Probing the Latencies of Software Timestamping

Benjamin Villain*, Matthew Davis†, Julien Ridoux†, Darryl Veitch† and Nicolas Normand*
*LUNAM Université, Université de Nantes, IRCCyN UMR CNRS 6597, Nantes, France
benjamin.villain@gmail.com, nicolas.normand@univ-nantes.fr
†Department of Electrical & Electronic Engineering, The University of Melbourne, Australia
davis@student.unimelb.edu.au, {jridoux, dveitch}@unimelb.edu.au

***Abstract*—Dealing effectively with latency is the key to accurate and reliable timekeeping over networks. Software components of timekeeping, including synchronisation algorithms such as *ntpd*, *RADclock*, and *ptpd*, must deal with the significant and highly variable latencies inherent to common operating systems. Using the DTrace system profiling tool, we provide an accurate breakdown of the latencies between common timestamping locations in the FreeBSD Operating System. We report on how these latency components react to stress patterns of different kinds, and determine which timestamping strategies result in the lowest latency, and the smallest in-host asymmetry. Our results can be used to improve timestamping and timekeeping for software clocks.**

***Index Terms*—*RADclock*, software clocks, synchronization, accurate timestamping, latency profiling**

I. Introduction

The essential variable to control when synchronising a local clock to a remote reference across a network is latency, which delays both the arrival of timing information, and the act of timestamping. Latencies are introduced within network elements, the local host system, and the time-servers themselves. With strong hardware support across the system end-to-end latency can be low, and more importantly, have very low variability. Software components however introduce much larger delays, and far more extreme variability.

Consider software clocks defined on common multi-tasking operating systems, built on commodity hardware. The latency components arising from such a host are primarily the result of the interrupt processing, process scheduling and context switching nature of the operating system, which both creates and disciplines the software clock, and performs the timestamping. They are present across the system and are ultimately unavoidable. However, by instrumenting the system to reveal the web of component latencies, to determine their sizes, variability, and relevant asymmetries, we can gain important insight into the inherent limitations of software timing. This information can then be used to improve both the software clock and timestamping, and to determine bounds on the associated errors.

In this paper we provide a breakdown of in-host latency components of key importance for timekeeping. These include the latencies between the packet timestamping locations typically used by *ntpd* [1], *RADclock* [2], *ptpd* [3], and *timekeeper* [4], and span user, kernel, and (to a close approximation) NIC based timestamping. Our study is focussed on the FreeBSD operating system, as it is a stable and mature platform within which to perform metrology, and because it supports the *DTrace* system profiler tool. We use *DTrace* to measure the latency components with total error under 1 µs. The host hardware is a common Intel based commodity server.

Our goal is not only to understand the latency breakdown in general terms, but also to examine how latency components change their characteristics under different conditions. To this end we apply a set of stress tests separately targeting kernel interrupt processing, disk access, and the network stack. To the best of our knowledge a careful breakdown of this kind for latency affecting timekeeping has not appeared before.

As an illustration of the potential applications of our work, we include an analysis of the in-host component of round-trip asymmetry corresponding to common choices for pairing packet timestamping locations from the outgoing and incoming directions. This asymmetry, which contributes to overall asymmetry in the host $\mapsto$ server $\mapsto$ host path, impacts directly on clock synchronization algorithms, including all common NTP and PTP clients using software timestamping, which all assume it to be zero. In LAN environments this asymmetry induced error can easily dominate other sources. Our approach can be used to measure and hence remove the host component of this error to the benefit of all host synchronization algorithms. We include a discussion of the suitability of different timestamping choices as a function of system stress scenarios.

The remainder of the paper is structured as follows. Section II provides needed background on the test system, DTrace, and timestamping. Section III describes in detail the locations defining the latency components we measure and evaluates the probe effect. Section IV then presents the main results on the latency components. Section V discusses applications of the results and describes the implications for in-host asymmetry. We conclude in Section VI.

II. Background

In this section we describe our test system, the profiling tool DTrace, and our timestamping methodology.

A. Test System

Experiments were conducted with the test host *dingo*, a 3.0GHz Xeon Quad Core running FreeBSD 9.0 with DTrace and *RADclock* (version 0.4.0) kernel support. Although results are architecture dependent, we would not expect them to vary greatly. Operating system dependence however is strong.

