Measurement: Time

Performance Evaluation of Computer Systems

Vojtěch Horký Peter Libič Petr Tůma

Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics
Charles University

2010 – 2023

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.

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.

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;
}
```

Based on code from Wikipedia

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) : : );
```

TSC Problems I

Frequency Scaling

What to do when processor frequency changes ?

- 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.

TSC Problems II

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.

TSC Problems III

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.

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 µs)”

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 µs)"
  - Resorts to the most precise time source
  - Supposed to work well even in vitualized environements

```
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` (µs, s).
  - Units are µs, 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) I

- 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.

Performance Evaluation (Linux) II

- 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 I

```
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)));
}
```

VDSO Call Optimization II

```
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;
}
```

VDSO Call Optimization III

```
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;
}
```

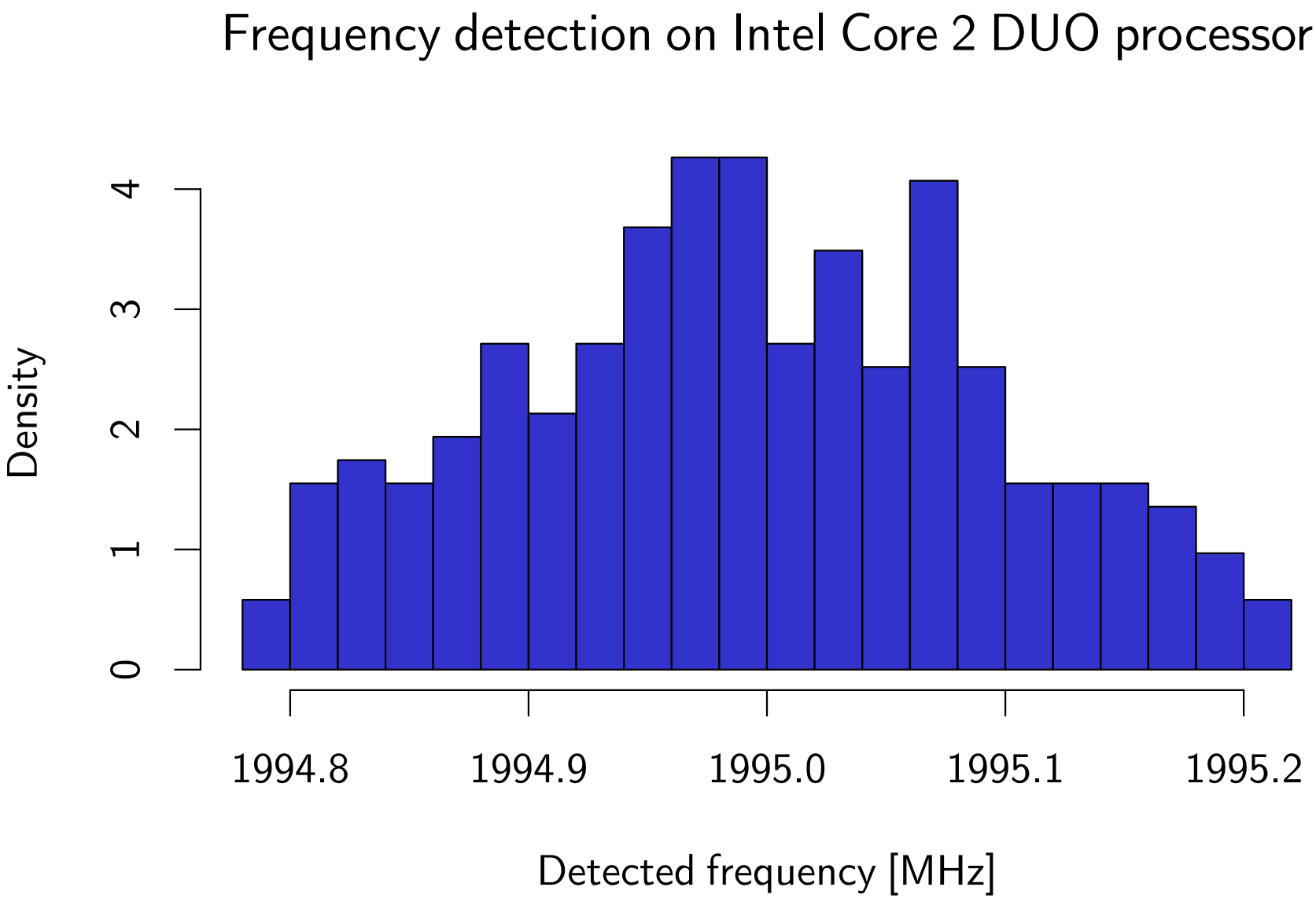
VDSO Call Optimization IV

