

Measurement: Time

Vojtěch Horký

Peter Libič

Petr Tůma

2010 – 2023

This work is licensed under a “CC BY-NC-SA 3.0” license. Created to support the Charles University Performance Evaluation lecture. See <http://d3s.mff.cuni.cz/teaching/performance-evaluation> for details.

Contents

Clock Sources

Hardware timers

- Each (m -th) pulse of internal (hardware) clock is counted.
- Count can be read from mapped memory or registers (user-mode or via OS).

Software timers (interrupt based)

- Each (m -th) pulse of internal clock causes an interrupt.
- OS interrupt handler increments a counter.

Other clock sources may also exist, these are just the two quite common.

Clock Properties

Before using hardware clock, we must ask many questions:

- What is the clock frequency ?
- Is the clock frequency constant ?
- What is the clock counter register size ?
- Does the clock stop while the system is idle or not ?
- Is there a single clock for all processors in the system ? If not, are the clocks synchronized ?

Think about how we can compensate when the answers are not ideal.

Compensating for non ideal clock properties

If the frequency is too low for the measured interval, the measurements will return series of 0 and 1. If there are n measurements, m return time 1, $n - m$ return 0, the estimate is $m/n * period$. Bernoulli trial and binomial distribution (assuming independence between samples). Getting confidence intervals is relatively easy. If the clock counter register is too small, or does not start from low enough values, single overflow can be detected, but cannot be distinguished from multiple overflows.

Common x86 Time Sources

On x86 following hardware time sources are (typically) available:

- RTC – Real Time Clock chip. Typically runs at 32 768 Hz.
- PIT – Programmable Interval Timer – 8253/8254 chip. Typically runs at 1 193 182 Hz.
- LAPIC, APIC PM timers. Frequency of the CPU so on the order of 1 GHz.
- HPET – High Precision Event Timer – inside chipset. Frequency at least 10 MHz. Replacement of RTC and PIT chips.
- TSC – Time Stamp Counter. Frequency of the CPU so on the order of 1 GHz.
- Hardware performance monitoring counters.

See also Amsden: Timekeeping Virtualization ... <https://www.kernel.org/doc/html/latest/virt/kvm/x86/timekeeping.html>.

TSC

A (mostly) monotonously increasing counter supported by all modern x86 processors.

- Per-(logical)-processor counter.
- 64-bit register (in fact MSR register 0x10).
- Read (also from userspace – can be restricted) using RDTSC (or RDTSCP) instruction.
- Increments with each CPU tick.
- Synchronous with CPU operations.
- Most (usefully) precise time source.

Reading TSC

```
#include <stdint.h>

inline uint64_t rdtsc() {
    uint32_t lo, hi;

    __asm__ __volatile__ (
        "xorl %%eax, %%eax\n"
        "cpuid\n"

        "rdtsc\n"
        : "=a" (lo), "=d" (hi)
        :
        : "%rax", "%rbx", "%rcx", "%rdx");
    return (uint64_t) hi << 32 | lo;
}

1
```

The CPUID instruction is inserted as barrier to serialize execution but causes VM exit when virtualized.

The volatile qualifier tells the compiler outputs do not depend on inputs.

Reading TSC

On newer processors (AMD, Core i7) RDTSCP instruction support:

- Guarantees serialization.
- Also returns per-core MSR value from TSC_AUX which provides core identification in ECX.

```
asm volatile ("rdtscp"
: "=a" (time_lo), "=d" (time_hi), "=c" (cpu_id) : : );
```

Linux kernel sets TSC_AUX during initialization. Lower bits indicate processor core, upper bits indicate processor package.

TSC Problems

Frequency Scaling

What to do when processor frequency changes ?

¹Based on code from Wikipedia

- On older CPUs, TSC ticks at scaled frequency (cannot be used as wall clock).
- On newer CPUs, the TSC may tick with constant frequency (cannot be used to count processor ticks) but may still stop in deep sleep states.
- Check for TSC invariant bit in Intel CPUID.
- It is (sometimes) possible to turn off the frequency scaling.

