FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Clock Synchronization for Modern Multiprocessors

André dos Santos Oliveira

PREPARAÇÃO DA DISSERTAÇÃO

PREPARAÇÃO DA DISSERTAÇÃO

Orientador: Pedro Alexandre Guimarães Lobo Ferreira Souto

February 18, 2015

Resumo

Com os circuitos integrados a ficarem cada vez mais complexos, torna-se cada vez mais difícil fornecer sinais de relógio precisos e sincronizados para coordenar as ações em todos os componentes. A frequência deste sinal, embora seja um fator muito importante, parece estar a saturar devido às penalizações excessivas de dissipação de energia. O aumento da frequência como principal melhoramento de performance chegou ao fim e os fabricantes estão a mudar para arquiteturas *multi-core*, em que múltiplos processadores (cores) cooperam entre si através de hardware partilhado. Estes chips tornam a computação mais eficiente, explorando a computação paralela: o aproveitamento de múltiplos processadores para trabalhar numa única tarefa, e explorar esse paralelismo é um dos desafios pendentes de ciência da computação moderna. Atualmente, processadores *multi-core* já são habituais para os servidores, bem como computadores pessoais, e o número de núcleos está projetado para crescer à medida que entramos na era das arquiteturas *many-core*.

Esta dissertação, caracteriza a noção de tempo em qualquer sistema digital síncrono, desde a distribuição e análise do sinal de relógio para coordenar as interfaces de dados, a métodos para a manutenção precisa e síncrona de uma base de tempo numa pluralidade de processadores interligados, concentrando-se em um dos maiores problemas em sistemas distribuídos que é a sincronização.

Abstract

As ICs become more complex, the problem of supplying accurate and synchronized clocks to coordinate actions in all the components becomes increasingly difficult. The frequency of this signal, although very important, seems to be saturating due to excessive power dissipation penalties. Instead, manufacturers are turning to *multi-core* architectures, in which multiple processors (cores) communicate directly through shared hardware. Multiprocessor chips make computing more effective by exploiting parallelism: harnessing multiple processors to work on a single task, and exploiting such parallelism is one of the outstanding challenges of modern computer science. Currently, *multi-core* processors are the norm for servers as well as desktops and laptops, and the number of cores is projected to grow as we enter the era of *many-core* computing.

This dissertation, characterizes the notion of time in any synchronous digital system, from the distribution and analysis of the clock signal to coordinate data paths, to methods for maintaining a time base accurately and synchronously across a plurality of interconnected processors, focusing on one of the biggest problems in distributed systems that is synchronization.

Contents

List of Figures

List of Tables

Abbreviations and Symbols

API	Application Programming Interface
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
FLOPS	FLoating-point Operations Per Second
FTA	Fault-Tolerant Average
GALS	Globally Asynchronous Locally Synchronous
HRT	High Resolution Timer
IC	Integrated circuit
KILL	Kill If Less than Linear
MIC	Many Integrated Core
MPCP	Multiprocessor Priority Ceiling Protocol
NOC	Network On Chip
NTP	Network Time Protocol
PIT	Programmable Interrupt Timer
PLL	Phase-Locked Loop
POD	Point-Of-Divergence
RBS	Reference Broadcast Synchronization
RT	Real-Time
RTC	Real Time Clock
RTL	Register-Transfer Level
SCA	Synchronous Clocking Area
SMP	Symmetric Multiprocessor System
SOC	System On Chip
SRP	Stack-based Resource Policy
TSC	Time Stamp Counter
UTC	Universal Time Coordinated
VLSI	Very Large Scale Integration

Chapter 1

Introduction

1.1 Contextualisation

The use of computer systems are greatly increasing in real-time applications. In these systems, the operations performed must deliver results on time, otherwise we may have effects of quality reduction, or even disastrous. Designing a real-time system requires good control of the time that operations take to run. This time is determined by many factors and it is naturally intended to be minimized.

Clock frequency is one of those and the major attribute of any microprocessor design, but for a number of applications, it is only loosely correlated with performance along with other design aspects such as memory system, parallelism, and hardware acceleration.

In order to introduce parallelism in performing tasks, the concept of multicore/multiprocessor was born, a computer system that contains two or more processing units that share memory and peripherals to process simultaneously. These systems are increasingly relevant in real-time (RT) applications and clock synchronization is a critical service on many of these systems.

