The periods when the computer is under stress show an interesting pattern. As shown by their distributions in Fig.3 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 optimisations. Under CPU load, the caches are "hot" and the apparent performance improvement is an artefact 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. The only noticeable difference concerns the extreme outliers. As shown by the distributions of Fig. 3, their relative representation in the measured data is equivalent to the normal case. However, the corresponding latency values they take are much higher. In other words, the same proportion of counter access code is interrupted, but for much longer in the presence of network interrupts. As an 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" optimisation 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.

IV. Raw Timestamping and Synchronization

The new kernel support for feed-forward synchronization algorithms is a large improvement over the former TSC-centric approach, as it provides a universal access to all

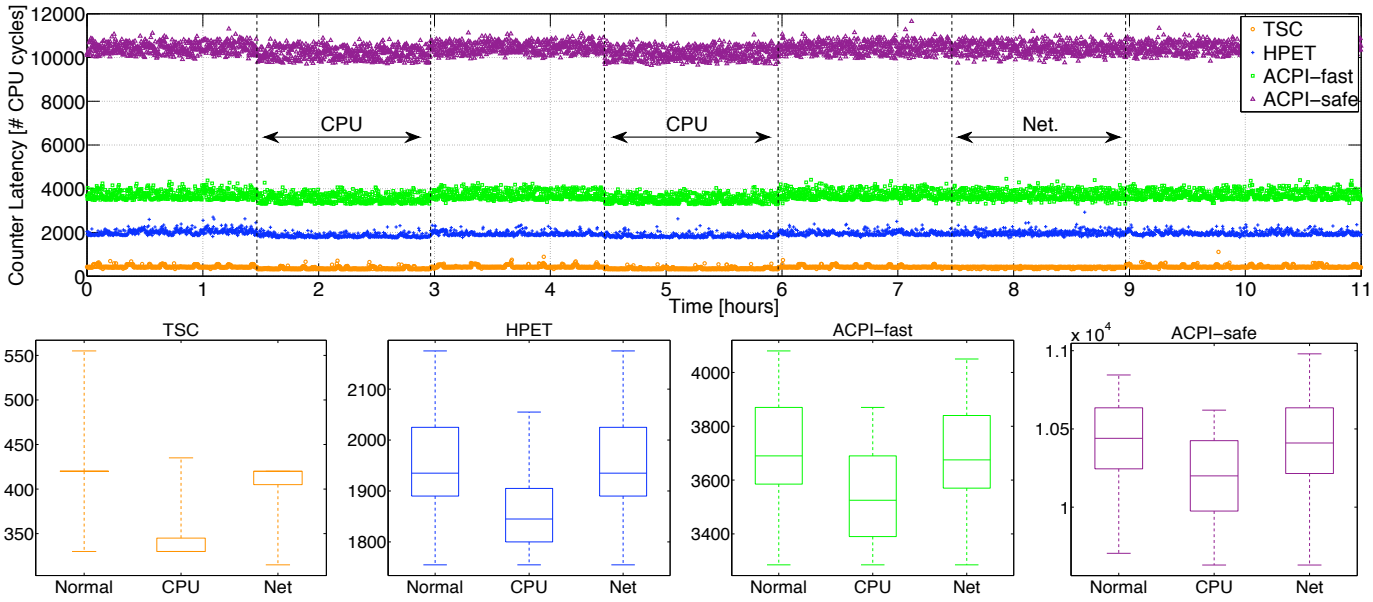

Fig. 3. Counters access latency on FreeBSD for the 4 counters available, under the stress scenario. Time series (top) and corresponding distributions from 0 to 95th percentile.

available counters. The original approach however enjoyed the advantage of quick access not only from the kernel but also from user space.

A. Timecounter/Clocksource Timestamping Latency

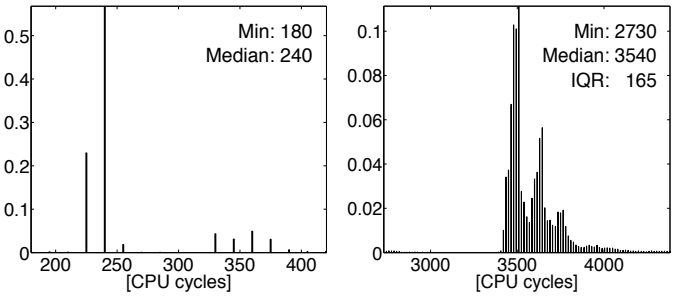

Fig. 4. In kernel *timecounter* interface latency (left), user space *getcounter()* system call (right).

From a kernel perspective, we are first interested in quantifying the overhead produced by the use of the *timecounter* or *clocksource* interfaces compared to the original *rdtsc()* function. For this purpose we implemented a kernel module that performs two calls to the *rdtsc()* function, then reads the TSC counter by using 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 a measure of the latency of the *timecounter* interface alone.