Multiprocessors

What to do when there are more TSC registers ?

- TSC registers of different processors may not be in sync. Frequency scaling may contribute to this effect.
- Many systems maintain TSC in sync.

Firmware

The TSC register can be written to by system software.

- Hypervisor may virtualize TSC.
- Some BIOS implementations hide SMM execution. Can be detected with TSC_ADJUST MSR.

Out Of Order Execution

How is RDTSC ordered with other instructions ?

- RDTSC is only self serializing but not serializing.
- Some code uses CPUID before RDTSC to force serialization.
- Newer processors have RDTSCP.
- Measurements where CPUID would matter are too short to be measured with RDTSC reliably anyway.

Asynchronous Signalling

What if RDTSC overflows ?

- No interrupts generated on overflow.
- Still can be used to measure intervals between otherwise generated interrupts.
- Hardware performance counters can be used if interrupts needed.

See also Yang: Pitfalls of TSC Usage ... <http://oliveryang.net/2015/09/pitfalls-of-tsc-usage>.

Calibrating TSC

Interested in how TSC timestamp is converted to wall clock time ? See the Linux kernel sources at arch/x86/kernel/tsc.c (online <https://elixir.bootlin.com/linux/latest/source/arch/x86/kernel/tsc.c>). The two essential methods are querying the processor information available with CPUID and the MSR registers, and calibrating the frequency against other known timer sources.

Configuring TSC

The TSD bit of register CR4 can be used to disable user mode TSC access.

The virtual machine control structure (VMCS) includes the RDTSC exiting bit, which can be used to restrict guest mode TSC access, and the TSC offset and TSC multiplier fields, which can be used to offset and scale the guest mode TSC.

Adjusting TSC

The TSC register can be written to using WRMSR, but adjusting it accurately while the clock ticks is tricky. Modern processors therefore permit adjusting the TSC register by writing to the TSC_ADJUST register. Whatever value is added to or subtracted from TSC_ADJUST is also added to or subtracted from TSC, and also vice versa. This makes it possible to detect external TSC adjustment, for example through SMM.

MPERF and APERF

More recent x86 processors offer per-(logical)-processor MSR registers called MPERF and APERF. These are intended to provide feedback on hardware power management behavior, with MPERF ticking at a fixed frequency and APERF ticking in proportion to actual hardware performance. The ratio of APERF to MPERF indicates actual processor speed, which can exceed one when boosting.

AMD provides the RDPRU instruction to read the two registers from user mode.

Windows Timers

Wall clock time:

- `GetSystemTimePreciseAsFileTime`
- `GetTickCount64`

Process accounting:

- `GetProcessTimes`

Performance evaluation:

- `QueryPerformanceCounter`

Wall Clock Time Sources in Windows

- `GetTickCount64`
 - Milliseconds since system start
 - Resolution in tens of milliseconds
- `GetSystemTimePreciseAsFileTime`
 - Returns current date and time, in UTC
 - “Highest possible resolution (under 1 μ s)”

Process Accounting in Windows

- `GetProcessTimes`
 - Returns creation and exit time of a process; amount of time spent in kernel and user mode.
 - Creation and exit time: amount of time since 1.1.1601
 - Kernel and user time: 100 ns units

Performance Evaluation (Windows)

- `QueryPerformanceCounter`
 - Returns internal ticks, counted from boot
 - Conversion to seconds: `QueryPerformanceFrequency ()`
 - “High-resolution (under 1 μ s)”
 - Resorts to the most precise time source
 - Supposed to work well even in virtualized environments

```
QueryPerformanceFrequency (&freq);
QueryPerformanceCounter (&ts1);
// ...
QueryPerformanceCounter (&ts2);

// Difference in seconds:
(ts2.QuadPart - ts1.QuadPart) / freq.QuadPart
```

Linux Timers

Wall clock time:

