Graham: Synchronizing Clocks by Leveraging Local Clock Properties

Ali Najafi, *Meta;* Michael Wei, *VMware Research*

https://www.usenix.org/conference/nsdi22/presentation/najafi

This paper is included in the Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation.

April 4–6, 2022 • Renton, WA, USA

978-1-939133-27-4

Open access to the Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Graham: Synchronizing Clocks by Leveraging Local Clock Properties

Ali Najafi
Meta†

Michael Wei
VMware Research

†*Work done while at VMware*

Abstract

High performance, strongly consistent applications are beginning to require scalable sub-microsecond clock synchronization. State-of-the-art clock synchronization focuses on improving accuracy or frequency of synchronization, ignoring the properties of the local clock: lost of connectivity to the remote clock means synchronization failure.

Our system, Graham, leverages the fact that the local clock still keeps time even when connectivity is lost and builds a failure model using the characteristics of the local clock and the desired synchronization accuracy. Graham characterizes the local clock using commodity sensors present in nearly every server and leverages this data to further improve clock accuracy, increasing the tolerance of Graham to failures. Graham reduces the clock drift of a commodity server by up to 2000×, reducing the maximum assumed drift in most situations from 200ppm to 100ppb.

1 Introduction

The ever increasing performance demands of strongly consistent distributed applications has driven a desire for tightly synchronized clocks. Instead of communicating over the network, servers can establish an order over messages using a timestamp from a local clock [7, 18, 30]. Leveraging synchronized clocks has become more pervasive as applications require tighter latencies that approach the latency of the network itself. However, deploying finely synchronized clocks at scale remains a significant challenge, often requiring the use of specialized hardware [20, 22],.

In an ideal system, synchronizing clocks would be a trivial task. Clocks would never *drift* (lose or gain time) and a synchronized clock would stay synchronized forever. In real systems, however, clocks drift, so synchronization needs to be done frequently to keep clocks in time. Spanner [7], for example, assumes 200ppm drift, which translates into 200μs/s, a second roughly every hour, or a minute every 4 days. This drift increases clock uncertainty (ε) and limits the performance of applications leveraging clocks, which must wait out the uncertainty. State-of-the-art systems today assume high clock drift and focus on increasing synchronization precision and frequency, with specialized hardware performing as many as 10K synchronizations per second to achieve sub-microsecond clock synchronization [17, 20, 22, 28]. Furthermore, systems assume missed synchronizations result in loss of synchronization, resulting in potentially unnecessary shutdowns due to clock uncertainty exceeding application requirements [20].

The clocks which drive the processor and timestamping hardware, however, are required to drift far less than these systems expect: datasheets from several vendors specify a clock crystal with at least ±20ppm temperature stability [9, 26]. An unstable clock could cause the system to violate the tight timing requirements required by the processor, memory and I/O subsystem. Local clocks can be much more stable than most systems assume. If we know that a system has lower drift, we can reduce the rate of synchronization, tolerate synchornization failures, reduce network congestion and the overhead of processing synchronization messages, as well as avoid the use of specialized hardware [4].

In this paper, we describe Graham[1], a system which models the stability of the local clock to determine the required synchronization rate. Graham leverages sensors available in every commodity server to characterize the clock against an accurate reference clock, such as GPS, PTP or even NTP. Graham uses this characterization to build a synchronization model, which determines how frequently the system must be synchronized and how many synchronization failures can be tolerated, and can achieve below 1ppm drift. In the servers we tested, we were able to achieve 100ppb stability in most cases, which is over 2000× better than the max drift rate assumed by Spanner. The guiding principle behind Graham is to improve the clock in software without adding additional hardware. This approach is challenging because existing sensors are not designed to characterize clocks, and are located at varying

[1] Named after George Graham (1673–1751), a clockmaker who improved the pendulum clock's accuracy by compensating for changes in pendulum length due to temperature.

distances away from the oscillators that drive system clocks. To address this challenge, Graham characterizes the system by observing the effect of temperature fluctuations at various sensors on clock error between synchronizations.

The contributions of this paper are:

- We debunk the myth that commodity computers have unstable clocks.
- We describe how to automatically characterize computer clocks using commodity sensors.
- We show that this characterization can be used to greatly reduce synchronization rates, resulting in 100ppb stability without specialized hardware.

2 Clock Generation and Synchronization

The term *clock* is often used for several related concepts. In this paper, we will use *clock* to mean a counter which is incremented at some frequency and can be used to measure time. Clocks are driven by *clock signals*, which oscillate between low and high logical states. A clock driven by this signal increments on a *clock edge*, which is the transition between two logical states (typically, the rising edge is used). Clock signals are provided by *clock sources*, which are typically quartz crystal oscillators in modern computers. In this section, we provide background on how clocks are generated and synchronized in a typical computer system.

2.1 A typical Linux Intel x86 clock system

Clock systems are architecture and vendor dependent, so we focus on a typical Intel Linux x86 machine as a model clock system. An Intel x86 system consists of multiple clocks which are driven by multiple clock sources. Some of the clocks accessible to users include the timestamp counter (TSC), real-time clock (RTC) and the precision time protocol clock (PTP). Each of these clocks run at different frequencies and serves different purposes, and which clock software ultimately can access has been shown to vary [23].

For the purposes of this paper, we center on the TSC, the clock typically accessed by applications via `clock_gettime(2)`. This clock is driven by a clock signal known as `BCLK` (typically 100MHz). The `BCLK` is driven by a phase-locked loop (PLL) which multiplies the frequency of a quartz crystal (48MHz on C620 ICC [9]). The `BCLK` is an important signal which not only drives the logic in the processor, but the memory controller and other components, depending on the processor model. Adjusting the `BCLK` is often done when overclocking by changing PLL parameters, but large adjustments can result in system instability and lockup.

So far, we have described how Linux enables applications to read a clock. In order to be able to compare one clock to another, clocks must be synchronized. Most Linux distributions rely on `ntpd` [24] or `chrony` [5] to synchronize local clocks to a remote server with a reference clock synchronized to wall clock time (UTC) via a time source such as GPS using the NTP protocol. The NTP protocol estimates the network delay between the server to client by dividing the round-trip delay in half and can achieve on the order of 1ms-100ms time synchronization error, with error increasing as the delay becomes more asymmetrical. In addition, since NTP is run in software, synchronization is subject to software jitter such as scheduling and interrupt handling which prevents NTP accuracy below 0.5ms, even in ideal conditions.

To achieve sub-microsecond accuracy, PTP (IEEE 1588) reduces software jitter [10]. First, instead of acting as a service where clients request the time, a PTP server continuously broadcasts the current time at periodic intervals. Clients estimate the network delay by sending a special message to the server to compute the round trip time and dividing that time in half. Finally, PTP introduces a new hardware clock located on the network card itself. This clock is driven by a different quartz crystal at the network card, usually corresponding to the frequency needed to drive the card's transceivers (25MHz for 10Gb Ethernet). The network card can capture the synchronization packets as they arrive to synchronize the PTP clock to the server, eliminating the inaccuracy introduced by software jitter. In Linux, `phc2sys` synchronizes the TSC clock to the PTP clock.

The accuracy of PTP is dependent on accurate delay estimation. Recognizing this limitation, Huygens [14] and Tick Tock [6] use coded probes and support vector machines to filter out queued packets from round trip delay estimation. Some commercial PTP implementations use packet delay variation (PDV) filters [27], and compensate for known latencies in the receive and transmit paths.

