Systems Research is Running out of Time

Ali Najafi
VMware Research
Seattle, WA, USA

Amy Tai
VMware Research
Palo Alto, CA, USA

Michael Wei
VMware Research
Palo Alto, CA, USA

ABSTRACT

Most sciences conduct experiments with a thorough understanding of the accuracy and precision of the instruments used for making measurements. Time is the most frequently used measurement in systems research, yet most of the literature does not consider the precision and accuracy of clocks. In this paper, we argue for the importance of understanding timekeeping and providing precise and accurate time for general systems research.

CCS CONCEPTS

• **Hardware** → *Testing with distributed and parallel systems*; • **Computer systems organization**; • **General and reference** → **Measurement**; **Experimentation**; **Metrics**; **Computing standards, RFCs and guidelines**;

ACM Reference Format:
Ali Najafi, Amy Tai, and Michael Wei. 2021. Systems Research is Running out of Time. In *Workshop on Hot Topics in Operating Systems (HotOS '21), May 31–June 2, 2021, Ann Arbor, MI, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3458336.3465293

1 INTRODUCTION

Measuring time is the foundation of systems research. The majority of systems evaluations consist of evaluating system artifacts through benchmarks and comparing the results against other state-of-the-art systems. Obtaining these measurements involves timing various operations through the use of a clock. The quality of these clocks is often unknown to the benchmark designer. For example, in Linux, benchmarks often obtain time from the `clock_gettime()` function, which returns a `timespec` from various user-selectable clocks. The `timespec` structure, however, returns no information about the precision or accuracy of the clock, only a counter value in nanoseconds. Notably, the quality of the clock can depend on many factors, such as whether or not the system is virtualized, the selected clock source, the overhead of reading that clock source, time synchronization activity on the system, and even the temperature of the system itself. Measurements taken using different clocks can be incomparable: not only can using different timing sources result in different absolute times, but the overhead of reading time can also induce variations in the runtime behavior of the experiment itself. While systems research follows best practices to reduce the amount of error introduced by the timing, most benchmarks trust the time output to not only be correct, but have little effect on the experiment itself. This practice runs in contrast to other fields, where measurement equipment is frequently calibrated, and its effects on the experiment are well known.

In this paper, we argue that understanding how to measure the passage of time accurately is critical to systems research and that current interfaces for measuring time are insufficient because they do not provide the error or accuracy of the underlying clocks. We show that the abstractions we have built for obtaining time make it difficult to verify the time provided, and failing to verify that the time is correct can lead to actual benchmarking errors. We argue that every system evaluation should calibrate clocks before running experiments and that this calibration is inexpensive. We hope that this paper encourages the use of and standardization of accurate and precise time in systems and systems research.

2 TIME IN COMPUTER SYSTEMS

System clocks are built by incrementing a counter at a regular interval. In hardware, an oscillator produces a regular waveform known as a clock which increments the counter at a regular frequency. In most computer systems, this oscillator is a quartz crystal cut to resonate at a specific frequency. Different oscillators can provide different levels of accuracy (how close to the advertised frequency the oscillator resonates) and drift (how much the oscillator's frequency changes over time), often specified on the oscillator's datasheet. Typical quartz has an error of about 20ppm [21], which is 0.002% error, or roughly 20 microseconds per second. This error is often temperature-dependent, and crystal manufacturers publish the temperature dependency in a datasheet.

While this clock is imperfect, we will see that the physical characteristics of this clock only contributes a tiny amount to the overall error that may skew experimental results.

HotOS '21, May 31–June 2, 2021, Ann Arbor, MI, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8438-4/21/05.
https://doi.org/10.1145/3458336.3465293

2.1 Time in x86 Linux

There are several system clocks in a modern x86 system, known as clock sources in Linux, which tick at different rates. At boot time, the Linux kernel reads the clock from the real-time clock (RTC). A battery powers the RTC, so this clock continues to tick even when the system is off. Because this clock ticks at a relatively low frequency and is expensive to access, it is not usable for precise measurements. Instead, the kernel uses another counter, known as the timestamp counter (TSC), which ticks at a much higher frequency, enabling measurements with nanosecond precision. The TSC is readable through the `RDTSC` instruction, which is unprivileged and accessible to userspace applications. To make the wall time (real time and date) available, the kernel uses the RTC to initialize an offset from the TSC. A userspace daemon such as NTPd [11] or chrony [3] synchronizes this offset to external clocks available on the network. The kernel calculates the frequency of the TSC at boot time and uses it to convert the TSC timestamp to a time in nanoseconds. Several other counters, such as the `HPET`, `APIC` and `ACPI PMT` also exist, which tick at different frequencies. Notably, Linux may automatically select one of these clock sources as the default if it deems that the TSC is unreliable. Historically, the TSC hardware has been unreliable for timekeeping for various reasons: early versions of the TSC used the actual processor clock as the input frequency, so mechanisms such as dynamic frequency scaling would affect the tick rate, or turned off the clock to the TSC when the processor was idle, making it impossible to use the TSC to measure time.

