Time, but *Faster*

**A COMPUTING
ADVENTURE
ABOUT TIME
THROUGH THE
LOOKING GLASS**

THEO SCHLOSSNAGLE, CIRCONUS

Every once in a while, you find yourself in a rabbit hole, unsure of where you are or what time it might be. This article presents a computing adventure about time through the looking glass.

The first premise was summed up perfectly by the late Douglas Adams in *The Hitchhiker's Guide to the Galaxy*: "Time is an illusion. Lunchtime doubly so." The concept of time, when colliding with decoupled networks of computers that run at billions of operations per second, is... well, the truth of the matter is that you simply never really know what time it is. That is why Leslie Lamport's seminal paper on Lamport timestamps was so important to the industry, but this article is actually about wall-clock time, or a reasonably useful estimation of it.

Even on today's computers, it is feasible to execute an instruction in less than a nanosecond. When the white rabbit looks at his pocket watch in *Alice's Adventures in Wonderland*, he is seeing what time it was a nanosecond

before, as the light travels from the hands on the watch to his eye—assuming that Lewis Carroll’s timeless tale took place in a vacuum and that the rabbit was holding the watch one-third of a meter from his eyes.

When you think of a distributed system where a cluster of fast computers are often more than one light-nanosecond away from each other, it is understandably difficult to time something that starts in one place and ends in another with nanosecond precision; this is the realm of physicists, not bums like us with commodity computing hardware run in environments we don’t even manage. To upset the newcomer even further, every computer today is effectively a distributed system itself, with each CPU core having its own clock ticking away, with its own subtly different frequency and sense of universal beginning.

All that said, computers need to give users the illusion of a clock. Without it, we won’t know what time it is. As computers get faster and faster, we are able to improve the performance of our systems, but one fundamental of performance engineering is that we cannot improve what we cannot measure; so measure we must. The fundamental paradox is that as what we measure gets smaller, the cost of measuring it remains fixed, and thus becomes relatively monstrous.

THE BEGINNING OF THE TUMBLE...

At Circonus, we write a database that’s fast and keeps getting faster and more efficient. We dump energy into this seemingly endless journey because we operate at scale and every bit of efficiency we eke out results in lower COGS (cost of goods sold) for us, and better service for our

While you and I like to think in terms of seconds, computers don't care about this concept of time. They care only about clock ticks.

customers, and fundamentally affords a cost effectiveness of telemetry collection and analysis that approaches reasonable economics to “monitor all the things.” In that context...

Let's assume we want to achieve an average latency for operations of one microsecond. Now let's wrap some numbers around that. I'll make some notes about certain aspects of hardware, but I'll really focus only on hardware from the past several years. While you and I like to think in terms of seconds, computers don't care about this concept of time. They care only about clock ticks.

THE TSC

Online CPUs are forever marching forward at some frequency, and the period of this frequency is called a *tick*. In an effort to save power, many computers can shift between different power-saving states, which cause the frequency of the CPU to change. This could make it excruciatingly difficult to tell high-granularity time accurately, if the frequency of the CPU were used for timing. Each core on a modern CPU has a TSC (time-stamp counter) that counts the number of ticks that have transpired. You can read this counter with the cleverly named `rdtsc` assembly instruction. Also, modern CPUs have a feature called an invariant TSC, which guarantees that the frequency at which ticks occur will not change for any reason (but mainly for power-saving mode changes). My development box has an invariant TSC that ticks approximately 2.5999503 times per nanosecond. Other machines have different frequencies.

The standard tooling to figure out how long an

operation takes on a Unix machine is either `clock_gettime(CLOCK_MONOTONIC,...)` or `gethrtime()`. These calls return the number of nanoseconds since some arbitrary fixed point in the past. The examples shown here use `gethrtime()` because it is shorter to write.

```
hrtime_t start = gethrtime();
some_operation_worthy_of_measurement();
hrtime_t elapsed = gethrtime() - start;
```

As these things are measured, the `gethrtime()` call itself will take some time. The question to ask is: where does the time it returns sit relative to the beginning and end of the `gethrtime()` call itself? That can be answered with benchmarks. The bias introduced by measuring the start and finish is relative to its contribution to overall running time. In other words, if the “operation” being measured is made to take a long time over many iterations, the measurement bias can be reduced to near zero. Timing `gethrtime()` with `gethrtime()` would look like this:

```
#define LOTS 10000000
hrtime_t start = gethrtime();
for(int i=0;i<LOTS;i++) (void)gethrtime();
hrtime_t elapsed = gethrtime() - start;
double avg_ns_per_op = (double) elapsed /
(double)LOTS;
```

Behold, a benchmark is born. Furthermore, you could actually measure the number of ticks elapsed in the test by bracketing the test with calls to `rdtsc` in assembly. Note

that you must bind yourself to a specific CPU on the box to make this effective because the TSC clocks on different cores do not have the same concept of “beginning.”