Because of clock drift, synchronization frequency is also important. While most of the latency sensitive paths of PTP are in hardware, it is still software driven, limiting the frequency of PTP synchronization, especially when filters are used that necessarily discard some synchronization data. This can be problematic if clock synchronization requirements are tight and clock drift is high. For instance, Huygens has a default sync interval of 2 seconds. A clock with 200ppm (0.02%) of drift will accumulate up to 400μs of drift between missed synchronizations. If there is a single transient synchronization failure resulting in a 4 second interval, up to 800μs of drift would accumulate, which would be problematic if an application required sub-microsecond clock accuracy. To increase synchronization frequency, most solutions require specialized hardware. For example, DTP modifies the Ethernet physical layer to exchange messages at the frequency of microseconds while reducing network delay nondeterminism [17]. Sundial leverages specialized hardware that synchronizes every 100μs and performs fast failure detection to notify software to recover by finding a backup clock [20].

2.2 Holdover Time

Notably, current state-of-the-art systems do not attempt to characterize the *holdover time* of the clock, which refers to the amount of time a clock can remain accurate without a synchronization. For instance, Spanner and Sundial both assume a static 200ppm maximum drift. If the maximum time uncertainty bound (ε) is 1μs, then a clock with 200ppm drift (200μs/s) will only be able to holdover the clock for 5ms without synchronization before *potentially* exceeding ε. The formula for holdover time can be given as:

$$t_h = \frac{\varepsilon}{\mathrm{d}f} \tag{1}$$

Where t_h is the holdover time, ε is the maximum time uncertainty, and $\mathrm{d}f$ is the clock drift. 200ppm, however, is very conservative: most quartz crystals used for computer systems are specified on the order of 100ppm of error, and only when operated under extreme operating conditions. In the next section, we describe how we can characterize oscillator error, and use this characterization to increase the holdover time.

2.3 Characterizing Oscillator Error

Oscillators provide the clock signals that ultimately drive the clocks used in computer systems, and are the source of most clock error. The most common oscillator in use in nearly all computers today is a quartz crystal, which uses the pizeoelectric properties of quartz to produce a clock signal at a given frequency.

Quartz is cut to resonate at a given frequency, however, as the cut is a mechanical process, tolerances in the cut process may result in a resonant frequency which is slightly offset from the advertised frequency, known as the frequency tolerance. Since any error in the cut is usually fixed, this tolerance results in a fixed offset from the advertised frequency. In typical computer crystals, this error is usually in the 50ppm range. Lower tolerances require more accurate (e.g., fine laser) cuts and are significantly more expensive.

Quartz crystals also age over time as mechanical devices which are constantly vibrating, slowly deviating from their advertised frequency. This error is usually small (5ppm/year) [1], and also results in a slight frequency offset.

So far, we have described sources of quartz crystal oscillator error which are relatively constant. As physical devices, the frequency of quartz crystals are also affected by environmental changes. The most prominent factor is temperature [36], which can result in a significant change in frequency over the crystal's operating temperature range. While temperature can induce variations in the frequency of the crystal, the temperature-frequency response of crystals are quite deterministic: in fact, some crystal manufacturers produce the response curve on the crystal datasheet. Typical crystals produce anywhere from a 30ppm-100ppm change in frequency over their operating temperature ranges [1].

Table 1: Frequency Error in Standard Quartz Oscillators

Name	Typical Range	Typical
Tolerance	±50 ppm	-
Aging	±5 ppm/year	-
Temperature	±1 ppm/°C	±15 ppm (25-40°C)
Voltage	±1 ppm/V	±0.1 ppm (±2% 3.3V)
Load	±0.1 ppm/pF	±0.1 ppm (±10% 15pF)
Acceleration	0.1 ppb/G	0 @ Rest
Time Dilation	0.1 ppq/m	0 @ Sea Level

In addition to temperature, a variety of other environmental factors will affect the frequency of the oscillator. However, these factors contribute a relatively small amount of frequency error compared to temperature. Changes in supply voltage usually result in a 0.1ppm-5ppb change in frequency. Another factor is variation in the load capacitance: in order for the crystal to resonate at the expected frequency, the correct amount of capacitance is required. Since the capacitors used to provide the load capacitance also have tolerances, the capacitance can vary depending on the properties of the capacitors used. Typically, load capacitance error is specified at 0.1ppm-5ppb [29, 33]. The frequency of quartz crystals are also sensitive to acceleration, depending on the axis it is applied to. For ordinary quartz crystals, this is typically in the range of 0.1-10ppb/G [29, 33]. For a 500G shock, such as that specified in MIL-STD-883H, representative of a device dropping to the floor, frequency error could be as high as 1ppm [19, 33]. Note that the recommendation for operational vibration and shock limits in datacenters is less than 5G [16] which is well below 500G. Finally, crystals are even sensitive to relativity: a crystal closer to the gravitational field of the earth will have a lower frequency than a crystal further away, such as on a mountain or in space. This error is around 0.1ppq/m from sea level, or $\approx$ 0.9ppt at the top of Mount Everest or $\approx$ 3ppb from geostationary orbit [33].

These sources of error are a result of the physical properties of quartz, and the data collected in Table 1 are collected from the datasheets of various quartz oscillators used in servers [1, 19, 29, 33, 36].

2.4 Debunking the Myth of Unstable Clocks

As we have seen, most of the frequency error in a quartz oscillator is either relatively static or dependent on temperature. Voltage and load only contribute a small amount of error and should be within small tolerances (otherwise, other parts of the system may begin failing). Servers in most datacenters are stationary, so the effects of acceleration and time dilation should be constant.

Static error can be easily corrected if it can be learned: if we learn that our crystal resonates at 32.769 KHz instead of 32.768 KHz, we simply need to adjust our accounting of time,

perhaps by using 32769 as a divider instead of 2^{15}. If our synchronization error is minimal and we keep the temperature constant, we can learn this value over several synchronization passes. NTPd and chrony both try to learn the static drift using the *driftfile*.

Most state-of-the-art systems, however, combine static and dynamic error in their uncertainty calculations, resulting in the assumption of an unstable clock. For instance, Sundial assumes that the clock error of their oscillator is 100ppm, but this number includes the static tolerance error from the cut, which is easily learned. Moreover, even if they had chosen a $\pm$100ppm temperature tolerance crystal, this shift would be over the entire operating range, as in a shift from -30°C to 85°C. An overheating server moving from 60°C to 80°C would experience only about 20ppm change in drift from temperature, an order of magnitude less than the conservative 200ppm error used in spanner.

In practice, most crystals used to generate processor clocks have temperature tolerances in the range of $\pm$20ppm. Intel Chipset Integrated Clock Controllers (ICC), for example, specify "Total of crystal cut accuracy, frequency variations due to temperature, parasitics, load capacitance variations is recommended to be less than 90ppm" [9], and external clock generators such as the common CK420BQ used in Intel systems specify a cut tolerance of $\pm$20ppm and a temperature tolerance of $\pm$20ppm over the entire operating range [26]. If we can filter out the static error, we will be left with 20ppm temperature error. Then this clock will have a 1 μs holdover time of 50ms, a 10$\times$ improvement over the 200ppm assumption.

2.5 Software Temperature Compensation

Once we have corrected the static frequency error, temperature remains as the dominant source of frequency error. This effect is well known, and software compensation techniques are described in the literature [13, 15, 25]. In computers, chrony can correct for temperature errors given the temperature-frequency relationship and a temperature sensor. In wireless networks, where minimizing clock error is critical, environment and temperature aware compensation are used [34, 35].

While temperature-frequency curves are sometimes published on the datasheet of a crystal, using them to correct errors on a commodity computer system requires knowing the crystal used. This can be difficult even for an expert given the small markings on most crystal packages. Moreover, the crystal used can be different even across the same model of motherboards, since manufacturers may substitute functionally equivalent parts due to cost or supply-chain reasons. Unless the system was purpose built with temperature correction in mind, temperature sensors are likely located some distance away from the crystal. Therefore, selecting the right temperature sensor may be a challenge. However, correcting for temperature error can effectively reduce the frequency error of the crystal to less than 1ppm, resulting in a 1 μs holdover time of 1s, a 200 $\times$ improvement over Spanner's assumption.

