2021 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS) | 978-1-7281-6484-7/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISPCS49990.2021.9615343

Multi-Core Intra-Process Clock Synchronization

James Coleman
Internet of Things Group
Intel Corp.
Chandler AZ, USA
James.a.coleman@intel.com

Clark Newman
Internet of Things Group
Intel Corp.
Chandler AZ, USA
clark.m.newman@intel.com

Yann-Hang Lee
Department of Computer Science
Arizona State University
Tempe AZ, USA
yhlee@asu.edu

***Abstract*—Real-time applications require not only a precise understanding of now (what time it is), but also the ability to initiate an externally visible event at the correct time. We define this as Intra-process clock synchronization.**

Our findings show that in the latest commercial, off-the-shelf (COTS) multi-core systems running with preempt-RT Linux, the uncertainty in the software visible clock is nearly 5 μs and the precision of user-space software in controlling an externally visible event has a jitter of over 13 μs. By employing emerging hardware mechanisms, the uncertainty can be reduced to just over 200 ns, with the jitter lowered to just 65 ns.

***Index Terms*—Real-time cyber physical systems, clock synchronization, clock disciplining mechanisms, software visible clock, Time Sensitive Networks**

I. Introduction

In addition to deterministic execution, hard real-time systems must perform the required operations at the correct time in terms of a "wall clock," or, the real-world clock to which the process is synchronized [1]. For example, an industrial controller on a factory floor must ensure that the movements of its robotic arm are synchronized with the other tools on the assembly line. The task must not only be guaranteed to execute in a fixed amount of time, but it must also start execution at the *correct* time.

To ensure execution starts at the correct time, the clock used by the scheduler must be synchronized to a "wall clock," which potentially introduces jitter. While the latest instantiations of TSN clock synchronization achieve sub-nanosecond accuracy over the network [2], the question remains as to what level of synchronization accuracy is available to the software visible clock. The software visible clock must be synchronized because it is used not only by the scheduler, but also by the real-time processes. In this work, we provide a detailed evaluation of the limits of clock synchronization in the latest COTS multi-core system running preempt-RT Linux. Using several novel hardware approaches, we demonstrate the improvements in clock accuracy that are currently possible.

We first look at the latency and jitter for the user process to determine what time it is *now*. To quantify this, we measure the latency and jitter of the Linux system call clock_gettime(), which is a relatively low latency call. Additionally, we evaluate the value being returned by clock_gettime() to determine how accurately it is synchronized to a reference clock. This evaluation of clock synchronization accuracy will ultimately be determined by the user processes' ability to generate an externally visible event at the correct time.

For the purpose of this work, our reference clock will be the PTP (Precision Time Protocol) hardware clock (PHC) on a network interface card (NIC). Note that Linux provides the capability to: 1) synchronize the system time returned by clock_ gettime(CLOCK_REALTIME) to a local PTP clock via PHC2SYS; 2) synchronize the local PTP clock to an external PTP clock via PTP4L [3], [4]. Figure 1 illustrates the various clocks synchronized via PTP4L and PHC2SYS.

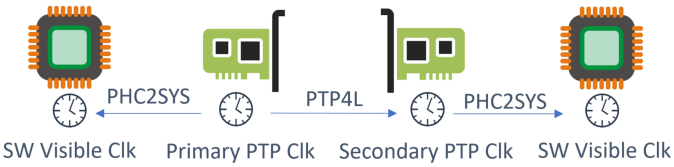

Fig. 1. Synchronization of PTP and Software Visible Clocks in CPS

We will quantify the accuracy of clock synchronization based on the delta from when an externally visible event is generated from the reference clock to when a user process's externally visible event occurs. We will show that this direct observation is necessary to effectively prove synchronization accuracy. However, this reliance on an externally visible event for evaluating the accuracy of synchronization depends both on the software implementation as well as any hardware components utilized by the software. In COTS multi-core systems, it is particularly challenging to achieve accurate clock synchronization due to the CPU's complex micro-architecture and due to resource interference from concurrently running software. For example, when issuing a read to PTP clock on a PCIe-based NIC card, a memory read instruction may experience varying delays accessing memory and PCIe devices because of out-of-order pipeline execution and concurrent traffic.