If this is run on our two primary platforms (Linux and Illumos/OmniOS on a 24-core 2.6-GHz Intel box), then these are the results:

OPERATING SYSTEM	CALL	CALL TIME
Linux 3.10.0	gettimeofday	35.7 ns/op
Linux 3.10.0	gethrtime	119.8 ns/op
OmniOS r151014	gettimeofday	304.6 ns/op
OmniOS r151014	gethrtime	297.0 ns/op

The first observation is that Linux optimizes both of these calls significantly more than OmniOS does. This has actually been addressed as part of the LX brand work in SmartOS by Joyent and will soon be upstreamed into Illumos for general consumption by OmniOS. Alas, that isn’t the worst thing: objectively determining what time it is, is simply too slow for microsecond-level timing, even at the lower 119.8 ns/op (nanoseconds per operation) number above. Note that `gettimeofday()` supports only microsecond-level accuracy and thus is not suitable for timing faster operations.

At just 119.8 ns/op, bracketing a one-microsecond call will result in:

$$(119.8 * 2) / (1000 + 119.8 * 2) \rightarrow 19.33\%$$

So 19.33 percent of the execution time is spent on calculating the timing, and that doesn't even include the time spent recording the result. A good goal to target here is 10 percent or less. So, how do we get there?

LOOKING AT OUR TOOLS

These same modern CPUs with invariant TSCs have the `rdtsc` instruction, which reads the TSC, yet doesn't provide insight into which CPU you are executing on. That would require either prefixing the call with a `cpuid` instruction or binding the executing thread to a specific core. The former adds ticks to the work; the latter is wholly inconvenient and can really defeat any advanced NUMA (nonuniform memory access)-aware scheduling that the kernel might provide. Basically, binding the CPU provides a super-fast but overly restrictive solution. We just want the `gethrtime()` call to work and be fast.

We're not the only ones in need. Out of the generally recognized need, the `rdtscp` instruction was introduced. It supplies the value in the TSC and a programmable 32-bit value. The operating system can program this value to be the ID of the CPU, and a sufficient amount of information is emitted in a single instruction. Don't be deceived; this instruction isn't cheap and measures in at 34 ticks on this machine. If you code that instruction call as `uint64_t mtev_rdtscp(int *cpuid)`, that returns the TSC and optionally sets a `cpuid` to the programmed value.

The first challenge here is to understand the frequency. This is a straightforward timing exercise:

```
mtev_thread_bind_to_cpu(0);
hrtime_t start_ns = gethrtime();
uint64_t start_ticks = mtev_rdtscp(NULL);
sleep(10);
hrtime_t end_ns = gethrtime();
uint64_t end_ticks = mtev_rdtscp(NULL);
double ticks_per_nano = (double)(end_ticks-
start_ticks) / (double)(end_ns-start_ns);
```

The next challenge becomes quite clear when testing this solution for timing the execution of a job—even the simplest of jobs:

```
uint64_t start = mtev_rdtscp(NULL);
*some_memory = 0;
uint64_t elapsed = mtev_rdtscp(NULL) - start;
```

This usually takes around 10 ns, assuming no major page fault during the assignment—10 ns to set a piece of memory! Remember, that includes the average time of a call to `mtev_rdtscp()`, which is just over 9 ns. That's not really the problem. The problem is that sometimes we get HUGE answers. Why? We switch CPUs and the outputs of the two TSC calls are reporting to completely unrelated counters. So, to rephrase the problem: we must relate the counters.

The code for skew assessment is a bit much to include here. The basic idea is that we should run a calibration loop on each CPU that measures `TSC*ticks_per_nano` and assess the skew from `gethrtime()`, accommodating

the running time of `gethrtime()`. As with most calibration loops, the most skewed is discarded and the remaining is averaged. This basically goes back to secondary-school math to find the linear intercept equation: $y = mx + b$, or:

$$\text{gethrtime}[] = \text{ticks_per_nano} * \text{mtev_rdtscp}[] + \text{skew}$$

As the TSC is per CPU, you need to track m and b (`ticks_per_nano` and `skew`) on a per-CPU basis.

Another nuance is that these two values together describe the translation between a CPU's TSC and the system's `gethrtime()`, and they are estimates. That means two important things: (1) they need to be updated regularly to correct for error in the calibration and estimation; and (2) they need to be set and read atomically. This is where the `cmpxchg16b` instruction enters.

Additionally, this calibration work is performed every five seconds in a separate thread, and we attempt to make that thread high priority on a realtime scheduler. It turns out that this all works quite well, even without the ability to change priority or scheduling class.