2.6 Other Oscillators

Many applications outside of general-purpose computing, such as wireless require low frequency error over a wide temperature range. The temperature compensated crystal oscillator (TCXO) consists of quartz crystal with a temperature compensation circuit and reduces the effect of temperature to $\approx \pm$1ppm of error. The oven compensated crystal oscillator (OCXO) takes temperature control one step further and places the crystal in a miniature oven which keeps the crystal at a constant temperature, reducing temperature effects to $\approx \pm$1ppb. This oven can be doubled (DOCXO) to achieve $\approx \pm$0.1ppb of temperature error. Atomic oscillators, which work based on electron transitions, can provide even more stability: rubidium oscillators provide up to 0.0002ppb/s. The cost of these oscillators is often cited as prohibitive, but can be quite inexpensive, relative to specialized hardware. For instance a 48MHz TCXO at 0.5ppm suitable for driving an Intel ICC costs around USD $2 [11], and a 25MHz OCXO at 10ppb suitable for driving a CK420BQ clock synthesizer costs around USD $70 [2].

While replacing the oscillators in computer systems might be an option in new, future hardware, it is an invasive and expensive procedure for existing hardware. The focus of Graham is to democratize accurate clocks using only existing hardware. Using software techniques, we can achieve low error without adding additional hardware.

3 Clocks and Sensors In Servers

In order to understand how temperature sensors can be used to estimate clock error in commodity systems, we studied the sensor and time configuration of a variety of platforms. One unexpected challenge was the difficulty of accurately measuring clock error.

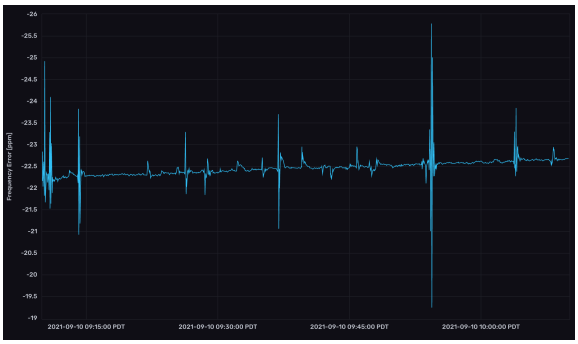
Clocks. The Linux pulse-per-second (PPS) [21] facility provides a mechanism for delivering an accurate reference time. PPS devices are devices that accurately emit a low-jitter pulse every second. A PPS driver calls the `pps_event` API whenever the pulse is received, and the kernel records the timestamp associated with that pulse. Typically, this pulse is a signal that causes an interrupt, and the PPS API is called by an interrupt service routine (ISR). However, even when using very low jitter PPS devices, such as the ublox ZED-F9T [32] GPS timing module that advertises $\pm$4 ns jitter, we saw jitter over 10μs. As we diagnosed the problem, we saw several sources of jitter throughout the hardware and software stack which made it difficult for our driver to call `pps_event` in a timely manner after the pulse interrupt is raised.

Our initial approach was to use GPS dongles with PPS support over USB[2], which are inexpensive (USD ≈$10), readily available, and usable on nearly every server. The GPS device presents itself as a serial device, and the PPS interrupt is encapsulated as a message over the USB bus. We saw that the polling message-driven nature of USB resulted in high jitter: not only was there a ≈100μs delay (which is easily corrected for), but also ±10μs of jitter that made it difficult to accurately time the pulse. Our next attempt involved using a FPGA to deliver an interrupt over PCIe, since PCIe slots are readily available in most commodity servers. However, while PCIe offered less jitter PCIe interrupts are also message signaled and also saw as much as ±5μs of jitter dependent on device traffic and serial transceiver jitter.

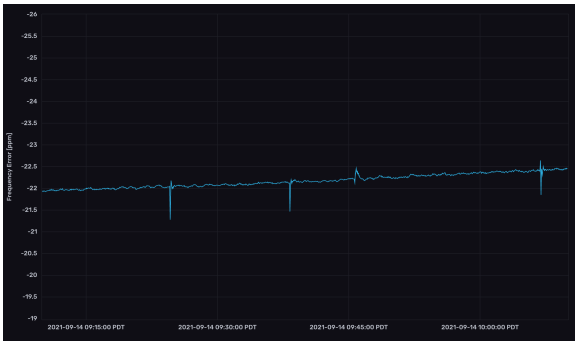
We needed a low-latency interrupt pin to accurately capture the PPS signal. We ended up resorting to using the legacy serial port, which exposes interrupts pins on the device carrier detect (DCD) and clear to send (CTS) lines. Unlike PCIe and USB, these legacy ports drive an interrupt pin on the low-pin count (LPC) bus and offer much lower jitter, on the order of 1μs. Even with the serial port, we still saw significant "blips" in our PPS signal. To reduce those blips, we made several changes: first, we pinned the serial port interrupt to a single core, disabled power management, disabled all watchdogs, installed a "lowlatency" kernel, turned on interrupt threading and set the serial interrupt priority to realtime. While these changes reduced the number of blips, there was still periodic noise present which made time daemons such as chrony detect as much as 10ppm of drift change over a second. This drift only disappeared when we forced the C-state of the machine to C0, disabling idling. This surprised us: the CPU advertised `FEATURE_NONSTOP_TSC`, so the TSC should not be affected by C-States. We realized that the most likely scenario was that when idling was enabled, the CPU would take a non-deterministic amount of time to wake up from sleep and fire the ISR that eventually causes `pps_event` to be recorded.

To deal with this scenario, we took advantage of the two time pulse outputs of the ZED-F9T module and connected the second time pulse to the CTS serial line. We configured the second time pulse with a 400ns delay from the first one, and modified the kernel PPS serial line discipline driver to only record the second pulse if is 400ns ± 100ns from the first pulse. While this caused some pulses to disappear, it greatly reduced the jitter we observed. To compensate for lost time pulses, we changed the time pulse frequency from 1Hz to 3Hz. Removing this software jitter enabled us to see that the clock was actually fairly stable over long periods of time, only deviating by about .5ppm per hour, as seen in Figure 1. We suspected most of this deviation was due to the rising ambient temperature.

[2]To expose the PPS signal, we used a common FT232H USB-to-RS232 converter and connected the PPS line to the DCD signal expected by the PPS serial line discipline driver.



(a) Frequency Error Without Dual Time Pulse, all C-states enabled.



(b) Frequency Error with Dual Time Pulse

Figure 1: **Software Frequency Error.** Interrupts and system activity give the illusion clock error.

Table 2: Systems Evaluated and Temperature Sensors

Name	Type	Crystal Location	Sensors
Server	2S 1U Rack	Near Chipset	50
Workstation	Desktop	Chipset	8
Pi 4	SoC	Under SoC	1
Pi 3	SoC	Under SoC	1

Sensors. Modern computer systems are littered with sensors for environmental conditions. The original use of these sensors were to monitor alarm conditions: for example, to shut off the system if there are abnormally high temperatures that would cause instability, or if a voltage regulator malfunctions. A more recent use of temperature sensors is for thermal throttling, which reduces the frequency of a processor or GPU based on the temperature. The goal of Graham is to reuse these temperature sensors for the purpose of performing software-based temperature compensation.

Using these temperature sensors can be challenging because their location relative to the clock crystal is not consistent. While crystals are usually located near the clock generator, the clock generator can be located in a number of locations, which might not be at all near a temperature sensor. Systems also have a varying number of sensors, as shown in Table 2. The server platform we evaluated, for example,

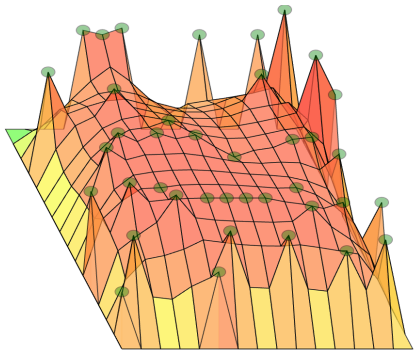


Figure 2: **Server Platform Temperature Map.** The server platform contains over 50 sensors with approximate positions labeled.