- `time`
- `ftime`
- `gettimeofday`
- Linux RTC clock

Process accounting:

- `clock`

- times

Performance evaluation:

- clock_gettime

Wall Clock Time Sources in Linux

- ftime function (obsolete, use time / gettimeofday):
 - Returns system time since Epoch, in struct timeb (s, ms, timezone).
 - Units are ms, resolution undefined.
- time function:
 - Returns system time since Epoch, in time_t (s, signed int).
 - Units are seconds, resolution undefined.
- gettimeofday function:
 - Returns system time since Epoch, in struct timeval (μ s, s).
 - Units are μ s, resolution undefined.
- Linux RTC:
 - Returns value of real-time clock, returns YYYY-MM-DD HH:MM:SS, can generate interrupts.
 - Direct driver access, via ioctl, see man 4 rtc.

Process Accounting in Linux

- clock function:
 - Returns processor ticks consumed by current process, in clock_t (signed int).
 - Units are CLOCKS_PER_SEC, 1000000 for POSIX, resolution undefined.
 - Arbitrary start value.
 - Can overflow (32-bit int: after ~ 72 minutes).
- times function:
 - Returns system ticks, user ticks consumed by current process and its children, in struct tms (four clock_t values - user/system, own/children).
 - Units are clock ticks, sysconf (_SC_CLK_TCK) to get ticks per second, resolution undefined.
 - Arbitrary start value.
 - Can overflow.

Performance Evaluation (Linux)

- Linux clock_gettime function:
 - Returns value of specified system clock, multiple supported:
 - CLOCK_REALTIME - epoch clock.
 - CLOCK_REALTIME_COARSE - faster but with smaller precision.
 - CLOCK_MONOTONIC - arbitrary start, can be changed via NTP or adjtime().
 - CLOCK_MONOTONIC_COARSE - faster but with smaller precision.
 - CLOCK_MONOTONIC_RAW - unaffected by NTP or adjtime().
 - CLOCK_BOOTTIME - like monotonic, but includes suspended time.
 - CLOCK_PROCESS_CPUTIME_ID - per-process CPU time.
 - CLOCK_THREAD_CPUTIME_ID - per-thread CPU time.
 - Return struct timespec (s, ns).
 - Resolution is returned by clock_getres function.

- Linux clock_gettime internals:

Reading timers may generally require system calls. Current implementations try to avoid that.

- System call can be delegated to VDSO module.
- The VDSO module reads time from variables exported by kernel.
- Time exported by kernel can be interpolated using TSC or other sources.

More issues:

- CPUTIME can fail with task migration. Can be checked by clock_getcpuclockid (0).
- Actual TSC frequency measured on boot.