As an external hardware reference we use a 3.7GE DAG card [5] to timestamp packets passing in and out of the host, via a 100Mbps Ethernet hub as tap. The PPS input to the DAG is provided by a PRS-10 rubidium atomic clock, synchronized to a roof mounted Trimble Acutime Gold GPS receiver, resulting in a final precision around $40\,\mathrm{ns}$.

B. DTrace

The "Dynamic Tracing" or DTrace framework [6] was originally developed by Sun Microsystems and released in 2005. DTrace is a free and open source system profiling framework that provides means for low overhead inspection of a system at runtime. It is available for Solaris, Mac OSX and FreeBSD (a port to Linux is incomplete).

A DTrace *probe* is an event in the system which can be monitored. Examples are input/output operations, or execution reaching the entry or exit point of any function or system call. Once a probe is fired, *predicates* are tested to control the execution of *actions* that can examine, and conditionally store, input arguments, the return value, or other information such as the process-id. For our purposes we define probes at points where timestamps are typically made, and the action is simply the making and storage of a timestamp, together with conditional printing actions to assist in timestamp identification.

A key advantage of DTrace is that it does not require modification of the profiled applications, instead placing thousands of pre-compiled probes throughout a system, which can be augmented by user defined probes. Another advantage is that only those probes which are actually activated generate any *probe effect* [7], that is a latency-inducing overhead that could distort the system latencies we seek to measure. Since DTrace probes can operate within kernel context natively, the probing effect of active probes is less than that of user context based solutions like *truss* or *strace*. It is also lower than that of earlier kernel-based tools, as the normal execution of probed functions is not interrupted. In Section III-B we measure the size of DTrace's probe effect for our experiments and describe how we account for it.

C. Timestamping Methodology

In this paper we are ultimately concerned with measuring time differences only, not absolute time.

The timestamping mechanisms of DTrace, *timestamp* and *vtimestamp*, are independent of the system clock. They are based on the TimeStamp Counter (TSC), which counts CPU cycles since boot, and an estimate p of its period. We are only concerned with *timestamp*, which returns a timestamp of $p \cdot \mathrm{TSC}$, and so measures time differences as $\Delta t = p \cdot (\mathrm{TSC}(t_2) - \mathrm{TSC}(t_1))$.

DTrace relies on the OS kernel to obtain an estimate of p, usually computed by counting the number of CPU cycles elapsed during a call to a *sleep(duration)* function whose actual duration is defined by the approximately known period of a second counter (such as HPET). Such estimates have an error, typically of the order of 50 Parts Per Million (PPM), which is a function of the temperature and system load at calibration time, and does not track changes over time. As most of time intervals we measure are under 1ms, the error in DTrace timing should still be below $100\,\mathrm{ns}$ in most cases, smaller than the error due to probe effect. For more general use however, where time intervals may be larger, the error may be unacceptable. In future work we will significantly reduce it by using *timestamp* in conjunction with the *RADclock* difference clock [2], [8].

One of the advantages of using the TSC, enjoyed by DTrace, is that it can be read with 'universally low' latency from either user or kernel context via the *rdtsc()* function, a wrapper for assembler instructions that read the TSC register(s). This fact, combined with DTraces's flexible probe access, means that we can *accurately measure the duration of any time interval across the system, even crossing the kernel/user divide, without any modifications either to the kernel or user programs.*

Care must be taken when using the TSC for timekeeping on systems with multiple CPU cores and/or power management. DTrace is designed to detect and compensate for such effects. Moreover, our test host is a recent generation with an 'invariant TSC', immune to such effects.

III. Experimental Methodology

Our methodology derives from that of previous work, notably [9]. Briefly, it is based on a series of UDP test packets sent from the host to an echo server on the LAN and back, as a set of triggers for timestamps made at a set of measurement points in both the incoming and outgoing directions, both within and external to the host (see Figure 1).

In this paper we use DTrace for all in-host timestamping, both in kernel or user context (the latter is supported from FreeBSD 9.0). For uniformity and convenience, we use DTrace even though in many cases, since we target typical timestamping locations, alternative mechanisms to recover the TSC exist.

We now detail and explain our probe placement. In Section III-B we evaluate the associated probe effect.

A. Probe Locations

Our aim is to cover the full set of timestamping locations normally used to timestamp packets in FreeBSD, namely: userland, the *so_timestamp* associated with sockets, and the *bpf* (Berkely Packet Filter) subsystem. In addition, we take timestamps deep in the driver, as close to the NIC as possible. Together with the external DAG timestamps, these allow us to evaluate the latency component within the NIC itself. More importantly, they also give an excellent approximation of the locations where hardware timestamps on the NIC would be taken, were they supported, and hence enable the associated potential improvement in latency to be evaluated.