To begin, we survey the state-of-the-art in Section II. In Section III, we discuss the limits of software-only synchronization accomplished through in-band Memory Mapped IO (MMIO) transactions to a network card. In Section IV, we demonstrate what can be accomplished when using the emerging, dedicated hardware features for clock synchronization. Finally, we conclude our analysis in Section V.

II. Background and Related Work

A. Clock Synchronization Across Network

The prerequisite for any Cyber Physical System (CPS), and the temporal determinism it requires, is a synchronized time base [5]. The precision of the clock synchronization is the limit of the resolution to which temporal determinism can be resolved. Sub-micro second precision can be achieved via the IEEE 1588 Precision Time Protocol (PTP) between different nodes on an Ethernet network [6].

The latest revision of the precision clock protocol for networked systems [7] allows compliant COTS networking hardware to achieve a far greater precision than required by almost any CPS today [8].

As a result, for this work, we treat network clock synchronization as a solved problem due to TSN clock synchronization, specifically IEEE 802.1AS and PTP.

Previous works have evaluated the clock synchronization accuracy for COTS works in practice [4] and have demonstrated that the accuracy achievable over both the network (via PTP4L) and over the SOC (via PHC2SYS) have improved dramatically with the adoption of the latest standards. This work builds upon the methodologies utilized by Libri et al. [9]. However our focus will be on measuring what level of synchronization is possible in a user-space application via PHC2SYS.

B. Timed IO Operations

To perform I/O operations at exact times, the approaches of dedicated, real-time IO processors have been adopted. These IO processors are usually programmable to invoke IO commands at a specified time (e.g., at a future instant) and run with cycle-level timing accuracy. The main goal is to avoid any uncertain latency while CPUs perform IO operations. Examples of real-time IO processors include GPIOCP [10] and TI's Programmable Real-Time Unit [11].

The operating clocks in an IO processor can be synchronized to a CPU's system clock. Given that IO processor is programmed by software, it is dependent upon the software's understanding of now. Even if the IO processor can achieve clock cycle accuracy, the issue of clock synchronization accuracy with respect to the "wall clock" should be considered as an essential subject.

C. Terminology

A *clock counter* is a discrete function that returns a monotonically increasing count of the number of clock cycles or ticks [12]. Since a CPS needs to interact with the real world as well as with other processes, simply counting the clock cycles from reset is not sufficient. Agreeing upon some standard epoch, such as 12:00am January 1, 1970 (Unix epoch), and counting the nanoseconds since this epoch [13] gives each CPS an agreed upon point of reference. The agreed upon epoch among multiple clocks in a CPS is referred to as *clock synchronization*.

Precision is how closely two clocks are synchronized to each other, and *accuracy* is how precisely a clock is synchronized to the real world [14]. A *clock* c_{unix} is a function that returns the number of nanoseconds elapsed from the Unix epoch.

Since c_{unix} is a discrete function returning an integer value, a *clock tick* is the limit of the synchronization precision. When c_{unix} returns n, we only know how many nanoseconds have elapsed since the epoch but have no further resolution. For example, was this sampling 999 ps after the returned nanosecond count, or just 3 ps after? Consequently, the grading of clock synchronization must be done with much greater resolution than that of a clock tick.

Additionally, we must be able to observe the clock's in question. Since the clock driving a modern CPU is generally not directly observable, we must observe a *disciplined* event and measure how closely it occurred to the expected time. In this paper, we choose Memory Mapped IO (MMIO) transactions as the externally visible event, given that such transactions can be easily controlled via software.