Userspace applications obtain time using `clock_gettime(3)`, which provides several clocks. Most benchmarks use `CLOCK_MONOTONIC`, which obtains the timestamp counter with limited monotonically increasing corrections from the system timer, typically TSC without adjustments from NTP. The vDSO mechanism, which injects kernel code and data into userspace, is used to avoid a transition into the kernel and provide the necessary data to calculate the returned time value, such as the conversion factor.

Virtualization can further complicate the path to obtaining time, as hypervisors may migrate virtual machines between processors or physical machines which have different TSC counters. Hypervisors can trap the `RDTSC` instruction to correct for time, provide time through paravirtual mechanisms such as `kvm-clock` or VMware VMI, or use the TSC scaling feature of newer processors, which enables a hypervisor to provide an offset and effective frequency for the `RDTSC` instruction. Each mechanism has its drawbacks: trapping RDTSC results in an expensive VM-Exit and reduces the precision of the timer, paravirtual mechanisms may have incompatibilities with mechanisms such as vDSO, and TSC scaling might not be supported on all hosts.

2.2 Time in other systems

Keeping track how time is obtained has become more difficult as computer systems have become increasingly heterogenous. For example, Microsoft® Windows® offers the `QueryPerformanceCounter()` API [18], which has its own mechanism to select whether the `HPET`, `ACPI PM` or `TSC` is used. On Android™, high precision timestamps are usually retrieved using native code [5]. On ARM-based SoCs, the clocksource can vary, and retrieving the time from the cycle counter requires privilege, unlike on x86 processors [2]. In the browser, timing calls are intentionally made noisy and imprecise to defend against timing attacks [9]. Despite the diversity of methods to obtain time, these timing calls all return a simple timestamp without the semantic information of where the time came from and how accurate it is. In the next sections, we show that this semantic gap can lead to problematic conclusions about time.

3 COMPLICATIONS OF TIME AND PITFALLS

While the abstractions we have built to make it easy for developers to retrieve precise timestamps, losing the semantic details about the clocks used to retrieve time can lead to incorrect, or worse, misleading evaluations.

For instance, in virtualized environments, the frequency of the TSC is unavailable to guest OSes in the mainline kernel.[1] LLVM's X-Ray profiling tool erroneously assumes that it can calculate the TSC frequency from the processors' maximum frequency, which is no longer valid in recent x86 processors that utilize turbo. Without the correct frequency, X-Ray can generate flame graphs where the sum of function runtimes are less than the total runtime of the test [26].

SMI handlers may actively modify the TSC in order to hide their execution [19], resulting in incorrect execution times, especially for very short microbenchmarks. Misconfiguration of the clocksource, especially in virtualized environments, can result in noisy or incorrect time measurements due to the overhead of obtaining time [20]. Overclocking can also affect a benchmark by changing the frequency of the underlying timers [13].

Despite the complexity of the pathway for obtaining time, we tend to treat evaluation results as comparable. If we run the same benchmark on two systems, a system that reports 10,000 op/s should objectively be faster than a system running at 9,000 op/s. However, if the measurement overhead of collecting time is higher in one system over the other, or if the clock is running faster or slower than expected, the results from one system can become incomparable to the other.

[1]It is, however, available in Google's proprietary production kernel [26], and from a custom kernel module [24].

To avoid these pitfalls, the systems community follows a set of best practices, which involve ensuring that the correct clock is selected, appropriate fencing instructions are used when necessary, and that benchmarks are run many times to eliminate outliers and other anomalies. Even when these guidelines are closely followed, timing may still be inaccurate. Unless we closely examine timing results for inconsistencies, these results could be relied upon to draw incorrect conclusions about a system.