The timestamping locations are illustrated in Figure 1. Apart from the network latency $gRTT \approx 210\,\mu\mathrm{s}$ (measured by DAG and compressed in the figure), the timestamps are shown to scale according to actual median values under light load. An exception is T^{o}_{so}, which is triggered roughly as shown, but the timestamp itself, being made of a multicast copy sent back up the stack as a received packet via the loopback interface, occurs must later, as we see in detail below. Since a *so_timestamp* can only be made on a received packet, this

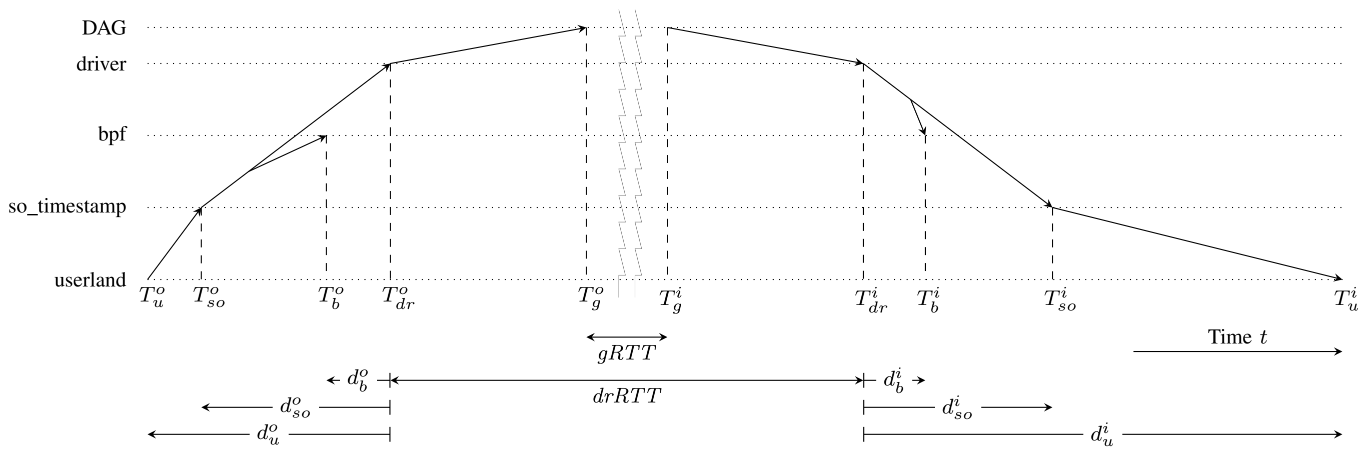

Fig. 1. Timeline showing the locations where probes are placed to track, via DTrace probe timestamps, the passage of a packet from userland, to an external server, and back. The common timestamping locations of userland, *so_timestamp*, and *bpf* (based on packet copies), are supplemented by timestamps close to the host-NIC interface. The role of the external DAG timestamps is to allow the network-side latency, and hence the in-NIC latency, to be evaluated.

trick is the only way in which one can obtain an outgoing *so_timestamp*.

The details are as follows. Unless otherwise specified, the probe is triggered at entry to the listed function.

userland: used by *ntpd*, *ptpd* on send if *so_timestamp* fails.
- **out-probe** (T_u^o) *clock_gettime* (on return) in UDP test client,
- **in-probe** (T_u^i) *clock_gettime* (on return) in UDP test client.

***so_timestamp*:** used by *ptpd*, various active probing tools.
- **out-probe** (T_{so}^o) *ip_savecontrols* via IP loopback interface,
- **in-probe** (T_{so}^i) *ip_savecontrols* in UDP part of IP stack.

***bpf*:** packet socket used by *libpcap*, *RADclock*, *wireshark*.
- **out-probe** (T_b^o) *sysclock_getsnapshot* (on return) in *bpf*,
- **in-probe** (T_b^i) *sysclock_getsnapshot* (on return) in *bpf*.

driver: locations as close as possible to the NIC
- **out-probe** (T_{dr}^o) *user defined probe* in *igb_xmit* in driver,
- **in-probe** (T_{dr}^i) *user defined probe* in *igb_rxeof* in driver.

DAG: used by passive monitors, our testbed validation
- **out** (T_g^o) obtain absolute timestamp using *dagconvert*,
- **in** (T_g^i) obtain absolute timestamp using *dagconvert*.