D. Hardware and Software Used for Measurements

The following data was collected on the latest generation of COTS hardware from Intel, specifically the Intel® Core™ i7-1185GRE processor. This system was running the Linux 5.4 kernel with the Preempt-RT patch set applied.

III. Intra-Process Clock Synchronization

We define intra-process clock synchronization as a measure of an already running process's ability to execute some instruction at a predefined time. This requires quantifying not only how accurately the process knows what time it is, but also quantifying the temporal determinism of its actions. As discussed earlier, we will use an MMIO read/write transaction as the externally visible event to prove clock synchronization. A modern RTOS, such as Linux Preempt RT, provides a system call clock_gettime(), which allows a process to quickly determine the current time. Linux implements this system call with minimal overhead, resulting in reduced jitter or latency.

However, any software latency and jitter resulting from the call to clock_gettime() will potentially introduce uncertainty in the level of clock synchronization accuracy. Another alternative is to directly rely on the hardware via the read timestamp counter instruction (RDTSC) in x86 architecture, or the PMU cycle counter in ARM architecture, rather than a system call. Table I shows the latency and jitter of each method with the caveat that we are using the RDTSC instructions in all cases to measure the latency and jitter of either a clock_gettime() system call or the latency and jitter of the RDTSC instruction itself.

The data shown in Table I represents 1 million samples, giving us greater confidence that the max observed latency is closer to the max that would be seen in practice. This data does show that the coarse real-time clock can be read faster than the more fine-grained real-time clock. However, the maximum read time for all of the clocks measured (coarse or fine) is over

TABLE I
CLOCK_GETTIME() AVERAGE LATENCY AND OBSERVED MAX LATENCY

Method to determine "now"	Average Latency	Max. Latency
RDTSCP	62.8 ns	182 ns
CLOCK_REALTIME	120 ns	4.44 μs
CLOCK_REALTIME_COARSE	75.2 ns	4.32 μs

4 microseconds, which would imply that a process can never be certain of the current time with a precision greater than 4 μs. Care was taken to not report any system call latency due to the call's being interrupted (by counting the instructions retired for each call and omitting any samples where the number retired was above the expected number). The data does show significantly less jitter in the latency of the RDTSC instruction. This can be seen more clearly in Figure 2, which shows plots of each of the one million reads to the respective methods.

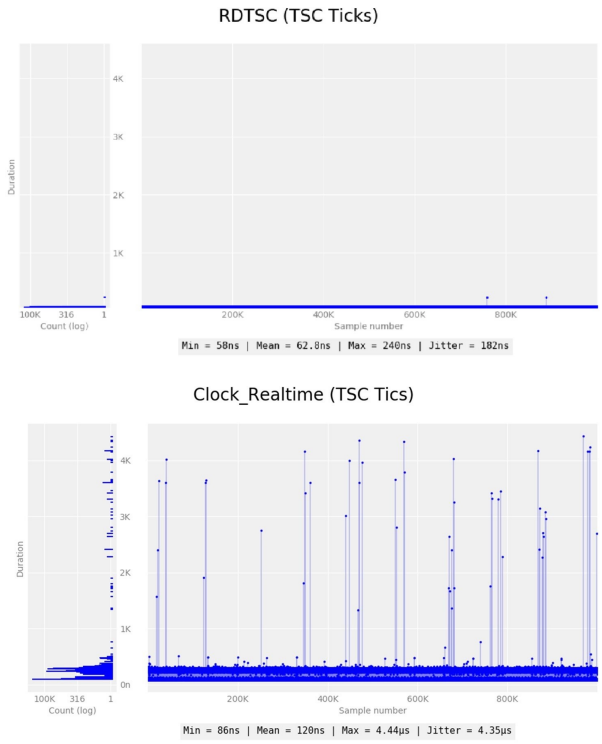

Fig. 2. Clock Read Latency

While the table and plots emphasize the contrast in determinism between the system call and the hardware instruction RDTSC, we must note that RDTSC simply returns the number of unhalted core ticks (at the rate of the max non-turbo frequency) since reset (assuming no manual adjustment of the current value). In contrast, Linux provides the capability to synchronize the system time returned by clock_gettime(CLOCK_REALTIME) to an external time source via the two-level PTP synchronization of PTP4L and PHC2SYS [3]. With both of these services running, the value returned by clock_gettime() should be externally synchronized via PTP's running on the network.