1.2 Motivation and Goals

Clock synchronization is commonly assumed as a given or even to be perfect in multiprocessor platforms. In reality, even in systems with a shared slock source, there is an upper bound on the precision with which clocks on different processors can be read.

If we are to take advantage of a clock service in a real-time system on a multi-processor platform, it is critical to be able to quantify the quality of that service.

In this report we set to study the literature and the state of the art related to clock synchronization. Our approach is bottom up: we start by reviewing the clock signal distribution issues at the level of the HW, next we review the various time management services available at the Linux kernel and finally we present some clock synchronization algorithms used in distributed systems.

In the Dissertation that will follow, we propose to estimate the quality of a clock service provided by Linux on common-of-the-shelf multicore processors, such as those of the x64 architecture.

1.3 Document Structure

Chapter 2

State of the art

This chapter presents some relevant state-of-art information related to the hardware trends in clock distribution networks, the software time services the keep track of time in processes level, and some clock synchronization algorithms present in today's distributed systems.

2.1 Clock Source Distribution

2.1.1 Introduction

The clock signal is used to define a time reference for the movement of data within a synchronous digital system, hence it is a vital signal to its operation [2].

Utilized like a metronome to coordinate actions, this signal oscillates between a high and a low state, typically loaded with the greatest fanout, travel over the longest distances, and operate at the highest speeds of any signal.

The data signals are provided and sampled with a temporal reference by the clock signals, so the clock waveforms must be particularly clean and sharp, and any differences in the delay of these signals must be controlled in order to limit as little as possible the maximum performance as well as to not create catastrophic race condition in which an incorrect data signal propagates within a register (latch or flip-flop).

In order to address these design challenges successfully, it is necessary to understand the fundamental clocking requirements, key design parameters that affect clock performance, different clock distribution topologies and their trade-offs, and design techniques needed to overcome certain limitations.

2.1.2 Synchronous Systems

A digital synchronous circuit is composed of a network of functional logic elements and globally clocked registers.

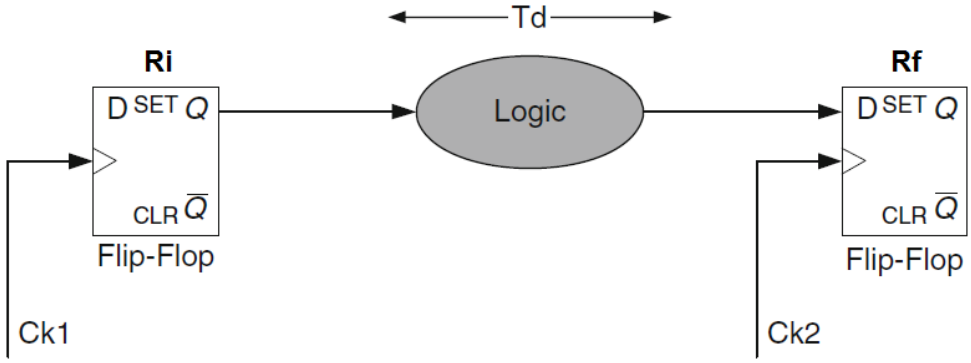

Figure 2.1: Local Data Path.

If an arbitrary pair of registers are connected by at least one sequence of logic elements, a switch event at the output of R1 will propagate to the input of R_2. In this case, (R_1,R_2) is called a sequentially-adjacent pair of registers which make up a local data path.

An example of a local data path R_i - R_f is shown in figure 2.1. The clock signals Ck_1 and Ck_2 synchronize the sequentially-adjacent pair of registers R_i and R_f, respectively. Signal switching at the output of R_i is triggered by the arrival of the clock signal Ck_1, and after propagating through the *Logic* block, this signal will appear at the input of R_f.

In order to the switch of the output of R_i to be sampled properly in the input of R_f in the next clock period, data path has to have time to stabilize the result of the combinational logic in the input of R_2, so the minimum allowable clock period $T_{CP}(min)$ between any two registers in a sequential data path is given by equation 2.1

$$\frac{1}{f_{clkMAX}} = T_{PD(max)} + T_{Skew} \tag{2.1}$$

where $T_{PD(max)} = T_{C-Q} + T_d + T_{int} + T_{setup}$.