Because of the need to reliably match the set of probes corresponding to each test packet, probes were sometimes set on functions which called the target clock reading routines. We denote loCation specific round-trip-times (RTT) as $cRTT = T_c^i - T_c^o$, where $c \in \{u, so, b, dr, g\}$. To isolate important RTT components we define the *partial RTTs* as $cRTTx = cRTT - xRTT$ where the eXternal component $xRTT$ has been excised. Thus $gRTT = T_g^i - T_g^o$ is the RTT of the external network as seen by DAG, whereas $drRTT = T_{dr}^i - T_{dr}^o$ is the RTT of the network plus NIC as seen by the driver timestamps. Hence the in-NIC latency is just the partial RTT $drRTTg = drRTT - gRTT$, whose measured minimum is 14.5 µs.

The cases of key importance for this paper are the in-host components relative to the driver timestamps, namely $cRTTx$ with $c \in \{u, so, b\}$, and $x = dr$. Each of these decomposes into the incoming and outgoing in-host delay components, shown in the figure, as $cRTTdr = d_c^o + d_c^i$, where $d_c^o = T_{dr}^o - T_c^o$ and $d_c^i = T_c^i - T_{dr}^i$.

B. Probe Effect

To quantity the probe effect, we measure the outgoing delay $d_u^o = T_{dr}^o - T_u^o$ across the system using the probes at the user and driver locations, but with no others active. We repeat this with the *so_timestamp* probe active, and then again after activating the *bpf* probe (the extra timestamps are not used). We perform the analogous operation on the incoming side to measure three variants of $d_u^i = T_u^i - T_{dr}^i$, and in each direction we repeat the experiment 3600 times.

Figure 2 gives summary histograms of the three cases in each direction. In each the box shows the inter-quartile range (iqr) containing the median line, with whiskers marking the minimum and maximum values. The increase from 0 up to 2 intermediate probes yields an increase due to the probe effect which is roughly linear, but can vary roughly from 1 to 8 µs.

In all experiments all probes were in place. To first order the *bpf* delays do not require correction as there are no probes between the driver and the *bpf*. The *so_timestamp* and userland delays have 1 and 2 intermediate probes respectively, which the relevant differences in medians shown in the figure were used to correct. We believe the resulting error is bounded by 1 µs in almost all cases.

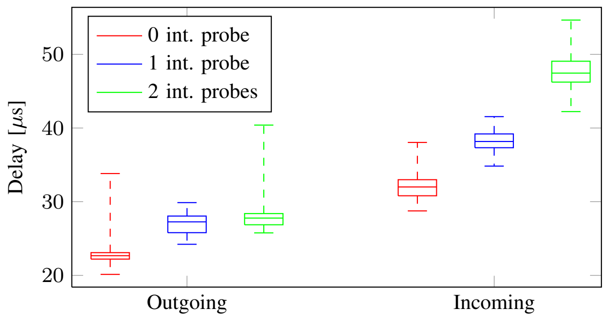

Fig. 2. Boxplots of the outgoing in-host delay (left set) and incoming delay (right set) measured with 0, 1 or 2 intermediate active probes. The increase in median latency is due to the probe effect and is close to linear.