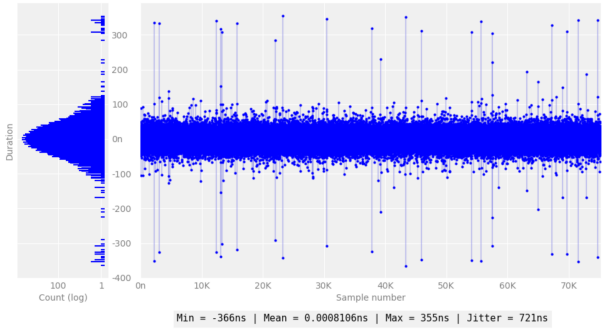

Fig. 3. PHC2SYS Offset Adjustment

The PHC2SYS service can report the instantaneous offset adjustment, i.e., a measure of how much the system time is being adjusted to follow the PTP time from the network controller. If the primary clock is not jumping (i.e., is strictly monotonic, which is our case), then this shows both how far the two clocks have drifted as well as any noise or error introduced through a lack of precision of the synchronization itself. Because of this, the smaller the offset adjustment, the better. Likewise, a larger offset adjustment that is wildly swinging over and under the the actual value represents a lack of accuracy in the clock synchronization.

Based on the PHC2SYS offset adjustment, the maximum jitter between system clock and PTP clock is only 721 nanoseconds in our experimental setting, as shown in Figure 3. Together with the jitter of clock_gettime() system call, the expected software jitter based on the synchronized PTP clock is around 5 μs.

To quantify the level of intra-process clock synchronization, the following approach is employed: First, software records what time it is now from the software visible clock (clock_gettime() or RDTSC), which is disciplined to an external PTP clock source (via PHC2SYS or manual correlation). Immediately after determining the current time, software generates an externally visible signal (MMIO transaction). Finally, the external device records when the signal (MMIO transaction) was observed in terms of the primary PTP clock.

Due to the serial execution of steps one and two above, we insert an additional step in between them. This allows software to calculate a virtual now between the before and after instants in an attempt to more closely match the recorded now to when the MMIO transaction was actually generated and externally visible. Additionally, the software will serialize the CPU pipeline and flush any pending stores before and after the MMIO transaction to again ensure that the virtual now corresponds as closely as possible to when the externally visible event actually occurs.

We acknowledge that, in addition to being dependent on the accuracy of clock synchronization, the above outlined methodology is also dependent on both software execution jitter as well as MMIO transaction propagation jtter through the System on a Chip (SOC). These shortcomings will be addressed later on in this work.

This approach is summarized in Listing 1 pseudo code

```
SFENCE
t0 = clock_gettime()
MMIOWr_2_latch_PTP_clk()
SFENCE
t1 = clock_gettime()

virtual_now = ((t1 - t0) / 2 ) + t0
now_at_PTP_Latch = ReadPTPLatchRegister()

plot(virtual_now - now_at_PTP_Latch)
```

Listing 1. Pseudo Code for Correlating clock_gettime() to NIC PTP Time

for correlating clock_gettime to PTP. The difference between the time that software determined via the system call clock_gettime() and the time the MMIO write latched the PTP clock on the NIC is plotted in Figure 4.

Two important observations from the data presented in Figure 4 are: 1) the absolute jitter of 13.7 μs, which is more than the combined jitters from the clock_gettime() call and the reported PHC2SYS offset adjustment, and 2) the lack symmetry in the plot. These two observations further highlight the likelihood that the desired clock synchronization accuracy measurement is being obscured due to software/MMIO transaction jitter.