An example of these pitfalls manifesting is the "HPET bug" [6], which resulted in incorrect performance benchmark results published on several popular websites that had to be pulled. In particular, these benchmarks showed that the AMD Ryzen 2000 series performed impressively well compared to other processors of the time. It turned out that starting with Intel's Skylake processors, Intel's HPET implementation became more costly to call compared to previous generations, reducing the number of HPET calls from 1.4 to 0.2 million per second, creating the illusion that the Intel systems were slower when that timer was relied on, even though the overall system performance was faster. In their post mortem analysis of the discrepancy, Anandtech wrote:

> ... if HPET was having any effect, it was unnoticed: our results were broadly similar to others, and each of the products fell in line with where they were expected. Over the several review cycles we had, there were a couple of issues that cropped up that we couldn't explain, such as our Skylake-X gaming numbers that were low, or the first batch of Ryzen gaming tests, where the data was thrown out for being obviously wrong however we never managed to narrow down the issue. [25]

This conclusion is troubling: if the AMD Ryzen's numbers were closer to the Intel Skylake's numbers, this error may not have been discovered at all. The reality is that the Anandtech authors were running experiments with uncalibrated equipment: due to the interface provided the operating system, they didn't know what the accuracy or cost of the system timer.

While the HPET bug issue was an example of timer overhead affecting the system's overall performance, errors in the actual timer can occur as well. For example, from kernels 4.10–5.3, the frequency of the TSC was determined using hardcoded values, and particular CPUs were on a "crystal quirk list" [12]. Intel Skylake-X Server and workstation SKUs used crystals with different frequencies (25MHz vs 24MHz), but the crystal quirk list did not account for the difference. As a result, Skylake-X workstations had a TSC clock with at least a 4% drift due to the kernel incorrectly calculating the TSC frequency [1]. Furthermore, EMI reduction circuitry can reduce the frequency by a further 0.25%. This bug was reported as early as 2017, fixed in 2018 and still reported by users as late as 2019 [4, 10]. With this bug, the system appears to run normally, but benchmarks can return faster than expected results due to the slow TSC

Newer server grade motherboards offer overclocking as a standard feature: for example, Supermicro offers a Hyper-Speed platform marketed for low-latency and high performance computing [23]. This server platform changes allows the user to change the BCLK slightly (by up to 6%). This caused the clock to be inaccurate in some versions of Linux which relied on the frequency provided by the processor until the end of 2019, where a patch was written to calibrate the clock to another local clock [14]. Notably, this caused some users to be surprised that their benchmarks returned the same results before and after the overclock was applied [22].

Another bug in the implementation of HPET (which is used to check the accuracy of the TSC) in relatively recent Intel processors (Coffee Lake, Ice Lake) also have caused Linux to fallback to using the buggy HPET, resulting in inaccurate time [8]. When the HPET is used in these systems, the clock stops when idle, resulting in extremely skewed timing results.

None of these bugs affected the typical use of the system: even though the clock was running significantly faster or slower, time synchronization would hide the bug from the user unless the user examined the synchronization logs carefully.

4 WHAT CAN WE DO?

The relative ease in which results based on incorrect timing information can be produced is alarming, and can distort the results of the benchmarks we trust. One of the problems we identified is that the `clock_gettime()` interface is opaque: to understand the accuracy of the timestamp, a benchmark must break through several layers of abstraction, such as which timer is currently being used by the system and its properties. Certainly, the Linux kernel already exposes some of this information: the current clocksource, for example, is accessible via sysfs. An application may also attempt to check the sanity of the time values returned by checking against an external clock source. We feel that given the importance of measuring time in a system, a more principled approach for measuring time is appropriate.

5 TOWARDS A PRINCIPLED APPROACH FOR TIME

Start at the Hardware. The hardware has the best understanding about the characteristics of the clocks in the system. The motherboard manufacturer knows which oscillators were selected and their properties, as well as changes to the clocks and their frequencies in the BIOS. This information should be exposed to software, for instance, via ACPI tables.

The processor should be able to combine the information from the motherboard with its own data to specify to software the exact frequency.

Calibrate the clock. Regularly, and when requested by the user, the system should check that each clock behaves as expected. Calibrating the clock requires synchronizing the clock against known, external time sources, similar to calibration of other measurement instruments. Without this calibration, we only are able to trust the data provided by the designer, and cannot account for run-time error due to configuration or physical effects, such as temperature. Regular calibration of the clock is critical: the properties of a clock can change over time as oscillators age, or when configuration changes are made.