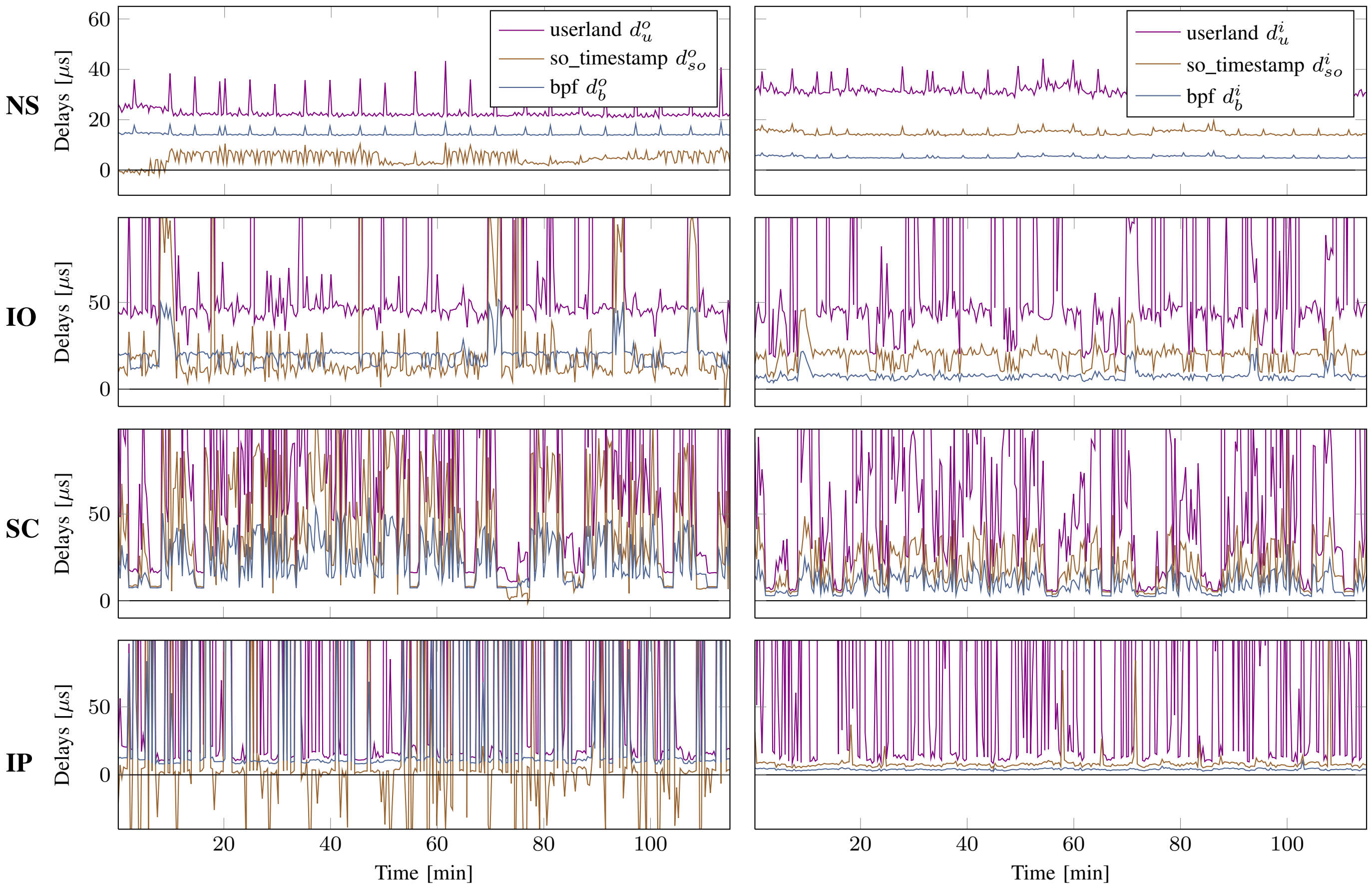

Fig. 3. Outgoing (left column) and Incoming (right column) delay timeseries over 3 hours. Stress scenario from top to bottom: **NS** (no stress), **IO** (disk activity), **SC** (system calls), **IP** (UDP on loopback). In both directions and all scenarios the *bpf* locations are superior. In all scenarios the orders $d_b^o < d_u^o$ and $d_b^i < d_{so}^i < d_u^i$ are respected, but d_{so}^o is not necessarily contained within $[d_b^o, d_u^o]$ and in all scenarios is even seen to take negative values.

IV. Latency Component Analysis

The driver probes are the closest points in the system to the host-NIC interface, which is where we would ideally like timestamp packets. Accordingly, we present results in terms of the distance from this ideal to the alternatives, namely the latencies d_c^o and d_c^i, $c \in \{u, so, b\}$. Other combinations of component latencies can also be studied, for example $d_u^i - d_{so}^i$ yields the latency between the user and *so_timestamp* probes on incoming packets. Note however that components including T_b^o or T_b^i must be interpreted carefully, since the *bpf* subsystem timestamps packet copies, not the original packets. The resulting components can be used to compare alternative timestamps, but not to decompose the path of the original packet from userland to the driver in an additive way. The same caveat hold for the outgoing *so_timestamp* T_{so}^o.

Figure 3 shows time series for the delays d_u^o, d_{so}^o, d_b^o (left) and d_u^i, d_{so}^i, d_b^i (right). The nominal no-stress scenario **NS** is given in the top row. The main observation is that, in each direction, the *bpf* delays are the lowest, and enjoy the lowest variability, followed by *so_timestamp* (incoming only), and then userland. The outgoing *so_timestamp*, d_{so}^o, is a special case. It takes unexpectedly small values which can even be negative, indicating that there is no guarantee that an outgoing *so_timestamp* will be taken before the original packet is actually sent, even under minimal load. The periodicity seen in the outgoing NS plot is, we believe, due to process scheduling effects (its true period is much shorter than it appears in the plotted data, which is sampled).