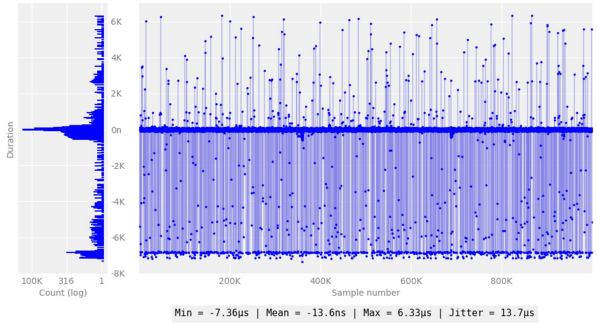

Fig. 4. Intra-Process Clock Synchronization via clock_gettime() with PHC2SYS

Before we employ emerging hardware features to try to minimize the software and MMIO transaction propagation jitter, we first replace the system calls clock_gettime in Listing 1 with the RDTSC instruction. Using the revised implementation of the pseudo code in Listing 1, the plot in Figure 5 is produced again for 1 million samples. While this change does improve the symmetry of the plot, the absolute jitter worsens.

IV. Emerging Hardware Features for Clock Synchronization

The data presented in section III shows the limits of a user-space process, not only in determining what time it is now, but ultimately in generating an externally observable event at the appropriate time. Our methodology is restricted by the jitter in external event generation. This jitter reduces the clock

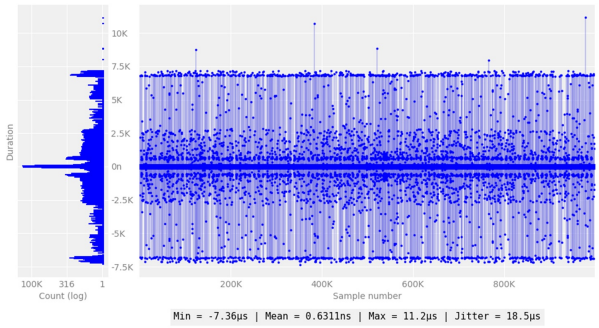

Fig. 5. Intra-Process Clock Synchronization via RDTSC

synchronization accuracy. Because of this we will employ emerging hardware features in an attempt to improve both the accuracy of the software visible clock's synchronization as well as software's ability to generate an external visible event at a predefined time.

A. SOC Clock Synchronization Invariant TSC/ART

First, we focus on improving the clock synchronization accuracy between the software visible clock (i.e., clock_gettime()). Again, we use the PHC2SYS offset adjustment as a proxy for how closely the software clock is synchronized to the reference clock.

With the inclusion of the invariant timestamp counter (TSC) [15], variations due to CPU frequency changes are eliminated. Additionally, the analog factors of spread spectrum clocking and other techniques for reducing RF radiation are removed in favor of a more precise concept of time for software running on COTS hardware. This is extended to the entire SOC via the inclusion of the Always Running Timer (ART) [16]. With this new capability, the software-visible, per-core timestamp counter is synchronized to the SOC's ART. In turn, peripherals synchronized with the ART are, by inheritance, synchronized with the TSC, including the PTP clock on the integrated Ethernet controller. This allows the PHC2SYS to read the hardware-latched cross timestamps and determine the adjustment needed for the system clock with respect to the PTP clock. This approach eliminates the impact on synchronization accuracy due to MMIO transaction propagation jitter. Figure 6 depicts the relationship between the three clocks, with RDTSC being a fixed multiple of ART, and PTP and ART being correlated via hardware cross-time stamping.