The total path delay of the data path $T_{PD(max)}$ is the sum of the maximum propagation delay of the flip-flop, T_{C-Q}, the time necessary to propagate through the logic and interconnect, T_d + T_{int}, and the setup time of the output flip-flop, T_{setup}, which is the time that the data to be latched must be stable before the clock transition.

The clock skew $T_{Skew_{ij}}$ can be positive or negative depending on whether Ck_1 leads or lags Ck_2, respectively, as shown in figure 2.2.

2.1.3 Clock Skew

The propagation delay from the clock source to the jth clocked register is the clock delay, Tc_j.

The clock delays of the initial clock signal Tc_i and the final clock signal Tc_f define the time reference when the data signals begin to leave their respective registers.

The difference in clock signal arrival time (clock delay from the clock source) between two sequentially-adjacent registers R_i and R_f is called Clock Skew.

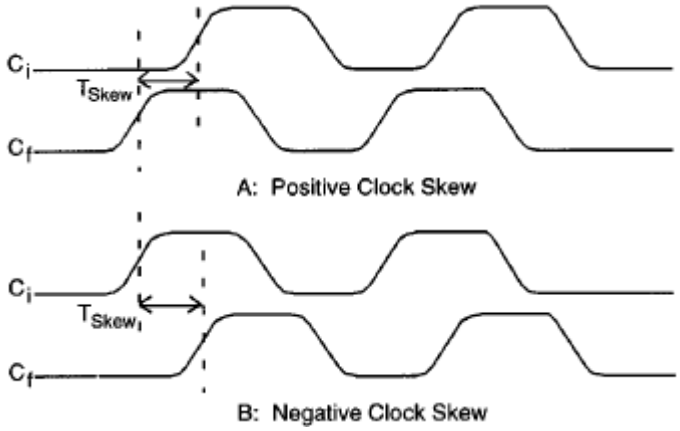

Figure 2.2: Positive and Negative clock Skew.

The temporal skew between different clock signal arrival times is only relevant to sequentially-adjacent registers making up a single data path, as shown in figure 2.1. Thus, system-wide (or chip-wide) clock skew between non-sequentially connected registers, from an analysis viewpoint, has no effect on the performance and reliability of the synchronous system.

Synchronous circuits may be simplified to have two timing limitations: setup (MAX delay) and hold (MIN delay) [1].

Setup specifies whether the digital signal from one stage of the sequential structure has sufficient time to travel to and "set-up" before being captured by the next stage of the sequential structure.

Hold specifies whether the digital signal from the current state within a sequential structure is immune from contamination by a signal from a future state due to a fast path.

The setup constraint specifies how data from the source sequential stage at cycle N can be captured reliably at the destination sequential stage at cycle N+1.

The constraint for the source data to be reliably received is defined by equation 2.2.

$$T_{per} \geq T_{d-slow} + T_{su} + \mid T_{Ck1} - T_{Ck2} \mid \tag{2.2}$$

where T_{per} is the clock period, T_{d-slow} is the slowest (maximum) data path delay, T_{su} is the setup time for the receiver flip-flop, T_{Ck1} and T_{Ck2} are the arrival times for clocks Ck_1 and Ck_2 (at cycle N) respectively.

In this situation, the available time for data propagation is reduced by the clock uncertainty defined as the absolute difference of the clock arrival times.

In order to meet the inequality, either clock period must be extended or path delay must be reduced. In either case, power and operating frequency may be affected.

The hold constraint specifies the situation where the data propagation delay is fast, and clock uncertainty makes the problem even worse and the data intended to be captured at cycle N +1 is erroneously captured at cycle N, corrupting the receiver state.

In order to ensure that the hold constraint is not violated, the design has to guarantee that the minimum data propagation delay is sufficiently long to satisfy the inequality 2.3.

$$T_{d-fast} \geq T_{hold} + \mid T_{Ck1} - T_{Ck2} \mid \tag{2.3}$$

where T_{hold} is the hold time requirement for the receive flip-flop.

In conclusion, the relationship in 2.4 is expected to hold.

$$T_{d-fast} < T_{d-nominal} < T_{d-slow} \tag{2.4}$$

Localized clock skew can be used to improve synchronous performance by providing more time for the critical worst case data paths.

By forcing Ck_1 to lead Ck_2 at each critical local data path, excess time is shifted from the neighboring less critical local data paths to the critical local data paths.