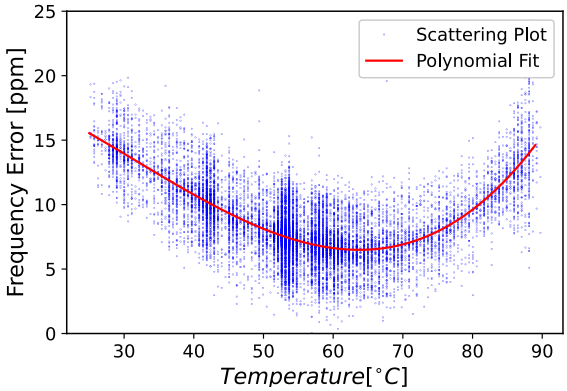


Figure 3: **Temperatures and Polynomial Fit.** Even though there is variation in the measured delay, a polynomial fit curve can still be plotted against it.

has nearly 50 sensors (Figure 2). However, even though the platform provides the position of these sensors, it is still of little help to determine which sensor is closest to the crystal. As an additional challenge, not all temperatures offer the same precision. For instance, some of the sensors in the server platform only reported $\pm 10^{\circ}$C changes, likely because they were designed only for use as an alarm. Finally, the response time of the sensors may vary depending on various environmental factors. For instance, a sensor located near the large copper ground plane of the motherboard may respond slower to rising temperatures than a sensor located on a the thinner PCB of a DIMM. An ideal sensor has high precision and responds quickly to changes in the same way as the crystal.

Establishing the Ground Truth. Armed with an accurate timing signal and a number of candidate sensors, our next goal is to attempt to establish the "ground truth", or the temperature-clock error response curve. If we can determine the clock error given a certain temperature, then we can correct the clock even in the absence of the accurate timing signal.

Nearly all quartz crystals used in computers today are AT-cut crystals. Their frequency relationship with temperature can be described by a 3^{rd} order equation [3, 8, 12, 36]:

$$\Delta f_T = k_0 + k_1 T + k_2 T^2 + k_3 T^3 \quad (2)$$

where Δf_T is the crystal frequency error due to temperature, T is the crystal temperature and k_i are coefficients of the frequency versus temperature curve. To find the relationship of the clock frequency versus temperature we need to solve for the k_i parameters using synchronization messages from a reference clock. Unfortunately, since the sensor data is noisy, we may need to obtain many temperature points to "average out" the sensor error. This required designing an experiment which required many passes, and was difficult to perform on a server platform. As a result, we performed most of our ground truth tests on the Raspberry Pi (Pi 3/Pi 4) SoC systems, though we show our full implementation of Graham in action on desktop and server platforms in Section 5. While the Raspberry Pi is an ARM-based SoC, it runs Linux like the x86 system and has a clock driven by a quartz crystal on the underside of the SoC PCB.

The Pi, as a bare SoC system, allowed us to easily subject it to various temperatures. The Pi includes a temperature monitor which measures the core SoC temperature. We provided an accurate PPS timing pulse using a uBlox Neo-M8N GPS module [31] to a Pi GPIO and exposed it to various temperatures using either a hair dryer or ice bucket. We used the difference in timing ticks between PPS signals to calculate the estimated frequency error of the crystal, and the result is plotted in Figure 3. The distribution we saw was around ± 5 ppm and probably attributable to interrupt delay and sensor error.

Once we saw that we were able to capture the temperature-error relationship, we wanted to ensure that the data we were generating was repeatable, so we collected several traces using varying temperature patterns, all exercising the same temperature range. Figure 4 shows that the curve we generated was similar even with different temperature inputs.

Next, we wanted to see if the curves differed across devices. Figure 5 shows that even across devices of the same model, curves are significantly different. Even the same crystal model could be cut slightly differently, resulting in two 25MHz crystals which are for example, 24.997MHz and 25.001MHz that meet the tolerance requirement, but yield different curves.

Finally, because age can have an effect on the crystals, we wanted to test if we could observe a change in the curve with age. In Figure 6, we ran two tests with a 7 month time difference, obtaining two slightly different curves, as expected. The 1ppm offset we obtained roughly matches the aging expected by a regular quartz crystal during this time period.

Now that we have obtained the ground truth using an accu-

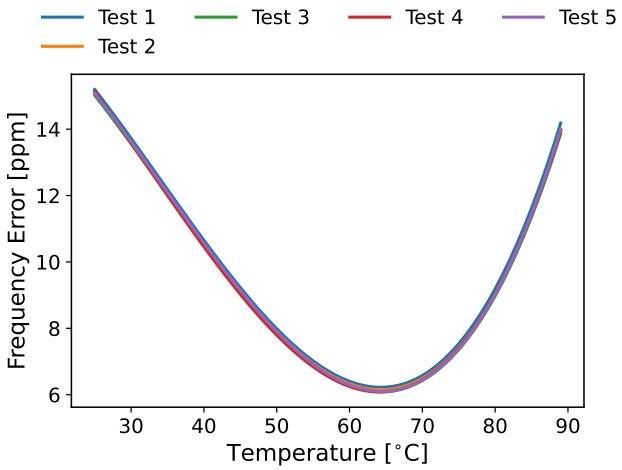


Figure 4: **Repeatability of the Curve.** We measured several traces using different temperature patterns (Test 1-5) by varying the use of ice and the hair dryer and we obtained similar temperature curves.

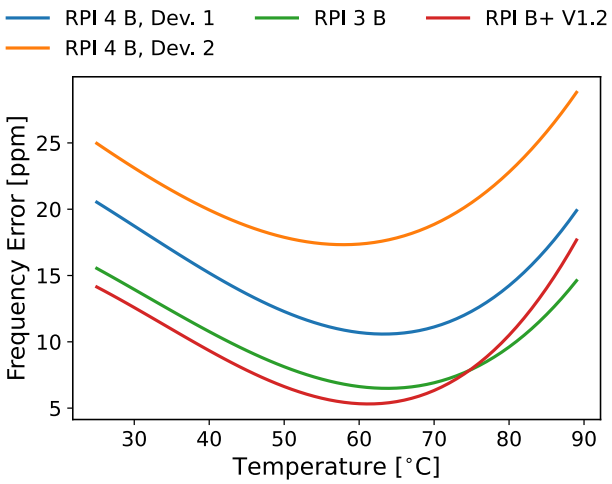


Figure 5: **Curves Across Devices.** We observed that different devices, even of the same model had varying curves.

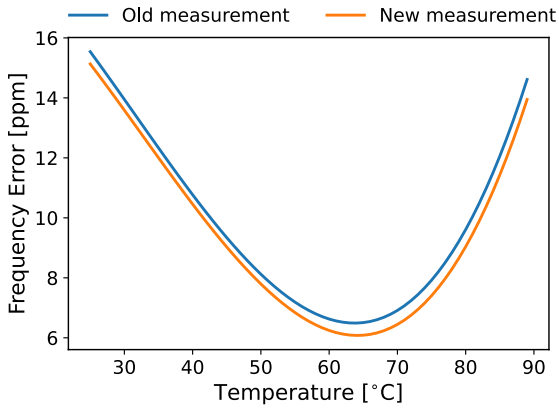


Figure 6: **Aging of Devices.** As a device ages, the curve can change due to crystal aging effects.

rate PPS signal, we use this knowledge to guide us in scaling our solution to many devices. Since each device will have its own unique curve, it became clear to us that we needed to design a way to automatically learn the curve of each device.