Figure 4 shows that this latency has an extremely compact distribution on FreeBSD. Also, with a minimum value of 180 cycles and a 99th percentile at 420 cycles, the spread of the interface latency is 240 cycles, or only 80 ns on a 3GHz processor. For the purpose of software timestamping and synchronization, the kernel interface latency can then be approximated by its median value, namely 240 cycles. The Linux kernel *clocksource* interface exhibited similar results.

In summary, using the new interface for kernel timestamping adds an extremely small penalty compared to the original *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 a timestamp normally issues a *gettimeofday()* system call to access the system clock. In the case of the original TSCclock, the same program only needed to use the *rdtsc()* function to create a raw TSC timestamp and convert it to 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 a system call to retrieve a raw timestamp from the kernel. Similarly to the kernel case, we compute the latency of the new *getcounter()* system call we implemented to capture raw timestamps from user space.

Figure 4 shows the distribution of the FreeBSD system call latency, and this time, the penalty induced by our solution becomes apparent. Whereas creating a raw timestamp based on *rdtsc()* costed about 150 cycles from user space, it now costs over 3500 cycles (almost 1.2 µs) and exhibits a distribution whose variance cannot be ignored (similar results were found on Linux). Even if the system was running with a perfect software clock, the timestamps would consistently exhibit errors above 1 µs on modern systems. This is even more important for synchronization algorithms that themselves rely on timestamps created on the system. For example, the *ntpd* daemon relies on timestamps created in user space and as such

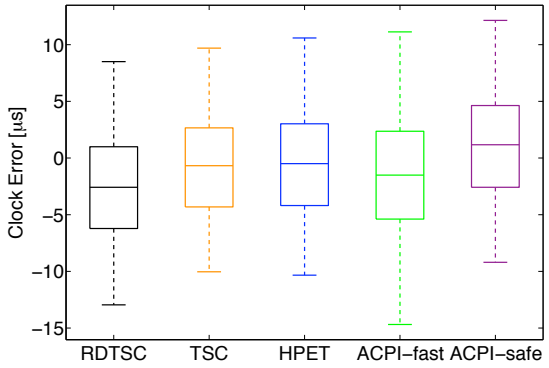

Fig. 5. RADclock final clock error after asymmetry compensation. Compact distribution of use of each counter for *tastiger* from 1st to 99th percentile.

is prone to such timestamping error. This again advocates for the use of timestamps taken from within the kernel.

C. Synchronization Algorithm Performance

As a final result, we are now interested in the impact of each counter on the RADclock, our synchronization algorithm. Using the testbed and methodology described in [15] we are able to compare the error of the RADclock against a clock on a "DAG card", which is a hardware clock synchronized to the PPS signal of our PRS-10 atomic clock.

Figure 5 shows the distribution of the RADclock errors using the original *rdtsc()* function and 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 [15], in order to focus on the error variability.

As expected from the results above on counter stability, the RADclock performs extremely similarly irrespective of the counter selected. 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 our 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.

Finally, we observed the impact of the stress scenario on the performance of the RADclock. While the performance of the synchronisation algorithm is 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 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.

V. Conclusion

In this paper we presented a set of modifications to the Linux and FreeBSD kernels to allow feed-forward applications to access a raw 64 bit consistent counter for the purpose of timestamping and timekeeping. Based on these modifications, we showed that the hardware counters present in common computers are extremely similar and are equally good candidates as the basis of timekeeping and timestamping if done in kernel space. If timestamping is required from a user space program, a small penalty applies due the use of the system call interface that is identical to the use of *gettimeofday()*.

We also demonstrated that our implementation of the RADclock software clock can easily be configured to make use of any of the hardware counters available through the above interface. We showed that the RADclock achieves micro-second level accuracy when synchronised over a LAN independently of which counter is used.

Acknowledgment

The authors acknowledge C. Chambers' work on the implementation of the kernel modifications proposed in this paper. 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] ——, "A Kernel Model for Precision Timekeeping," IEFT, Network Working Group, RFC-1589, 1994.

[4] J. Ridoux and D. Veitch, "The Cost of Variability," in *Int. IEEE Symp. Precision Clock Synchronization for Measurement, Control and Communication (ISPCS'08)*, Ann Arbor, Michigan, USA, Sep 24-26 2008, pp. 29–32.

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

[6] ——, "PC Based Precision Timing Without GPS," in *Proc. ACM Sigmetrics Conf. Measurement and Modeling of Computer Systems*, Del Rey, California, 15-19 June 2002, pp. 1–10.

[7] D. Veitch, S. B. Korada, 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.

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

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

[10] J. Ridoux and D. Veitch, "Ten Microseconds Over LAN, for Free," in *Int. IEEE Symp. Precision Clock Synchronization for Measurement, Control and Communication (ISPCS'07)*, Vienna, Austria, Oct.1-3 2007, pp. 105–109.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] J. Ridoux and D. Veitch, "A Methodology for Clock Benchmarking," in *Proc. IEEE Tridentcom*, Orlando, FL, USA, May 21-23 2007.