By synchronizing a clock multiple times, we calibrate the time measured against a known time source, by observing the difference measured by the two clocks. While time synchronization might only correct the local time value, calibrating the clock records the error in measuring a timespan, using the same facilities available to applications for reading clocks. Using a calibrated clock enables us to accurately compare results between measurements which use different timing mechanisms.

Several mechanisms already exist for synchronizing clocks. For instance, NTP synchronizes with external time servers. Because of the asymmetric delay involved in NTP, state-of-the-art NTP clients such as Chrony [3] can synchronize clocks within the range of milliseconds. For most evaluations, NTP synchronization should be sufficient to determine if a clock has a large error, as one would expect from a misconfiguration. These are the types of errors most likely to cause significant errors for systems evaluations. Unfortunately, time synchronization can be a double-edged sword: while time synchronization can detect a mismatch against an external time reference, it is typically used to transparently correct for errors in the local clock instead. As a result, a clock that is very fast or slow due to misconfiguration might not be obvious to the user but still produce incorrect results between synchronizations. It is critical that synchronization error be exposed to the user in a meaningful way.

For more detailed evaluations, such as those which compare function level or microarchitecture level performance, obtaining a stronger bound on the clock is desirable.

The Precision Time Protocol (PTP), or IEEE 1588 [7], is a mechanism which can be used to synchronize clocks with hardware support and is available on most server networking hardware. PTP can synchronize clocks to the range of 100s of nanoseconds, but the clock synchronized is located on the network card. A simple daemon can be used to compare another clock, such as the timestamp counter, to the network clock.

If a general purpose I/O (GPIO) pin is available, an external timing input can also be used to calibrate clocks. This timing input is known a pulse-per-second (PPS), and Linux provides a facility to capture PPS inputs for the purpose of synchronizing clocks [17]. We were able to use an inexpensive $10 GPS dongle to provide a PPS output. Unfortunately, as servers have removed legacy ports, most modern servers have made traditional access to GPIO, which was accessible via legacy and serial ports. However, we were able to synchronize the clock by wiring a GPS PPS output to the NMI button and applying a small modification to the kernel to capture the clock when the NMI is triggered, and the same setup could easily be made by and FPGA triggering a message signaled interupt over PCIe. The NMI is exposed on most modern server hardware. GPS clocks, when synchronized, offer on the order of 5-30ns synchronization with GPS satellites, and more importantly, the PPS mechanism offers synchronization accuracy on the order of a few 100ns. This solution offers an extremely low cost approach for calibrating nearly any server. GPS is not strictly necessary: since we are primarily interested only in the clock error and not the absolute time, a waveform generator can be used in the case a GPS signal is not available.

Exposing the calibration. Once we have measured the error in the clock, that calibration should be exposed to userspace applications. First, when the kernel calibrates clocks at boot time, it should discard clocks with errors that are outside the specification declared by the hardware.

Second, the kernel should return calibration information whenever time is requested. This could be an addition to the `timespec` structure, and could be thought of as analogous to a "calibration certificate", specifying that the clocks' accuracy has been verified against a known time source, and what error against the known source was found during the last calibration. When calculating and presenting results, application and benchmarks should present the calibration and apply the calibration data to the results.

6 FUTURE DIRECTIONS

As system research continues to advance as a science, so will the ability to generate repeatable results. Calibrating timers is the start of this process. Our work and experience with timers has highlighted several areas to be explored:

6.1 Reliable Time

Part of the reason that measuring time can lead to misleading results is that the hardware interface for time is complex and ambiguous. For instance, in an x86 machine there are at least six different timers, ranging from the RTC to the TSC, all with different time characteristics [16]. Even the TSC, which is most commonly used for time today, was originally intended to count processor cycles. At first, processor frequencies were constant so the counter could be used to measure time, so programmers needing high precision time used it for timing measurements. However, when technologies such as dynamic frequency scaling and idle states arrived, the cycle count diverged from time and caused programs which relied on it to misbehave. As a result, Intel changed the behavior of the timer to produce a constant frequency even while idle, without making strong guarantees about time.

Since time, and measuring time is so critical to computer science, we believe in the importance of a standardized, high resolution timer that is fed by a consistent uniform frequency across all hardware. This effort needs to be led by processor manufacturers, and the recent increase in architectural diversity has made the need for such a mechanism all the more important.