4 Graham Design

The overall approach of Graham is to learn the temperature-clock error relationship by fitting curves as new data points are learned. Unlike the experiments we designed when trying to learn the ground truth, we cannot expect to be able to point a hair dryer or dump a production server in ice. In addition, since a truly scalable solution should not require a precise PPS timing signal, we need to ensure that we can perform this learning with traditional synchronization protocols such as NTP or PTP. As a result, Graham must fit these curves over time on incomplete and noisy data. Once we determine that the we have observed enough data points, we can use the derived curve to correct the time error. To fit this data on a curve, we begin by formalizing the variables and equations required to solve for the time error.

4.1 Formulating the problem

We previously described the relationship of the crystal error with temperature as a cubic polynomial in Equation 2. However, we cannot directly measure the frequency of the crystal to obtain the error. Instead, we can obtain two timestamps from the clock using a known time interval and calculate the difference to see how much it deviates from the expected difference.

For example, a clock crystal may have an ideal frequency (f_0) of 32.768KHz. We would expect two timestamps taken exactly 1 second apart to have a difference of 1 (Δts_i). But if we actually observe 1.5 seconds (Δts_o), then we know the actual frequency is 49.152KHz (f_1), or $1.5 \times f_0$. If we subtract the two frequencies, we obtain 16.384KHz of frequency error (Δf). We can express this as an equation:

$$\Delta f \Delta ts_i = \Delta ts_o - \Delta ts_i \tag{3}$$

in which Δf is the relative frequency error. If we assume most of the frequency error is from temperature, we can replace Δf_T in Equation 2 with Δf. Then we obtain:

$$(k_0 + k_1 T + k_2 T^2 + k_3 T^3)\Delta ts_i = \Delta ts_o - \Delta ts_i \tag{4}$$

Eq. 4 is a linear equation with 4 unknowns – k_0, k_1, k_2 and k_3. Timestamp interval Δts_o can be obtained from the

system's local clock and the timestamp interval Δts_i can be obtained from synchronization messages. If we receive N synchronization messages then we can build N linear equations as follows:

$$AK = B \tag{5}$$

in which A, K and B are matrices equal to:

$$K = \begin{bmatrix} k_0 \\ k_1 \\ k_2 \\ k_3 \end{bmatrix}, \quad A = \begin{bmatrix} 1 & T_1 & T_1^2 & T_1^3 \\ 1 & T_2 & T_2^2 & T_2^3 \\ \dots & \dots & \dots & \dots \\ 1 & T_N & T_N^2 & T_N^3 \end{bmatrix} \tag{6}$$

$$B = \begin{bmatrix} \Delta ts_{o,1} - \Delta ts_{i,1} \\ \Delta ts_{o,2} - \Delta ts_{i,2} \\ \dots \\ \Delta ts_{o,N} - \Delta t_{i,N} \end{bmatrix} \tag{7}$$

in which T_N, $\Delta ts_{i,N}$ and $\Delta t_{o,N}$ are respective parameters for the N^{th} synchronization message and equation. Graham solves Eq. 5 using linear least square methods.

So far, we assumed that the temperature is constant for the duration of Δts_o. If the synchronization messages are infrequent, as in the case of a protocol such as NTP, the temperature can change during this period. To solve this problem, Graham records temperatures during this period and when it receives a synchronization message, it aggregates the effects of temperatures. Assume there are n intervals in which we record temperatures during a period. The equation for the j^{th} time interval is:

$$\Delta f_j \Delta ts_{i,j} = \Delta t_{o,j} - \Delta t_{i,j} \tag{8}$$

$$\Delta t_o = \sum_j^n \Delta t_{o,j} \tag{9}$$

$$\Delta t_i = \sum_j^n \Delta t_{i,j} \tag{10}$$

$$\sum_j^n \Delta f_j \Delta t_{i,j} = \sum_j^n \Delta t_{o,j} - \sum_j^n \Delta t_{i,j} \tag{11}$$

Using Eq. 2, 9 and 10, we get:

$$\begin{aligned} &k_0 \sum_j^n \Delta ts_{i,j} + k_1 \sum_j^n T_j \Delta ts_{i,j} + k_2 \sum_j^n T_j^2 \Delta ts_{i,j} \\ &+ k_3 \sum_j^n T_j^3 \Delta ts_{i,j} = \Delta ts_o - \Delta ts_i \end{aligned} \tag{12}$$

where T_j is the temperature at the j^{th} time interval. Note that, $\Delta ts_{i,j}$ is an unknown parameter. We can be approximated it by $\alpha \Delta ts_{o,j}$ in which $\alpha = \frac{\Delta t_i}{\Delta t_o}$.

$$\begin{aligned} &k_0 \sum_j^n \Delta ts_{o,j} + k_1 \sum_j^n T_j \Delta ts_{o,j} \\ &+ k_2 \sum_j^n T_j^2 \Delta ts_{o,j} + k_3 \sum_j^n T_j^3 \Delta ts_{o,j} = \\ &\frac{\Delta ts_o - \Delta ts_i}{\alpha} \end{aligned} \tag{13}$$

Similar to Eq. 4, Eq. 13 is a linear equation with 4 unknowns and we can solve it using similar linear least square methods.

4.2 Implementation

We implemented a prototype daemon in C which solves for the equations by using temperature sensors exposed through sysfs or a network management interface such as SNMP. We record temperatures with 1°C precision at a configurable frequency, which defaults to 1Hz. For synchronization data, we modified chrony to collect the Δts_o and Δts_i necessary from synchronization messages over NTP.

Graham keeps a FIFO queue of equations with known size for each temperature, bounding the number of equations that need to be solved. Graham assumes an operating temperature range of 40-80°C and does not start applying corrections until the curve errors are within 20ppm. Graham constantly collects temperature data to learn the curve before corrections are applied.

4.3 Addressing practical issues

In 4.1, we assumed an ideal case in which all the known parameters to solve for the clock frequency versus temperature are accurate. However, that is not the case in practical systems. We outline these inaccuracies and non-idealities and explain how we can address them.

4.3.1 Timestamp Error

There are two main sources of timing error in the system:

Error in $\Delta t_{o,i}$. Since the temperature changes happen in the timescale of seconds, even several milliseconds error in the observed $\Delta t_{o,i}$ values will have a limited effect on the result.

Error in $(\Delta t_i - \Delta t_o)$. This value is the combination of 3 parameters: crystal frequency error (Δf), jitter in timestamps and network asymmetry from the time server to our system. Graham is interested in only Δf, but the last two parameters are error (δt_{err}) and add noise to our measurements.

$$\Delta t_o - \Delta t_i = \Delta f \Delta t_i + \delta t_{err} \tag{14}$$

Note that δt_{err} is only dependent on the type of timestamping (software and hardware) and the method of the synchronization (NTP, PTP, PPS and ...). The error in curve estimation

is determined by $\frac{\delta t_{err}}{\Delta t_i}$. Therefore, as we increase Δt_i, the first term in Eq. 14 increases while the second term is constant and we can increase the curve estimation accuracy. Moreover, δt_{err} can be modeled as a random variable with zero mean. As we increase the number of equations, we can average out the δt_{err} and in turn the lower the estimation error. By having a high enough number of equations and building equations for longer durations we can increase the curve estimation accuracy.

4.3.2 Temperature Sensor Challenges

Leveraging already existing temperature sensors requires addressing several challenges:

Accuracy. Temperature sensor accuracy has limited effect on correction performance since both learning the relationship of the clock frequency versus temperature and applying the correction is done using the same temperature sensor.

Precision. Low precision means that temperature measurement readings have a random variability. Having a higher number of equations will average out these random errors. This means that as the temperature sensor's statistical measurement variance increase we need higher number of equations.