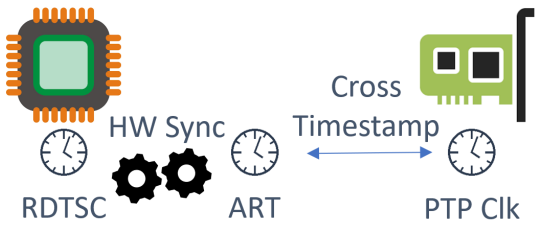

Fig. 6. Clock Synchronization Among TSC, ART, and PHC

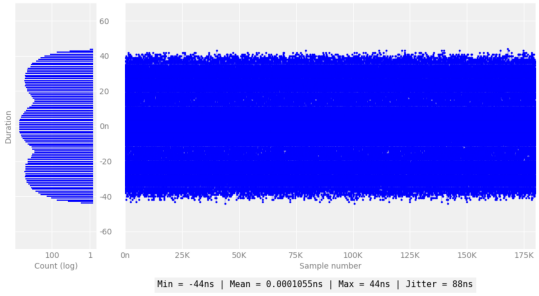

Fig. 7. ART-Based TSC PHC2SYS Offset Adjustment for Primary PTP Clock

This enhanced precision is demonstrated in practice by looking at the PHC2SYS offset adjustment between the PTP clock on the integrated Ethernet controller (acting as the primary clock) and the operating system's clock, as shown in Figure 7. From this figure, we can see a perfectly symmetrical +/- 44 nanoseconds of adjustment. The ART frequency can be discovered by having software read CPUID[15H].ECX, which, for the platform used, is 38.4 MHz, resulting in a 26 ns period. Due to differences between the resolution of the ART clock (26 ns per tick) and the PTP clock (1 ns per tick), it is quite impressive that hardware is able to keep the two clocks in sync by +/- 2 ART ticks, especially when we compare Figure 7 to Figure 3.

B. Clock Synchronization with TGPIO

In the previous section, we looked at PHC2SYS offset as a proxy for clock synchronization accuracy. While the results are impressive, they do not actually quantify the synchronization accuracy as observed by an external event. To address this, we will now rely on an oscilloscope to observe any jitter between a reference event and a disciplined event controlled by user-space software as shown in Figure 8. While the reference event is based on the NIC's PTP clock, the disciplined event is generated at a Timed General Purpose IO (TGPIO) pin, which is based on the software visible clock. TGPIO pins can be found in the latest generation of Intel COTS systems [17]. ART is used as the reference clock for TGPIO pins, which can be programmed to generate an edge at a specific future ART value.

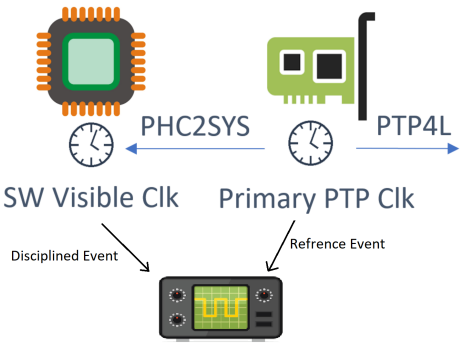

Fig. 8. Oscilloscope-Based Observation Points and Clock Inheritance

```
do
   MFENCE
   t0 = RDTSC
   PTP_Time_Now = MMIORd_PTP_clk()
   MFENCE
   t1 = RDTSC
   MMIO_Rd_Latency = t1 - t0
while (MMIO_Rd_Latency > Filter_Threshold)

Virtual_Now = ((t1 - t0) / 2 ) + t0
NextEvent = Delta_To_Next_Event(PTP_Time_Now)
program_TGPIO_Edge(Virtual_Now + NextEvent)
```

Listing 2. Code for Correlating TSC to PTP Via MMIO Read & TGPIO

For the following tests, software will both attempt to 1) determine what time it is now and 2) attempt to generate an externally visible signal at a predefined future time. To eliminate the jitter due to software execution and MMIO propagation, we will program the TGPIO pin to generate the externally visible event at the predefined time. While the use of TGPIO could potentially eliminate the jitter due to the MMIO transaction propagation, it would not eliminate the software jitter for determining what time it is (previously found to be over 5 μs). To address this source of software jitter, we will use filtered MMIO reads directly to the PTP register. The filtering is accomplished by using RDTSC to measure how long the MMIO read took and discarding slow MMIO reads. The pseudo code for this methodology is documented in Listing 2. The function *Delta_To_Next_Event* knows the rate of the external event being generated by the reference clock and returns the expected TSC ticks until the next reference event from the PTP time provided.