VDSO Call Optimization

```

notrace int __vdso_clock_gettime (clockid_t clock, struct timespec *ts) {
    switch (clock) {
        case CLOCK_REALTIME:
            if (do_realtime (ts) == VCLOCK_NONE) goto fallback;
            break;
        ...
    }
    return 0;
}

notrace static void do_monotonic_coarse (struct timespec *ts) {
    unsigned long seq;
    do {
        seq = gtod_read_begin (gtod);
        ts->tv_sec = gtod->monotonic_time_coarse_sec;
        ts->tv_nsec = gtod->monotonic_time_coarse_nsec;
    } while (unlikely (gtod_read_retry (gtod, seq)));
}

notrace static int __always_inline do_monotonic (struct timespec *ts) {
    unsigned long seq;
    u64 ns;
    int mode;

    do {
        seq = gtod_read_begin (gtod);
        mode = gtod->vclock_mode;
        ts->tv_sec = gtod->monotonic_time_sec;
        ns = gtod->monotonic_time_snsec;
        ns += vgetsns (&mode);
        ns >>= gtod->shift;
    } while (unlikely (gtod_read_retry (gtod, seq)));

    ts->tv_sec += __iter_div_u64_rem (ns, NSEC_PER_SEC, &ns);
    ts->tv_nsec = ns;

    return mode;
}

notrace static inline u64 vgetsns (int *mode) {
    u64 v;
    cycles_t cycles;

    if (gtod->vclock_mode == VCLOCK_TSC) cycles = vread_tsc ();
    else if (gtod->vclock_mode == VCLOCK_PVCLOCK) cycles = vread_pvclock (mode);
    else return 0;
    v = (cycles - gtod->cycle_last) & gtod->mask;
    return v * gtod->mult;
}

notrace static cycle_t vread_tsc (void) {
    cycle_t ret = (cycle_t) rdtsc_ordered ();
    u64 last = gtod->cycle_last;
    if (likely (ret >= last)) return ret;
    return last;
}

static __always_inline unsigned long long rdtsc_ordered (void) {
    alternative_2 ("",
        "mfence", X86_FEATURE_MFENCE_RDTSC,
        "lfence", X86_FEATURE_LFENCE_RDTSC);
    return rdtsc ();
}

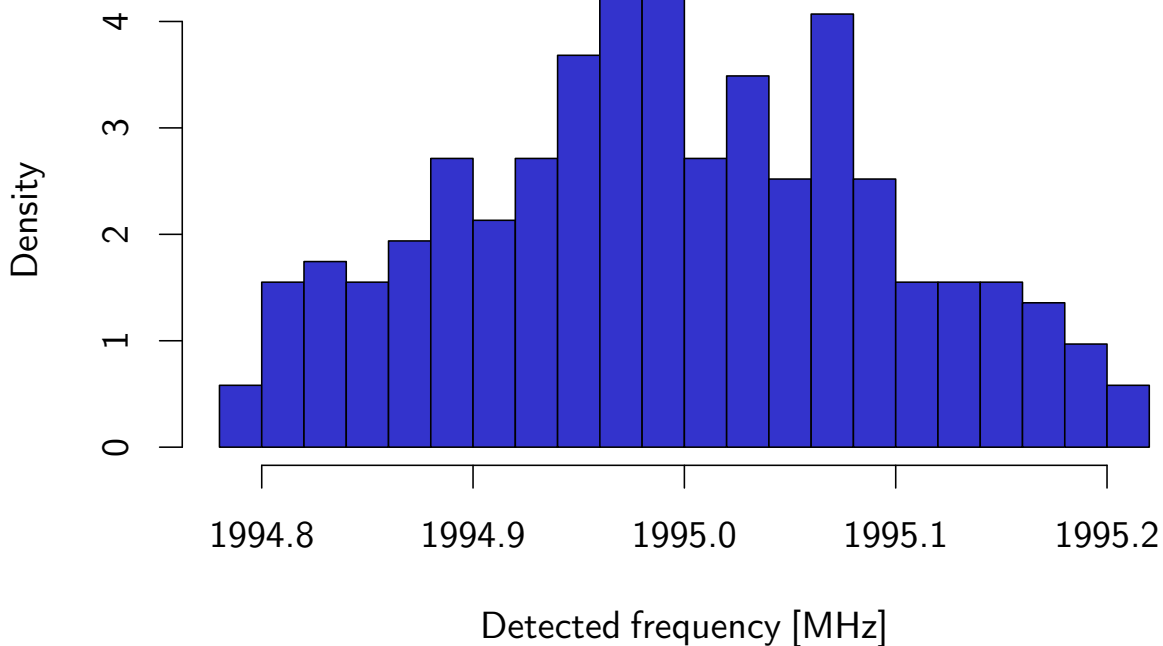
static __always_inline unsigned long long rdtsc (void) {
    DECLARE_ARGS (val, low, high);
    asm volatile ("rdtsc" : EAX_EDX_RET (val, low, high));
    return EAX_EDX_VAL (val, low, high);
}
2

```

TSC Frequency Calibration

²Based on code from Linux Kernel 4.9.13

Frequency detection on Intel Core 2 DUO processor



Java Timers

In Java, these three methods are available:

- `java.lang.System.currentTimeMillis` method.
 - Returns value of system time since Epoch, in long.
 - Units are ms, resolution undefined.
- `java.lang.System.nanoTime` method.
 - Returns value of highest resolution system clock, in long.
 - Units are ns, resolution undefined.
 - Arbitrary start value, can overflow.
 - Platform specific issues:
 - Possibly involves synchronization (Solaris ?).
 - Possibly low resolution (Java 1.5 on the order of 1 μ s).
 - Possibly adjusted by NTP (Linux `CLOCK_MONOTONIC`).
- `java.lang.management.ThreadMXBean` class.
 - `getCurrentThreadCpuTime` method. Returns total CPU time of current thread (user and kernel).
 - `getCurrentThreadUserTime` method. Returns total CPU time of current thread executed in user mode.
 - Returns long, units are ns.
 - Resolution undefined, but usually low – ms.
 - Can throw an exception if not implemented.

Evaluating timing sources

For some code examples that evaluate the timing sources see <https://github.com/d-iii-s/teaching-performance-evaluation/tree/master/src/measurement-time-java>, <https://github.com/d-iii-s/teaching-performance-evaluation/tree/master/src/measurement-time-rdtsc>, and <https://github.com/d-iii-s/teaching-performance-evaluation/tree/master/src/measurement-time-sleep>. You may want to read an article discussing the `java.lang.System.nanoTime` () behavior by Shipilev: Nanotrusting the Nanotime ... <https://shipilev.net/blog/2014/nanotrusting-nanotime>.

You may want to read a paper discussing potential software timer issues by Kuperberg et al.: TimerMeter ... doi:10.1109/QEST.2009.49.

Time sources against security

Many security exploits require the availability of accurate high resolution timers. A paper by Schwarz et al.: Fantastic Timers and Where to Find Them ... doi:10.1007/978-3-319-70972-7_13 discusses a number of ways to obtain accurate timing information in JavaScript despite browser countermeasures. In addition to using busy loops coupled with low resolution timing information or concurrent workers, the paper discusses some quite amusing examples, such as reading animated properties in CSS. More recent papers add other examples, such as using the timing of DNS queries, or even simply querying a remote server for time information.

Time in network

Network measurement needs synchronized clock. Without specialized hardware that is difficult.

... let us see what is usually available.

Time in network: clockdiff

System command that measures difference between clocks of two systems.

- Uses ICMP timestamp messages.
- Timestamps in milliseconds.

Stratum 1 means node connected directly to reference clock source.

Assumes multiple time sources as servers. Estimates time of each server as an interval. Then finds interval that intersects with most server intervals. Adjusts clock speed so that local time is in the middle of this interval.

Time in network: NTP

Protocol that synchronizes clocks of multiple systems.

- Internet wide deployment.
- Precision usually in 10 millisecond range.
- Precision in local area networks in 1 millisecond range.
- Timestamps currently 32 bit seconds and 32 bit fraction.

Much depends on configuration and environment.

- Absolute precision hinted by stratum value.
- Typical configuration with client queries.
- Server broadcasts also available.
- Some hardware support available.

Clients observe broadcasts and select best master clock available. Selection is based on priority and accuracy. Roundtrip to server measured occasionally.