Responsiveness. A temperature sensor which does not respond to temperature in the same way the crystal does will limit the effectiveness and potentially contribute to error. This responsiveness of a sensor can be measured by checking the temperature error curve. In a system with multiple temperature-error curves, we select the sensor which minimizes the frequency error during learning runs.

4.3.3 Computation Accuracy

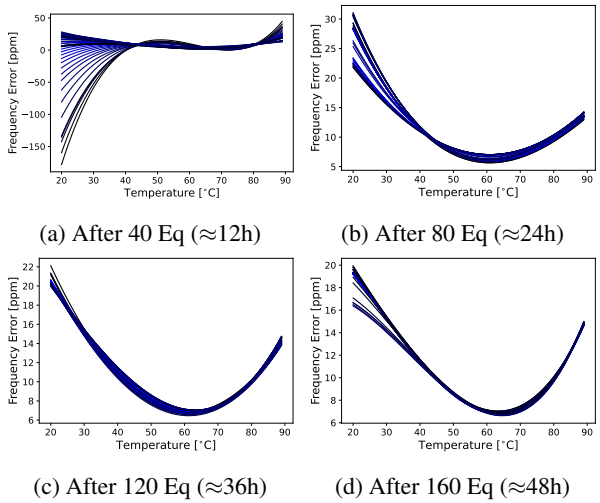
The computed curve is only accurate for the temperature ranges that the system has experienced. For example, if Graham only has equations for temperatures from 50°C to 80°C, the curve is accurate in that range and close to boundaries of that range. As we go far from this boundary the accuracy of the curve decreases. One of the main reasons for this is that the temperature-error curve is cubic, but the typical operating range of the server is only within a small convex region of the curve. Two of the roots are likely at the extreme temperature ranges, and one root is likely in the extreme negative (below freezing region).

To exercise a variety of temperature ranges without using a heater or ice, we load the CPU and allow the system to cool off.

5 Evaluation

Our evaluation of Graham is motivated by the following:

- How effective is learning over a noisy synchronization channel such as NTP? (§5.2)



(a) After 40 Eq (≈12h) (b) After 80 Eq (≈24h)

(c) After 120 Eq (≈36h) (d) After 160 Eq (≈48h)

Figure 7: **NTP Learning.** While learning over NTP takes longer, the curve converges towards the same curve produced by faster, more accurate synchronization sources.

- What is the holdover time Graham can achieve, and how many synchronization failures can it tolerate? (§5.3)
- Can Graham compensate for rapid changes in temperature, as in with a HVAC failure?(§5.4)

Test Platforms. The primary system requirement to be able to apply Grahamis the presence of a temperature sensor which is present in nearly all modern computer systems. We evaluated Graham on several platforms, as shown in Table 2. For the Pi tests, we used a ublox M8N [31] GPS receiver with a time pulse accuracy of ±60ns (99%). The M8N module does not specify jitter, but we observed ±20ns jitter using a RIGOL MSO5074 oscilloscope. For the x86-based platforms, we used a ublox ZED-F9T [32] GPS module which specifies a time pulse accuracy of ±5ns (1σ), and a jitter of ±4ns. In our tests, we are mainly concerned about jitter, as the timing accuracy specifies the accuracy of the timing pulse to GPS time, and these GPS modules have their own TCXO oscillator.

5.1 Learning over PPS

We obtained baseline curves with Graham using PPS. With PPS, we generate 1 new equation per second, corresponding to the frequency of the synchronization signal. As shown in Figure 8, even though the temperature data we used to generate each curve was quite different, the curves are almost the same. While the curves look similar, the constants for each curve varies. This is because there are many cubic equations which can fit the small convex portion of the curve that we observe within the operating temperature range.

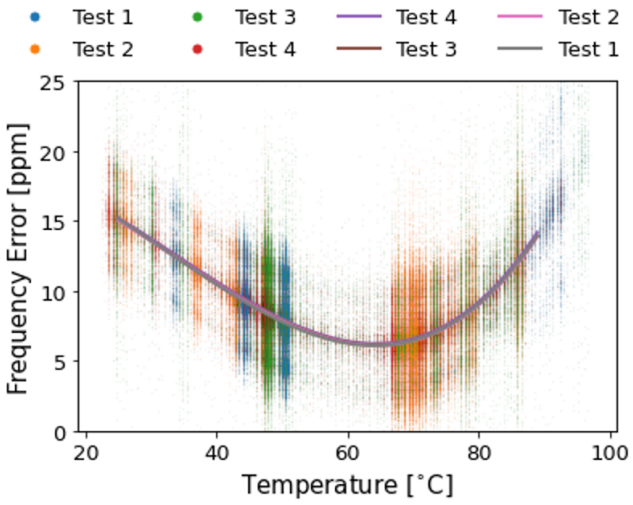
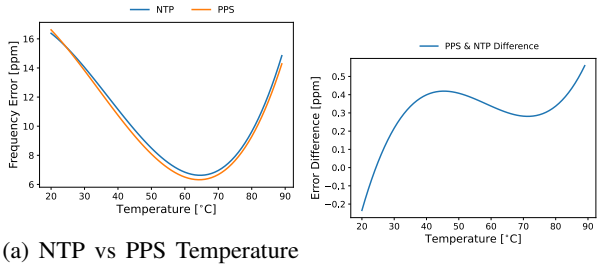


Figure 8: **PPS Curves and Input Data.** Curves learned over each experiment have a similar shape despite having variable input data.



(a) NTP vs PPS Temperature Error Curves

(b) NTP vs PPS Error

Figure 9: **NTP Performance Compared to PPS.** The learned curves for NTP are within 0.5ppm of the PPS curves for most of the operating range.

5.2 Learning over NTP

To evaluate learning over NTP, we used chrony to obtain NTP synchronization data for Graham against public NTP servers over a standard home cable broadband connection, with a ping latency to the NTP server between 30–40ms. This resulted in synchronization accuracy in the ms range. In order to compensate for this, we needed to use high Δts_o. For NTP, we use Δts_o = 1000s, which results in one equation every 1000s as opposed to 1 equation per second with PPS. Note that Δts_o is independent of the synchronization periods and intervals used by chrony, which has its own algorithm for NTP synchronization frequency.

Figure 7 shows the 160 equations we collected over the course of a 48 hour run. This resulted in a curve within ±0.5ppm of the curve generated using PPS signals, as shown in Figure 9. We suspect that the error of the curve is not constant because of lack of data points at temperature extremes for both sets of data.

5.3 Holdover

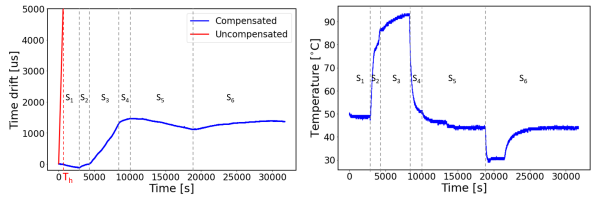
Once we have learned the temperature-error relationship, we wanted to evaluate how well Graham's time frequency correction would perform in the absence of synchronization messages. To test the accuracy of the frequency correction, we recorded the accurate PPS time pulse, but did not provide it to Graham. We measured the accuracy of Graham's time correction versus the real time. We then exposed the system to a new temperature trace.