```
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);
}
```

Based on code from Linux Kernel 4.9.13

TSC Frequency Callibration



Java Timers I

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 µs).
    - ★ Possibly adjusted by NTP (Linux `CLOCK_MONOTONIC`).

Java Timers II

- `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.

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.

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.

Time in network: PTP

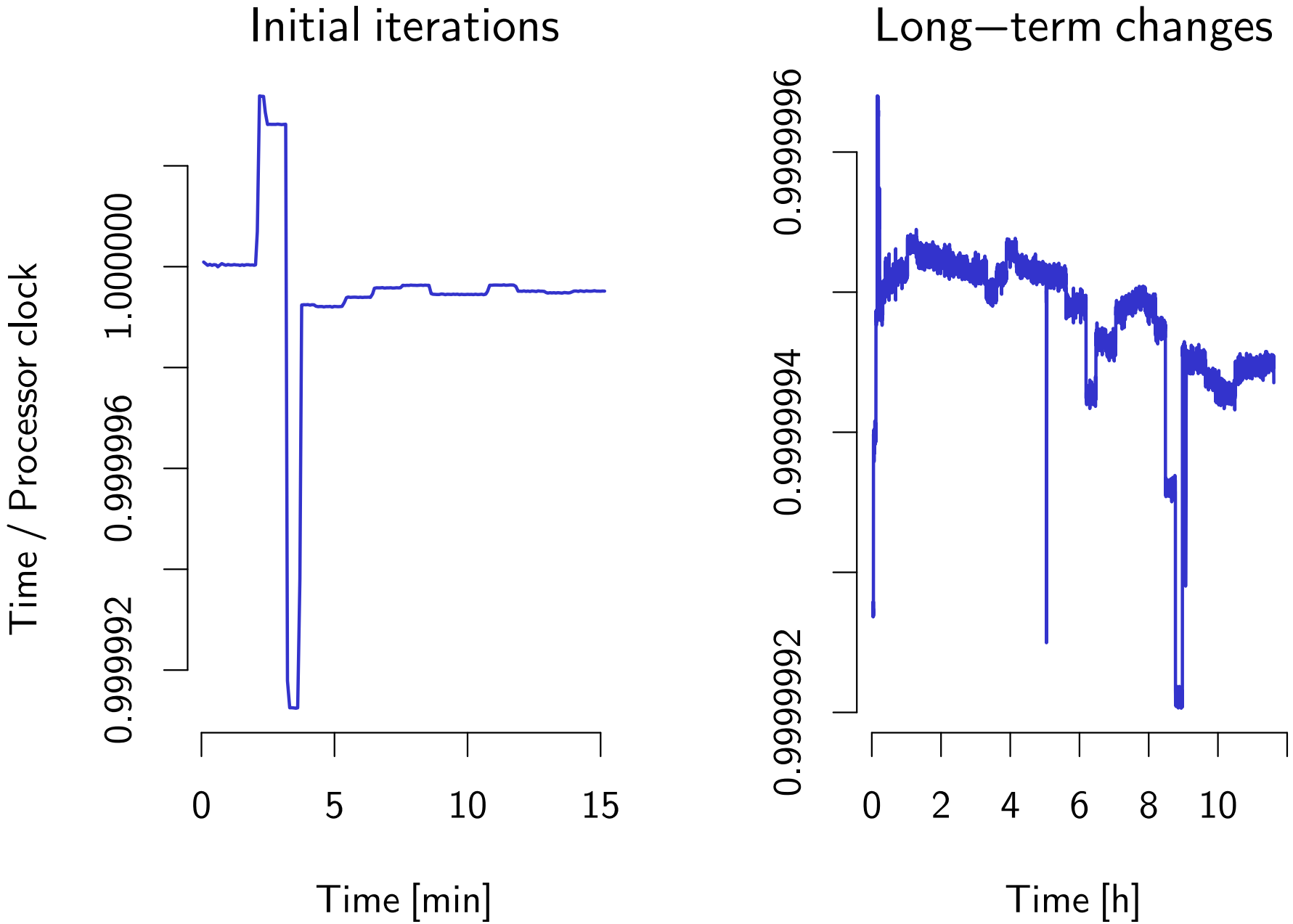
Protocol that synchronizes clocks of multiple systems.

- 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



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.