The remaining rows in Figure 3 give the results under the following stress scenarios:

IO: writes and reads of files of random size to generate high disk activity, plus memory swapping,

SC: system calls resulting in many user/kernel contexts switches, plus memory swapping,

IP: transmission of UDP packets to stress the IP stack (but not the driver, as we select the loopback interface as host)

The stresses were generated using the FreeBSD Foundation's *STRESS2* toolkit [10], consisting of a set of C-programs and configuration scripts that can be used to stress a wide range of kernel functions. Our data is extracted from a continuous experiment with stress pattern: **NS**; **IO**; **NS**; **SC**; **NS**; **IP**; **NS**, each period being 2 hours long. Each stress period consists of back-to-back runs of a stress program two minutes long. The parameters for each run are partially randomised, as are various details of the stress applied within the run. These tools are designed for 'destructive' testing. We use the nominal load levels supplied, but despite the pauses between runs and the partial randomisation of load level which can cause stress to drop for short periods, we observed the resulting stress levels to be very high.

From Figure 3, the conclusions for **NS** continue to hold under each of **IO**, **SC**, and **IP**, though the variability of course has increased dramatically. To better capture the essence of the 24 delay series shown, Figure 4 gives boxplots summarising the histograms arising from the timeseries, reorganised on a per-location basis. Each boxplot shows the minimum, the iqr, and a whisker out to the 90th percentile which in some cases lies beyond the plot's range.

The main observations are as follows.

Direction: outgoing has both larger delays, and higher variability. This is particularly true under **IP** stress, which primarily targets the outgoing side.

Location: As before, *bpf* performs much better than *so_timestamp*, which in turn is much better than userland, in each of the minimum, median, iqr, and outlier senses. As noted earlier, a special case is the outgoing *so_timestamp*, which, although nominally invoked before the *bpf*, actually occurs after an interval which has variability as great as for userland.

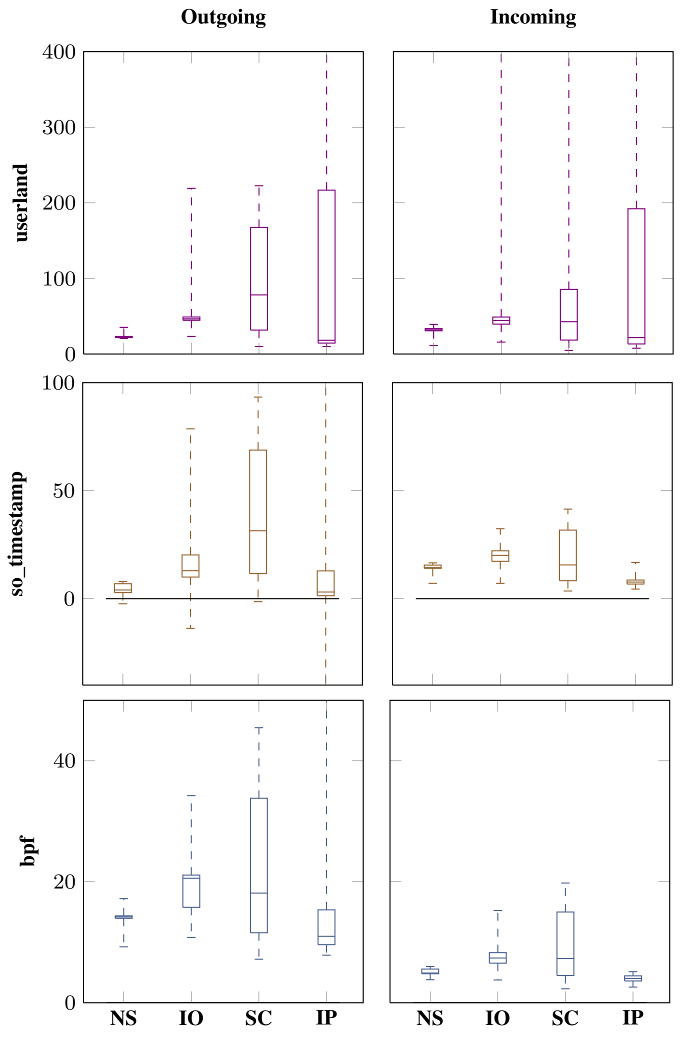

Fig. 4. Boxplots of Outgoing (left column) and Incoming (right column) delay timeseries from Figure 3. Rows group delays according to probe location, from bottom to top: *bpf*, *so_timestamp*, userland. Each plot gives a set of 4 boxplots covering each stress scenario.