GRAVY

Since we're clearly having to correct for skew to align with the system `gethrtime()`, and the point in the past to which `gethrtime()` is relative is arbitrary (according to the documentation), we've elected to make that "arbitrary" point the Unix epoch. No additional instructions are required, and now the replacement `gethrtime()` can be used to power `gettimeofday()`. Therefore, $y = mx + b$ is

actually implemented as

```
nano_second_since_epoch = ticks_per_nano * mtev_rdtscp()  
+ skew
```

Obviously, we'll pick up changes to the wall clock (via `adjtime()` et al.) only when we recalibrate.

SAFETY

Obviously, things can and do go wrong. A variety of fail-safe mechanisms are in place to ensure proper behavior when the optimizations become unsafe. By default, if the lack of an invariant TSC is detected, the system is disabled. If a calibration loop fails for too long (15 seconds), the CPU is marked as bad and disabled. During rudimentary performance tests, if the system's `gethrtime()` can beat the emulation, then we disable. If all those tests pass, we still check to see if the underlying system can perform `gettimeofday()` faster than we can emulate it; if so, we disable `gettimeofday()` emulation. The goal is for `mtev_gethrtime()` to be as fast as or faster than `gethrtime()` and for `mtev_gettimeofday()` to be as fast as or faster than `gettimeofday()`.

RESULTS

The overall results are better than expected. The original goal was simply to provide a way for our implementation on Illumos to meet the performance of Linux. The value of ZFS is deeply profound, and while Linux has some performance advantages in specific arenas, that doesn't

matter much if you have undetectable corruption of the data you store.

Further optimization is possible in the implementation, but we’ve stopped for now, having achieved the initial goals. Additionally, for the purposes of this test, we’ve built the code portably. We can find a couple of nanoseconds if we compile `-march=native` on machines supporting the AVX (Advanced Vector Extensions) instruction set.

It is true that an approximately 40-ns `gethrtime()` can be considered slow enough, relative to microsecond-level efforts, that very prudent selection is still necessary. It is also true that 40-ns `gethrtime()` can open up a new world of possibilities for user-space instrumentation. It has certainly opened our eyes to some startling things.

OPERATING SYSTEM	CALL	SYSTEM CALL TIME	MTEV-VARIANT CALL	SPEEDUP
Linux 3.10.0	<code>gettimeofday</code>	35.7 ns/op	35.7 ns/op	x1.00
Linux 3.10.0	<code>gethrtime</code>	119.8 ns/op	40.4 ns/op	x2.96
OmniOS r151014	<code>gettimeofday</code>	304.6 ns/op	47.6 ns/op	x6.40
OmniOS r151014	<code>gethrtime</code>	297.0 ns/op	39.9 ns/op	x7.44

This all comes for free with https://github.com/circonus-labs/libmtev/blob/master/src/utls/mtev_time.c [see https://github.com/circonus-labs/libmtev/blob/master/src/utls/mtev_time.c for reference]. As of this writing, Linux and Illumos are supported platforms, and Darwin and FreeBSD do not have “faster time” support. The faster time support in libmtev was a collaborative effort between Riley Berton and Theo Schlossnagle.

Theo Schlossnagle is the founder and chief executive officer at Circonus, where he applies his heart, soul, and brain to bettering the world of monitoring through large-scale numerical data analysis. A widely respected industry thought leader, Schlossnagle is the author of *Scalable Internet Architectures* (Sams Publishing, 2006) and a frequent speaker at technology conferences worldwide. He founded OmniTI, an Internet consultancy that has helped more than 10 percent of the world's largest websites. From OmniTI, he incubated Message Systems, which provides messaging infrastructure software to the world's largest Internet companies. He earned his bachelor's and master's degrees in computer science from Johns Hopkins University; additionally he spent an extended time researching distributed systems at JHU's Center for Networking and Distributed Systems. He enjoys spending time with his wife and three daughters at their home in Maryland.

Related Articles

Passively Measuring TCP Round-trip Times

Stephen D. Strowes

A close look at RTT measurements with TCP

<http://queue.acm.org/detail.cfm?id=2539132>

The One-second War (What Time Will You Die?)

Poul-Henning Kamp

As more and more systems care about time at the second and sub-second level, finding a lasting solution to the leap seconds problem is becoming increasingly urgent.

<http://queue.acm.org/detail.cfm?id=1967009>

Principles of Robust Timing over the Internet

Julien Ridoux and Darryl Veitch

The key to synchronizing clocks over networks is taming delay variability.

<http://queue.acm.org/detail.cfm?id=1773943>

Copyright © 2016 held by owner/author. Publication rights licensed to ACM.

SHAPE THE FUTURE OF COMPUTING!

We're more than computational theorists, database managers, UX mavens, coders and developers. We're on a mission to solve tomorrow. ACM gives us the resources, the access and the tools to invent the future.

Join ACM today at acm.org/join

BE CREATIVE. STAY CONNECTED. KEEP INVENTING.

Association for
Computing Machinery

Advancing Computing as a Science & Profession