A NIC, in our case the i225, has its Software Defined Pin (SDP) configured as output and will generate an edge at a predefined rate based on the current value of PTP. We again read the PTP time off the NIC, bounded by RDTSC, and then program the TGPIO to generate an edge aligned with software's understanding of PTP time. Slow TGPIO reads are filtered, thus eliminating the jitter due to software execution and MMIO transaction propagation. This shows the limits of the actual clock synchronization between the CPU and an external NIC.

Although we test for 1 million iterations, our use of an oscilloscope prevents software from collecting each individual data point. As a result, we rely on the oscilloscope's own analytic capabilities to report the jitter and standard deviation between the two signals, which is just 313 nanoseconds. While significantly better than the 14 + microsecond jitter initially reported in Figures 4 and 5, this does fall short of the invariant TSC results reported earlier in Figure 7. However, the PHC2SYS offset is determined by hardware cross-timestamps and not by MMIO reads. In the following section we will employ cross-timestamping rather than MMIO reads.

C. Full TGPIO Synchronization

In the previous section, we demonstrated that a software-only approach (filtered MMIO Read) can actually synchronize a clock within nearly 300 nanoseconds by eliminating the

software jitter (including the MMIO transaction propagation jitter through the SOC). Despite filtering slow MMIO reads, we are unable to demonstrate the 88 ns of jitter expected from the reported PHC2SYS offset adjustment. We will now employ the hardware cross-timestamping in an attempt to demonstrate the 88 ns of clock synchronization accuracy expected.

In addition to generating an edge at a predefined ART value, the TGPIO can also latch the ART value when it detects an edge. This eliminates the latency and jitter of the software execution when the CPU issues an MMIO read to the PTP register. Again, we use the SDP of the i225 configured as output to generate an edge at a predefined rate based on the current value of PTP. This SDP event triggers TGPIO to latch the current ART value. Software periodically queries the TGPIO latch register and correlates the latched ART value with the PTP value corresponding to the i225's SDP event. Software then programs a second TGPIO pin to generate an edge in the future as close as possible to the i225's periodic SDP edge.

Again, an oscilloscope captures the i225's SDP edge (the reference clock) as well as the TGPIO edge (the disciplined clock). The variation between the two edges is the clock synchronization accuracy for the full TGPIO synchronization method.

TABLE II
CLOCK SYNCHRONIZATION ACCURACY

Clock	Jitter	StdDev
clock_gettime() & MMIO Wr	13.7 μs	204 ns
RDTSC & MMIO Wr	18.5 μs	262 ns
TGPIO Disciplined (via MMIO read)	313 ns	21 ns
TGPIO Disciplined (via TGPIO latch)	65 ns	16 ns

The data shown in Table II summarizes the different clock synchronization approaches covered in this work. The data in the final row is note worthy. It shows that hardware cross-timestamping achieves an accuracy higher than what was reported by the PHC2SYS offset adjustment (65 ns vs. 88 ns), nearly a one ART tick improvement.

V. CONCLUSION

This work has outlined a methodology for quantifying a CPS' ability to support a precise and accurate understanding of *now* to a user-space application, including quantifying the user-space application's ability to do something *now*. This work suggests that utilizing the emerging COTS hardware for clock synchronization achieves an accuracy that exceeds the determinism delivered by the COTS software to user-space applications (i.e., clock_gettime()). The challenges in the COTS space are not limited to software alone. As this work observed, jitter in MMIO transaction propagation highlights the limits of achieving deterministic data movement into and out of the COTS SOC.