Unlike userland however, there is no natural ordering with respect to the driver (or *bpf*) timestamp. In fact under all stress scenarios (including NS) there are negative d^o_{so} values. Under **IP** the minimum delay is -356 µs, i.e. well after most packets have returned! and well below the range of the plot.

Stress: for each location and each direction, **SC** is much more disruptive than **IO** which is considerably worse than **NS**. An important caveat however exists with respect to minimum delays, which actually drop in all cases under **SC** and **IP** compared to **NS**. We believe that this is due, in one form or another, to 'hot cache' type phenomena. The picture is more complex for **IP**, which can be even more disruptive in some respects than **SC** not only for outgoing but for userland, but leaves *bpf* and *so_timestamp* almost unaffected on incoming.

During all experiments the external component $gRTT$ of the RTTs was monitored and found to be stable with a constant minimum value. This independently confirms that the behaviours we observe in the in-host delays, in particular the shifts in minimum values, are indeed due to in-host effects.

V. APPLICATIONS

The immediate application of the results is to point to the benefits of *bpf* based timestamping, the cost of userland timestamping, and the dangers of *so_timestamp* timestamping. If the first of these seems clear, and the second is well known, the third deserves further discussion.

For the timestamping of incoming packets only, for example PTP's Sync messages, the *so_timestamp*, though noisier than hardware or *bpf* timestamping, is a valid option. As we have seen however, the outgoing *so_timestamp* is highly variable, and worse, its error cannot be bounded. More importantly still, the fact that it does not respect causality (packets can, and often do, leave before their timestamp is made), means it cannot even be used to measure RTTs, and fatally, cannot be bounded by them. This is very dangerous for timekeeping as it excludes effective methods for the filtering of delay variability [2]. An alternative approach could be to pair an *so_timestamp* on incoming with a different timestamp on outgoing. This however leads to a higher level of in-host asymmetry, and hence to increased error for clocks based on the timestamps.

The other main application lies in the evaluation of in-host asymmetry A_h. Over LANs this can be a significant component, up to 50%, of the total path asymmetry A. If a system can be 'asymmetry calibrated', then the asymmetry value used by synchronization algorithms can be adjusted from the usual value of zero, reducing clock error by $A_h/2$. We now use our measurements to calculate the in-host asymmetries for the different timestamping locations and to compare them, something which has not been possible in the past to any level of precision.

More precisely, by host asymmetry here we mean the underlying asymmetry based on the in-host delay minima $\underline{\mathrm{d}}^o_c = \min_t d^o_c(t)$ and $\underline{\mathrm{d}}^i_c = \min_t d^i_c(t)$, namely $A_h = \underline{\mathrm{d}}^o_c - \underline{\mathrm{d}}^i_c$ for timestamping location c, and not the variable asymmetries experienced by individual packets. In principle, delay and therefore asymmetry variability can be filtered out, whereas the constant A_h is a fundamental unknown that must be measured

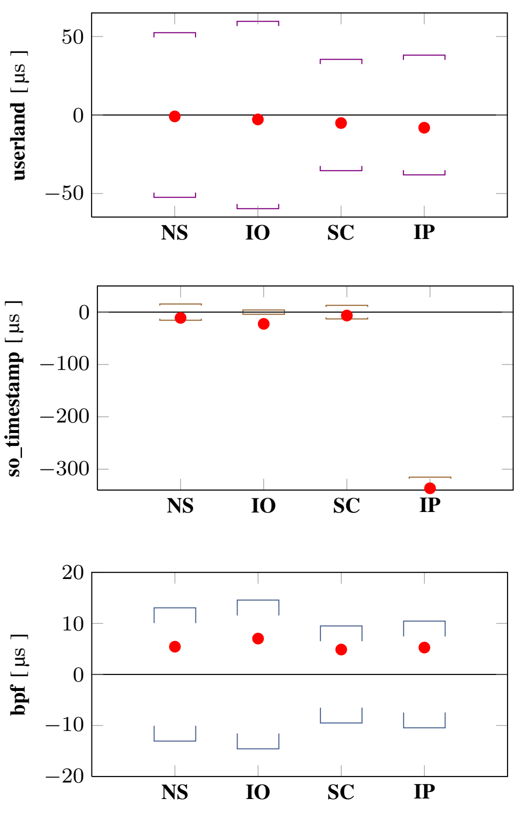

Fig. 5. Asymmetry measurements and minimum RTT bounding intervals based on the minimum delay values from Figure 4. Rows group asymmetries according to probe location, from bottom to top: *bpf*, *so_timestamp*, userland. Each plot gives a set of 4 estimates and bounds covering each stress scenario. The reduction in minimum delays under **SC** and **IP** is clearly seen.