Pi Experiments. Figure 10 shows a trace of one of these experiments on the Pi 3. In this particular experiment we exposed the system to both ambient air effects of the 8 hour time period as well as artificial cooling (ice) and heating (hair dryer). The red vertical line shows the rapid growth of time error if Graham did not perform any compensation. At 620s, this well exceeds 5000μs of drift, which corresponds to the the 8ppm of temperature drift Graham is trying to correct for. On the other hand, Graham's corrections perform very well, never exceeding 1500μs of error over the course of the entire 8 hour run, even though the temperature is shifting significantly. For most of the test, the slope never exceeds 100ns/s of error, which means the clock is performing as well as one with only 100ppb of error, a 200× improvement over the performance of the 20ppm crystal, performing nearly as well as a high quality TCXO or some OCXOs. We can calculate the holdover time using the slope from Equation 15, given a maximum time uncertainty (ε). If ε=1μs, then the holdover time during the 100ns/s region is:

$$t_h = \frac{1\mu s}{100ns/s} \tag{15}$$

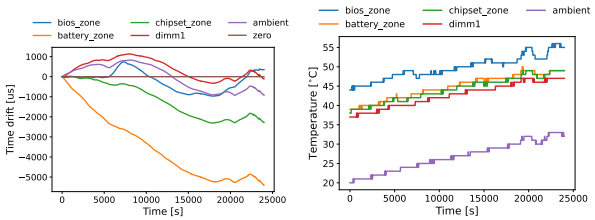
or $t_h = 10s$. In other words, the corrected clock will not exceed 1 μs of error for at least 10 seconds *without* any additional synchronization. This would enable more infrequent synchronizations, or enable the system to tolerate the very real potential of missed synchronization messages. In one part of the graph, we experience a 330ns/s slope, when the temperature exceeds 85°C. We speculate that this slope is because the training temperature data we used had very few points at or above this temperature. 330ppb still is very good: we obtain a 1μs holdover time of 3 seconds, which still allows for lower frequency synchronizations.

Server Experiments. We also evaluated the holdover time on desktop and server x86 systems. These systems are much more complex and contain multiple sensors and fans, so Graham needs to determine which sensor works best, given a variety of factors. There are also multiple components which can generate heat load, which vary from system to system. Notably, the many fans in the server made it more difficult to



(a) Time drift (b) Temperature

Figure 10: **Holdover.** The uncompensated temperature drift quickly increases while Graham is able to maintain the time with minimal drift. The slope of each part of the graph corresponds to the frequency error performance: $s_1 = 50$, $s_2 = 80$, $s_3 = 330$, $s_4 = 100$, $s_5 = 30$, $s_6 = 15$ ns/s.

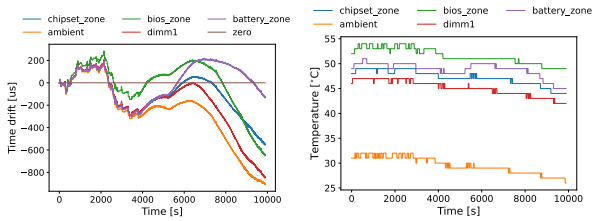


(a) Server drift (b) Server sensors

Figure 11: **Warming Server Holdover.** With a server sitting in a garage on a hot summer day, Graham is able to maintain 0.1ppm of error.

create rapid changes in temperature. To get a picture of the server sensor's performance, we performed a 24 hour learning run exposing the server to various temperatures while heating it from the fan intake and letting it cool via ambient cooling, and running `stress-ng` in various modes to create load on the system. We then exposed the server to a new temperature curve.

Figure 11 shows the holdover graph for the server during one of our first tests, which is just ambient warming of the server in a garage on a hot summer day. We selected the 5 best performing sensors. Surprisingly, even though we thought the "chipset_zone sensor" would perform the best, "dimm1" actually produced the best correction curve. We wanted to ensure that this would be the case even in a loaded system, so we performed a memory test using `stress-ng` to see if heat from a memory load would affect our learned result. Figure 12 shows the holdover curves from that run, with the memory test running at time 0. The DIMM 1 sensor still remained one of the top performing sensors, producing less than 200μs of drift over the first 2000 seconds of the run, or 0.1ppm error. Many of the other sensors perform well too, likely because they experience similar patterns of temperature changes. The impact of the load, however, can be seen across Figure 11 and Figure 12: while the ambient temperature works well without a load, its performance is worse when a load is present. In all our runs with the server workload, we never observed more



(a) Server drift (b) Server sensors

Figure 12: **Memory Load Holdover.** The DIMM sensor remains the best sensor, even when the server is under memory load.

than a 0.2ppm error with Graham.

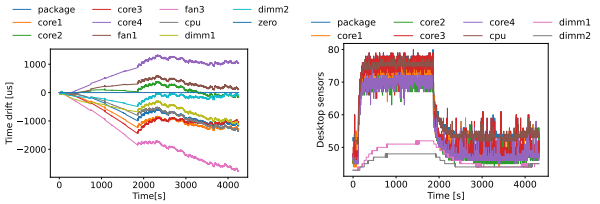
Desktop Experiments. Finally, we evaluated Graham on a typical desktop machine. Unlike the server, which is fully instrumented with sensors throughout, the desktop machine we used only had a few sensors exposed by default, just on the CPU die and the DIMMs. However, during our experiments, we made an error to include the output of the fan sensors (in RPM) as training data. Surprisingly, the fan sensors worked well even though they were not directly measuring the temperature. We suspect that the speed of the fan is driven by a combination of the ambient temperature, (which is not exposed to the user) and dynamic CPU load by the hardware monitor. However, we ended up using the second core sensor, which is located closer to the chipset and crystal. This gave us 0.1ppm error on nearly all experiments.

Figure 13 shows a peculiar experiment on the desktop platform where we failed to expose the server to all temperature points. In the first 2000 seconds we run a CPU load experiment, which resulted in the a higher than expected error (0.5ppm vs 0.1ppm). After debugging, we realized this was because the temperatures we exposed to Graham during testing (Figure 13c) were not learned (Figure 13d). In particular, the testing temperatures were above 70°C for the first 2000s, while the learning temperatures were below. Still, we thought this test showed that even without learning temperatures, Graham can provide some correction to the temperature error.

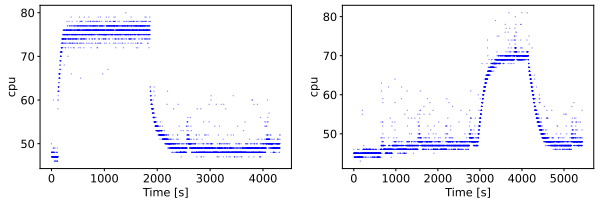
5.4 Rapid Changes

One of the often cited sources of timing instability in computer systems is a thermal shock event, such as an HVAC failure. To evaluate Graham's performance in dealing with a rapid thermal shock, we used the Pi system and pointed a hair dryer directly at it, attempting to raise the temperature rapidly to the maximum operating temperature. As with the holdover tests, we turned off synchronization and only relied on Graham's temperature-based frequency error correction.

Figure 14 shows the time drift after correction by Graham (left) which results from the rapidly rising slope in tem-

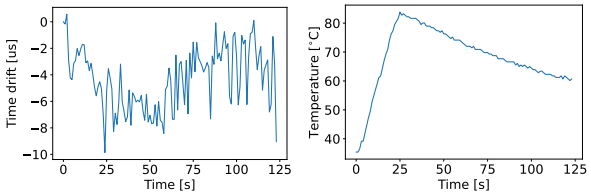


(a) Desktop Drift (b) Desktop Sensors

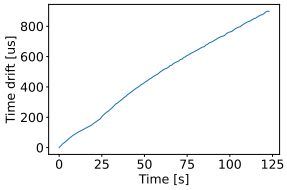


(c) Testing Temperatures (d) Learning Temperatures

Figure 13: **Desktop Holdover with Missing Points.** Even with missing points, Graham is able to make corrections to keep error within 0.5ppm.



(a) Time drift (b) Temperature



(c) Uncorrected

Figure 14: **Thermal shock.** Even with a hair dryer's rapid heat, Graham is able to quickly compensate for errors in time, never drifting beyond 10μs.

perature (right) from the hair dryer. Using the hairdryer, we were able to produce a 2°C/s slope, which cools at about 0.2°C/s. While we feel that such fast heating is unlikely to happen, it may be representative of an HVAC failure, and is an indication of the robustness of Graham's temperature correction: the time drift never exceeds more than 10μs over the initial 25 second slope, a drift of only -0.4ppm. Once the temperature slope decreases, Graham is able to maintain the time without exceeding the initial 10μs of error. Without Graham, the system accumulates nearly 1ms of error during this time period (bottom).