In addition to the emerging technologies covered by this work (Invariant TSC, and Timed GPIO), a particularly important mechanism to be considered is the PCIe Precision Time Measurement (PTM). While the Full-Timed GPIO methodology yielded the best results presented in this work, it does require additional sideband signals for clock synchronization. To reduce the cost and complexity of additional wires, a COTS system designer could utilize the in-band synchronization mechanism, PCIe PTM, to synchronize the clocks. On Intel COTS systems, the PCIe root complex cross-timestamps PTM messages with ART. Currently, driver support does not exist for the PTM hardware functionality on the i225, and, consequently, this investigation is reserved for future work.

REFERENCES

[1] D. Broman, P. Derler, and J. Eidson, "Temporal issues in cyber-physical systems," *Journal of the Indian Institute of Science*, vol. 93, no. 3, pp. 389–402, 2013.

[2] IEEE, "IEEE standard for a precision clock synchronization protocol for networked measurement and control systems," *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pp. 1–300, 2008.

[3] R. Cochran, "The Linux PTP Project." 2020. [Online]. Available: http://linuxptp.sourceforge.net/

[4] R. Cochran, C. Marinescu, and C. Riesch, "Synchronizing the linux system time to a ptp hardware clock," in *2011 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, 2011, pp. 87–92.

[5] E. Lee, B. David, T. Martin, and S. S. Sunder, "Cyber physical systems," in *Proc. of the IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, 2008.

[6] H. Weibel, "High precision clock synchronization according to ieee 1588 implementation and performance issues," *Proc. Embedded World*, vol. 2005, p. 96, 2005.

[7] IEEE IM/ST - TC9 - Sensor Technology, "IEEE draft standard for a precision clock synchronization protocol for networked measurement and control systems," *IEEE P1588/D1.6, August 2019*, pp. 1–535, 2019.

[8] M. Lipinski, T. Wlostowski, J. Serrano, P. Alvarez, J. David Gonzalez Cobas, A. Rubini, and P. Moreira, "Performance results of the first White Rabbit installation for CNGS time transfer," in *2012 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication Proceedings*, 2012, pp. 1–6.

[9] A. Libri, A. Bartolini, M. Magno, and L. Benini, "Evaluation of synchronization protocols for fine-grain hpc sensor data time-stamping and collection," in *2016 International Conference on High Performance Computing Simulation (HPCS)*, 2016, pp. 818–825.

[10] Z. Jiang and N. C. Audsley, "GPIOCP: Timing-accurate general purpose i/o controller for many-core real-time systems," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, 2017, pp. 806–811.

[11] R. Birkett, "Enhancing real-time capabilities with the PRU." 2014. [Online]. Available: https://elinux.org/images/1/1c/Birkett–enhancing_rt_capabilities_with_the_pru.pdf

[12] F. B. Schneider, "Understanding protocols for byzantine clock synchronization," Tech. Rep. TR87-859,, Tech. Rep., 1987.

[13] V. Singh and P. Chaudhary, "Y2k38: The bug," *International Journal of Engineering and Advanced Technology (IJEAT)*, vol. 2, 2012.

[14] D. L. Mills, "Precision synchronization of computer network clocks," *ACM SIGCOMM Computer Communication Review*, vol. 24, no. 2, pp. 28–43, 1994.

[15] Intel, "Intel® 64 and IA-32 architectures software developer's manual," *Volume 3B: System programming Guide, Part 2*, p. 153, 2016.

[16] J. Coleman, S. Almalih, A. Slota, and Y.-H. Lee, "Emerging cots architecture support for real-time tsn ethernet," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 258–265.

[17] Intel, "Intel® 400 series chipset family onpackage platform controller hub datasheet," *Volume 1*, p. 119, 2019.