6.2 Synchronizing Time

There has been a significant effort in the systems community to synchronize clocks, typically to GPS time so that timestamps can be used across hardware and geographical boundaries [28, 31]. While calibrating clocks is similar to synchronizing them, it has several different properties. Calibrating clocks is about ensuring two clocks agree on elapsed time, whereas synchronizing clocks is about ensuring that two clocks both step at and reflect the same time. As a result, calibrating clocks is an easier problem: whereas synchronizing a clock requires constantly measuring the time error and correcting it, to calibrate a clock only requires several synchronizations to obtain a bound on the error of the local clock.

Synchronizing clocks can have several interactions with timing and calibration. In order to correct for the difference in the local clock to the remote clock, clock synchronization software must alter the timestamp to correct for any accumulated error. If a synchronization occurs while the timer is running, updating the value of the timer, this could result in a benchmark that runs slightly faster or slower than expected due to time "jumping" backwards or forwards.

For this reason, benchmarks are usually run against an uncorrected timer, but this timer does not get the benefits of the correction provided by synchronization. A hardware interface for correcting the clock transparently may enable the clock to be slowly corrected without causing the same effects as software reloading the timer. For instance, if synchronization software detects that the clock is running too fast, an interface to slow down the clock could be used to maintain continuous time.

6.3 Calibrating Other Sensors

Computer systems contain a variety of other sensors which may be more complex to calibrate. For example, the processor has temperature sensors and the motherboard has voltage monitors with varying error. Ensuring that these sensors are accurate may be much more intrusive, but may be possible if the right interface is exposed. For example, testing voltage sensors maybe be possible if a interface to an external multimeter or voltage reference is available.

6.4 The Observer Effect

The observer effect states that observing a system can alter its behavior. Best known from physics, this effect applies to systems research as well. Adding timer calls and storing timing information inevitably alters execution. The choice of which timer calls are used also affect execution, and differences in the timer used are what caused the inconsistencies leading to the "HPET bug".

Systems can reduce or eliminate the observer effect by minimizing the instrumentation overhead. For example, instead of software manually invoking a timer and saving the values to memory, A hardware based system could be used. Platforms like Intel's Precise Event Based Sampling (PEBS) [33] might provide this functionality, but a similar, more universal interface is desirable.

In addition, the system should expose information about the timing method used and its overheads to the application.

6.5 Reproducible Results

Evaluating computer systems accurately and reproducibly is difficult because they are subject to various sources of non-determinism. While calibrating clocks reduces one source of non-determinism, many other sources of non-determinism still exist. Many of these sources of non-determinism are opaque because of abstractions designed to hide detail from applications. While ensuring that every parameter is the same between experimental setups is not tractable, it might be possible to develop a set of standardized conditions. For instance, in the physical sciences, standard temperature and pressure (STP) is used to enable comparisons to be made between different sets of data. It may be possible to create a similar analog in computer systems.

6.6 Statistical Analysis

Several papers in the literature [29, 32] highlight flaws in the statistical presentation or methodology in systems research. By calibrating the clock, we enhance statistical analysis by providing the actual error of the underlying clock, enhancing the confidence of presented results. Furthermore, statistical analysis can also be used to enhance the soundness of performance evaluations [27].

6.7 Security

Interestingly, there is an active movement to obfuscate and prevent applications from reading accurate time, because of its utility as a side channel. For example, in light of Spectre [30], many browsers may round or reduce to precision of the clock to as much as 100ms [15]. Often, these changes are opaque: for example, the `performance.now()` API merely returns a timestamp, with no way to determine what the actual precision being returned is without making multiple calls. We believe that an application developer should know when the time is obscured for security reasons so a user can take appropriate action.

7 CONCLUSION

We hope that this paper highlights the need for systems researchers to calibrate timers, the primary instrument used for systems experiments. We have shown that calibrating timers is critical for comparable, repeatable system results, and that the equipment for calibrating timers is inexpensive and easily performed by any research lab. In the future, we will release a device and/or software which can help users calibrate their systems, and keep systems research running in time.

8 ACKNOWLEDGEMENTS

The authors would like to thank Nadav Amit for his comments and insights that shaped this paper, a discussion with Jeremy Elson that inspired this paper, and the helpful feedback of the HotOS reviewers.