6 Discussion

Our evaluation has shown that Graham can maintain clock frequency error below 1ppm using commodity sensors in a variety of conditions. Graham is only one part of the solution, however – while Graham can maintain a long holdover time, the synchronization maintained will only be as good as the initial synchronization.

Graham works in synergy with other synchronization mechanisms, such as Huygens [14], PTP [10] and FaRMv2 [28] to maintain synchronization. Our experiments with NTP show that Graham can maintain 1μs ε for 10 seconds after loss of synchronization. As Sundial [20] shows, however, missed synchronizations can occur for a number of reasons. For Huygens, significant CPU load on the system could occur causing the SVM processing to be delayed, and in PTP and FaRMv2, synchronization messages could be missed, leading to increased uncertainty of time. Using our 1μs holdover result for Graham, we could reduce the standard 1s synchronization frequency of PTP to 3s and tolerate 2 lost synchornization messages.

Graham also aims to democratize precise time by enabling commodity servers, desktops and even SoCs to have access to stable clocks without adding specialized hardware. One of the barriers we see in adopting precise time for these devices is the myth of the unstable clock, which is perpetuated by the challenge of measuring the drift in the clock in the first place. Software noise can give the illusion that a clock is drifting rapidly, even though hardware clocks are relatively stable. Unfortunately, without specialized hardware, drift is measured by software itself, further exacerbating the problem. By characterizing the clock. Graham enables applications to trust the hardware instead of relying on noisy software measurements.

In the future, we may consider incorporating multiple sensors to the equations Graham solves for better accuracy. As more applications require precise time, we expect systems with TCXOs or OCXOs to come on the market, and expect that Graham performs favorably against them.

7 Conclusion

It has been long thought that computer clocks are unstable, and that stability cannot be achieved without frequent synchronizations. We hope that this work dispels that myth and convinces the reader that much perceived clock instability is due to software measurement error. By understanding the sources of clock error, we have built Graham, which can reduce local clock error well below 1ppm using commodity clock sensors. Combined with an accurate synchronization source, Graham can maintain microsecond clock accuracy without additional hardware.

References

[1] Abracon. Abracon ABLKJO crystal oscillator. https://abracon.com/Precisiontiming/ABLJO.pdf.

[2] abracon. Aoc2012vajc ocxo datasheet. https://abracon.com/datasheets/AOC2012-Series.pdf.

[3] Abracon. Tuning fork crystals and oscillator. https://abracon.com/Support/Tuning-Fork-Crystals-and-Oscillator.pdf.

[4] BCM53903. Bcm53903 timing over packet (top) processor for precision timing applications. https://www.broadcom.com/products/embedded-and-networking-processors/communications/bcm53903.

[5] Chrony. chrony – introduction. https://chrony.tuxfamily.org/.

[6] Clockwork. Tick tock networks is now clockwork. https://www.clockwork.io.

[7] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[8] CTS Corporation. Crystal basics. https://www.ctscorp.com/wp-content/uploads/Appnote-Crystal-Basics.pdf.

[9] Intel Corporation. C620 platform controller hub datasheet. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/c620-series-chipset-datasheet.pdf.

[10] John C Eidson, Mike Fischer, and Joe White. Ieee-1588™ standard for a precision clock synchronization protocol for networked measurement and control systems. In *Proceedings of the 34th Annual Precise Time and Time Interval Systems and Applications Meeting*, pages 243–254, 2002.

[11] Epson. Tg2016smn ((tcxo / vc-tcxo) high stability). https://www5.epsondevice.com/en/products/tcxo/tg2016smn.html.

[12] Fil-Tech. Frequency-temperature behavior of at-cut crystals. https://www.filtech.com/tech-library/document/frequency-temperature-curve-cut-crystals/?dl=1.

[13] Marvin Frerking. *Crystal oscillator design and temperature compensation*. Springer Science & Business Media, 2012.

[14] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 81–94, 2018.

[15] D Habic and D Vasiljevic. Temperature compensation of crystal oscillators using microcontroller-/spl mu/ctcxo. In *Proceedings of IEEE 48th Annual Symposium on Frequency Control*, pages 587–593. IEEE, 1994.

[16] IBM. https://www.ibm.com/docs/en/power-blade-server/version_undefined?topic=planning-vibration-shock, Accessed: Feb 7 2022.

[17] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 454–467, 2016.

[18] Bojie Li, Gefei Zuo, Wei Bai, and Lintao Zhang. 1pipe: scalable total order communication in data center networks. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 78–92, 2021.

[19] Chao Li and Arvind Sridhar. Vibration and shock sensitivity: A comparative study of oscillators. *Texas Instruments, Dallas, TX, USA, Appl. Note SNAA296*, pages 1–11, 2017.

[20] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, et al. Sundial: Fault-tolerant clock synchronization for datacenters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1171–1186, 2020.

[21] LinuxPPS. Linuxpps wiki. https://www.linuxpps.org.

[22] Pedro Moreira, Javier Serrano, Tomasz Wlostowski, Patrick Loschmidt, and Georg Gaderer. White rabbit: Sub-nanosecond timing distribution over ethernet. In *2009 International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, pages 1–5. IEEE, 2009.

[23] Ali Najafi, Amy Tai, and Michael Wei. Systems research is running out of time. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 65–71, 2021.

[24] ntpd. ntpd - network time protocol (ntp) daemon. https://docs.ntpsec.org/latest/ntpd.html.

[25] Bruce M Penrod. Adaptive temperature compensation of gps disciplined quartz and rubidium oscillators. In *Proceedings of 1996 IEEE International Frequency Control Symposium*, pages 980–987. IEEE, 1996.

[26] Renesas. 932sql456 - low-power ck420bq derivative for pcie separate clock architectures. https://www.renesas.com/us/en/document/dst/932sql456-datasheet?r=166281.

[27] Renesas. Limiting IEEE 1588 slave clock wander caused by packet delay variation. https://www.renesas.com/us/en/document/whp/limiting-ieee-1588-slave-clock-wander-caused-packet-delay-variation.

[28] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data*, pages 433–448, 2019.

[29] SiTime. Sitime 5146 super-tcxo. https://www.sitime.com/support/resource-library/datasheets/sit5146-datasheet.

[30] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1493–1509, 2020.

[31] ublox. ublox neo-m8n datasheet. https://www.u-blox.com/sites/default/files/NEO-M8-FW3_DataSheet_UBX-15031086.pdf.

[32] ublox. ublox zed-f9t datasheet. https://www.u-blox.com/en/docs/UBX-18053713.

[33] John R Vig. Quartz crystal resonators and oscillators for frequency control and timing applications. a tutorial. *Nasa Sti/recon Technical Report N*, 95:19519, 1994.

[34] Miao Xu, Wenyuan Xu, Tingrui Han, and Zhiyun Lin. Energy-efficient time synchronization in wireless sensor networks via temperature-aware compensation. *ACM Transactions on Sensor Networks (TOSN)*, 12(2):1–29, 2016.

[35] Zhe Yang, Lin Cai, Yu Liu, and Jianping Pan. Environment-aware clock skew estimation and synchronization for wireless sensor networks. In *2012 Proceedings IEEE INFOCOM*, pages 1017–1025. IEEE, 2012.

[36] Hui Zhou, Charles Nicholls, Thomas Kunz, and Howard Schwartz. Frequency accuracy & stability dependencies of crystal oscillators. *Carleton University, Systems and Computer Engineering, Technical Report SCE-08-12*, 2008.