directly via the delay minima. Failing this, all that one can say is that $A_h \in [-r_c, r_c]$, where $r_c = \min_t cRTTdr(t) = \underline{d}^o_c + \underline{d}^i_c$.

Figure 5 gives the asymmetry value and its associated bounding interval $[-r_c, r_c]$ for each of the 12 (location, stress) combinations. For each of *bpf* and userland the asymmetry is very small in absolute terms, as well as in comparison to the conservative bounds in the userland case. Although the userland asymmetry is actually lower than that of *bpf* under both NS and IO, it increases in magnitude with stress, whereas that of *bpf* is largely unaffected at around 4 µs. It would therefore be possible to measure and remove a single load-independent value, to reduce in-host asymmetry to around the 1 µs level. More generally, as the *bpf* sits much lower in the OS, it is less likely to move as kernel versions and hardware evolve. The 'precise balance' we see here at the userland level is not likely to be seen in general.

It is clear that different combinations with mixed timestamping locations would yield asymmetries of much higher magnitude, given the large differences at the delay level. Low asymmetry requires that the two directions be well matched.

In sharp contrast to the other locations, for *so_timestamp* the asymmetry is large compared to the bounds under **NS** and **SC**, does not even fall within the bounds under **IO**, and the bounds themselves fail to include zero (which they should by definition!) in the case of **IP**. These a priori impossible behaviours are a direct consequence of the causality disrespect explained above for d^o_{so}.

VI. Conclusion

We have outlined a methodology, based on DTrace and a sequence of incoming and outgoing test packets, which can be used to perform timing-focussed system profiling in the FreeBSD operating system without the need for kernel modifications. We used it to measure accurately and in detail, for the first time, the characteristics of the 3 timestamping locations used by software clocks (*bpf*, *so_timestamp*, userland), benchmarked against a 4th location (driver) that closely approximates that used by NIC hardware timestamping.

We compared timestamp locations via the measurement of in-host 'delays' relative to the driver location, both nominally and in three stressed environments. The most important results were i) the *bpf* timestamps gave by far the smallest latency and variability, were the least influenced by stress, and had a small and stable asymmetry, ii) for bidirectional-based timekeeping the *so_timestamp* is dangerous, as the timestamp on the outgoing side is not bounded and does not respect 'causality'.

We exploited the availability of the in-host delays to measure the corresponding in-host components of path asymmetry for the first time. The asymmetry of both userland and *bpf* was very low at just a few µs, far smaller that the RTT-based bound, even under stress, and the value for *bpf* was almost independent of stress. These values are unexpectedly small, and we do not expect them to remains so for other platforms such as Linux. Nonetheless, they remain a barrier to the achievement of 1 µs precision level for software clocks over LANs. Precise measurement allows software clocks to correct for the in-host asymmetry and to tighten bounds on total asymmetry.

We point out that *RADclock* [2] uses *bpf* based timestamping, whereas *ptpd* uses *so_timestamp*, and *ntpd* timestamps in userland. *RADclock* packages for Linux and FreeBSD can be found at http://www.synclab.org/radclock/.

VII. Acknowledgements

We gratefully acknowledge the donation of the 3.7GE DAG from Endace, and Symmetricom's loan of the Syncserver 350. This work was supported in part by a grant-in-aid research grant from Symmetricom Inc.

References

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

[2] 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.

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

[4] "Timekeeper software, FSMlabs," http://www.fsmlabs.com/.

[5] Endace, "Endace Measurement Systems. DAG series PCI and PCI-X cards," http://www.endace.com/networkMCards.htm.

[6] B. Gregg and J. Mauro, *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall, 2011.

[7] (2011, Sep.) "Linux Trace Toolkit: Tracing Wiki", DTrace: Kernel Tracing Impact. http://lttng.org/tracingwiki/index.php/DTrace#Kernel_Tracing_Impact.

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

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

[10] "STRESS2, The FreeBSD kernel test suite," http://people.freebsd.org/~pho/stress/index.html.