REFERENCES

[1] [2/3] x86/tsc: Fix erroneous TSC rate on skylake xeon. https://lore.kernel.org/patchwork/patch/866472/.
[2] c15, cycle counter register (CCNT). https://developer.arm.com/documentation/ddi0360/f/control-coprocessor-cp15/register-descriptions/c15--cycle-counter-register--ccnt-.
[3] chrony. https://chrony.tuxfamily.org.
[4] excessive system clock drift? (2+ minutes per hour). https://askubuntu.com/questions/1014285/excessive-system-clock-drift-2-minutes-per-hour.
[5] Getting high precision timing on android. https://www.gamasutra.com/view/feature/171774/getting_high_precision_timing_on_.php?print=1.
[6] The HPET bug: What it is and what it isn't. https://www.overclockers.at/articles/the-hpet-bug-what-it-is-and-what-it-isnt.
[7] Ieee 1588-2019 - ieee standard for a precision clock synchronization protocol for networked measurement and control systems. https://standards.ieee.org/standard/1588-2019.html.
[8] The linux kernel disabling HPET for intel coffee lake. https://www.phoronix.com/scan.php?page=news_item&px=Linux-Disabling-HPET-CoffeeLake.
[9] Mdn docs: performance.now(). https://developer.mozilla.org/en-US/docs/Web/API/Performance/now.
[10] native_calibrate_tsc(): possibly incorrect tsc frequency on newest intel skylake-x cpus, i7-7820x in particular. https://bugzilla.kernel.org/show_bug.cgi?id=197843.
[11] ntpd - network time protocol (ntp) daemon. http://doc.ntp.org/4.1.0/ntpd.htm.
[12] [patch 2/2 v2] x86/tsc: Add additional intel cpu models to crystal_khz whitelist. https://lore.kernel.org/lkml/1474289501-31717-3-git-send-email-prarit@redhat.com/.
[13] [patch] x86/tsc: Don't use cpuid 0x16 leaf to determine cpu speed. https://lkml.org/lkml/2019/12/6/492.
[14] [patch] x86/tsc: Don't use cpuid 0x16 leaf to determine cpu speed. https://lkml.org/lkml/2019/12/5/670.
[15] performance.now(). https://developer.mozilla.org/en-US/docs/Web/API/Performance/now.
[16] Pitfalls of tsc usage. http://oliveryang.net/2015/09/pitfalls-of-TSC-usage/.
[17] Pps - pulse per second. https://www.kernel.org/doc/html/latest/driver-api/pps.html.
[18] Queryperformancecounter function (profileapi.h). https://docs.microsoft.com/en-us/windows/win32/api/profileapi/nf-profileapi-queryperformancecounter.
[19] Re: [patch] x86: Export tsc related information in sysfs. https://lwn.net/Articles/388286/.
[20] Running a database on ec2? your clock could be slowing you down. https://heap.io/blog/engineering/clocksource-aws-ec2-vdso.
[21] Smd microprocessor crystal. https://www.mouser.com/datasheet/2/3/ABLS7M2-1774905.pdf.
[22] Supermicro FAQ 21337. https://www.supermicro.com/support/faqs/faq.cfm?faq=21337.
[23] Supermicro ultra. https://www.supermicro.com/en/products/ultra/.
[24] A `tsc_freq_khz` driver for everyone. https://github.com/trailofbits/tsc_freq_khz.
[25] A timely discovery: Examining our amd 2nd gen ryzen results. https://www.anandtech.com/show/12678/a-timely-discovery-examining-amd-2nd-gen-ryzen-results.
[26] TSC frequency for all: Better profiling and benchmarking. https://blog.trailofbits.com/2019/10/03/tsc-frequency-for-all-better-profiling-and-benchmarking/.
[27] Charlie Curtsinger and Emery D. Berger. Stabilizer: Statistically sound performance evaluation. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 219–228, New York, NY, USA, 2013. Association for Computing Machinery.
[28] 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.
[29] Tomas Kalibera and Richard Jones. Quantifying performance changes with effect size confidence intervals. *arXiv preprint arXiv:2007.10899*, 2020.
[30] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.
[31] 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.
[32] Erik Van Der Kouwe, Gernot Heiser, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. Benchmarking flaws undermine security research. *IEEE Security & Privacy*, 18(3):48–57, 2020.

[33] Vincent M Weaver. Advanced hardware profiling and sampling (pebs, ibs, etc.): creating a new papi sampling interface. Technical report, Technical Report UMAINE-VMWTR-PEBS-IBS-SAMPLING-2016-08. University of Maine, 2016.