Time in network: PTP

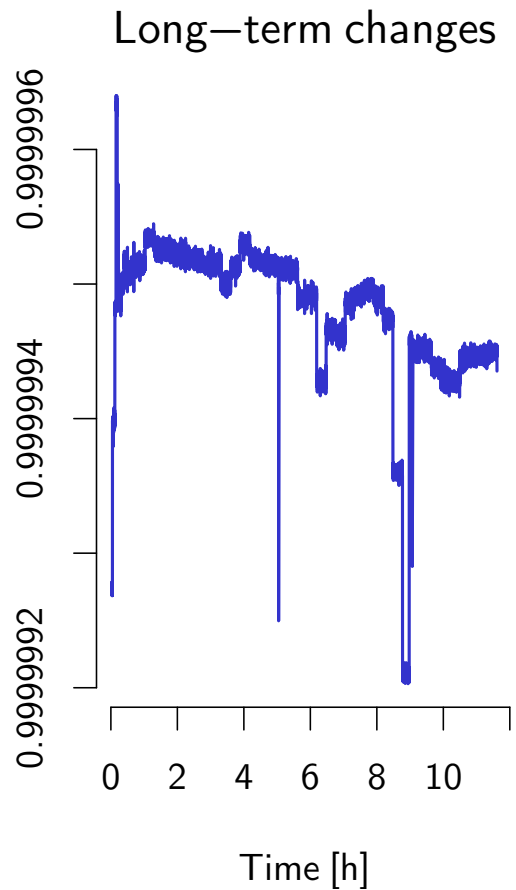
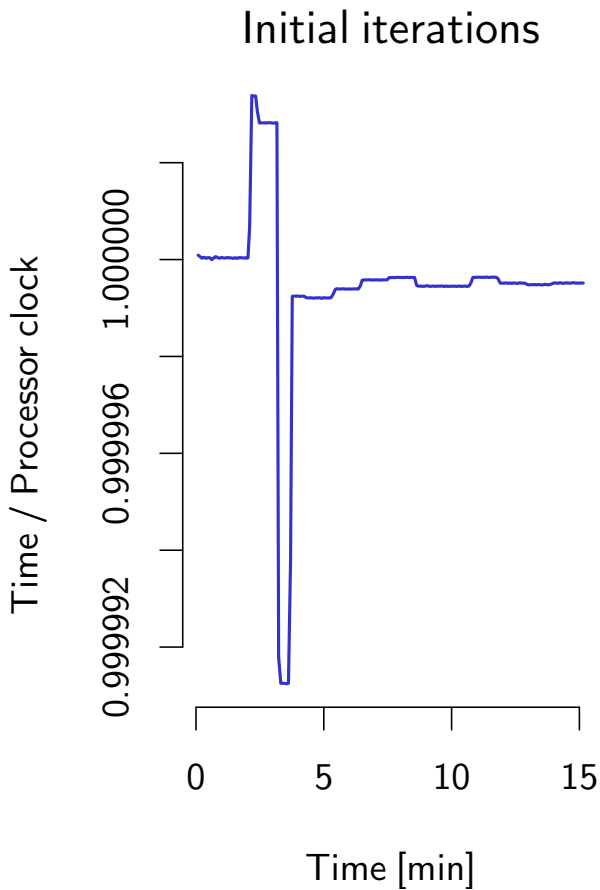
Protocol that synchronizes clocks of multiple systems.

- Local deployment.
- Precision can reach 1 microsecond range.

Much depends on configuration and environment.

- Typical configuration with server broadcasts.
- Hardware support with modern cards available.
- Can send timestamp later if no hardware support.

NTP Time Adjustment



NTP

Curious about NTP polling interval? See for example <https://stackoverflow.com/a/45244298/345436>. A running NTP client can usually inform about the next polling interval. The poll value is a binary logarithm of the next polling interval in seconds, usually between 6 and 10 for a range of 64 s to 1024 s.

If you are running the chrony daemon.
chronyc sources

MS Name/IP address	Stratum	Poll	Reach	LastRx	Last sample
=====					
^- pyrrha.fi.muni.cz	2	8	377	66	+52us[+68us] +/- 33ms
^- argus.finaltek.net	2	6	377	17	-493us[-473us] +/- 2399us
^- time.cloudflare.com	3	8	377	148	-900us[-848us] +/- 9446us
^* lx.ujf.cas.cz	1	7	377	5	+180ns[+22us] +/- 238us

Time in network: GPS

Some GPS receivers provide precise time information.

- Reported to achieve microsecond to nanosecond precision.
- But it is usually difficult to get signal in server room.