Negative clock skew subtracts from the logic path delay, thereby decreasing the minimum clock period. Thus, applying negative clock skew, in effect, increases the total time that a given critical data path has to accomplish its functional requirements by giving the data signal released from R_i more time to propagate.

2.1.4 Clock Distribution Network Topologies

Distributing a tightly controlled clock signal within specific temporal bounds is difficult and problematic.

The design methodology and structural topology of the clock distribution network should be considered in the development of a system for distributing the clock signals. Furthermore, the trade-offs that exist among system speed, physical die area, and power dissipation are greatly affected by the clock distribution network. Intentional or unintentional structural design mismatches could lead to clock uncertainties, which can be corrected by careful pre-silicon analysis and design or post-silicon adaptive compensation techniques. Therefore, various clock distribution strategies have been developed over the years.

The trend nowadays is to the adoption of clock distribution topologies that are skew tolerant, more robust design flow, and the incorporation of robust post-silicon compensation techniques, as well as multi-clock domain distributions, with the concept of design called globally asynchronous and locally synchronous (GALS).

Table 2.1 lists distribution topologies encountered in modern processors.

2.1.4.1 Unconstrained Tree

A very common strategy for distributing clock signals used in the history of VLSI-based systems was to insert buffers at the clock source and along a clock path, forming a tree structure.

The clock source is frequently described as the root of the tree, the initial portion of the tree as the trunk, individual paths driving each register as the branches, and the registers being driven

Table 2.1: Clock distribution topologies.

Style	Description
Unconstrained Tree	Automated buffer placements with unconstrained trees
Balanced tree	Multiple levels of balanced tree segments H-tree is most common
Central spine	Central clock driver
Spines with matched branches	Multiple central structures with length (or delay) matched branches
Grid	Interconnected (shorted) clock structure
Hybrid distribution	Combination of multiple techniques Common theme is tree + grid or spine + grid

as the leaves. The distributed buffers serve the double function of amplifying the clock signals degraded and isolating the local clock nets from upstream load impedances.

In the unconstrained tree clock network, there is little or none constraints imposed on the network's geometry, number of buffers or wire lengths. It is typically accomplished by automatic RTL synthesis flow tools with a cost heuristic algorithm that minimizes the delay differences across all clock branches.

But due to limitations regarding process parameter variations, this style is usually used for small blocks within large designs.

2.1.4.2 Balanced Trees

Another approach is to use a structural symmetric tree with identical distributed interconnect and buffers from the root of the distribution to all branches. This design ensures zero structural skew, hence the delay differences among the signal paths is due to variations of process parameters affecting.

Figure 2.3 shows the alternative balanced tree topologies: the tapered H-tree, the X-tree and the binary tree. The trunk widths in a tapered H-tree increase geometrically toward the root of the distribution to maintain impedance matching at the T-junctions.

Full balanced tree topologies are designed to span the entire die in both the horizontal and vertical dimensions. Binary tree on the other hand is intended to deliver the clock in a balanced manner in either the vertical or horizontal dimension.

Since the buffers in a binary tree are physically closer to each other, resulting in a reduced sensitivity to on-die variations, and H-tree and X-tree clock distribution networks are difficult to achieve in VLSI-based systems which are irregular in nature, binary trees are often the preferred structure over an idealized H-tree.

2.1.4.3 Central Spines

A central spine clock distribution is a specific implementation of a binary tree.

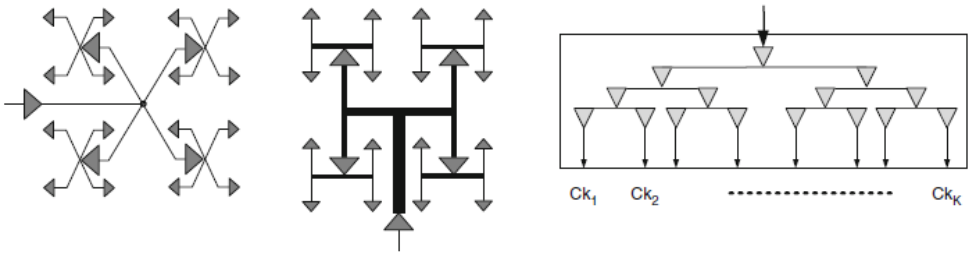

Figure 2.3: Variations on the balanced tree topology.

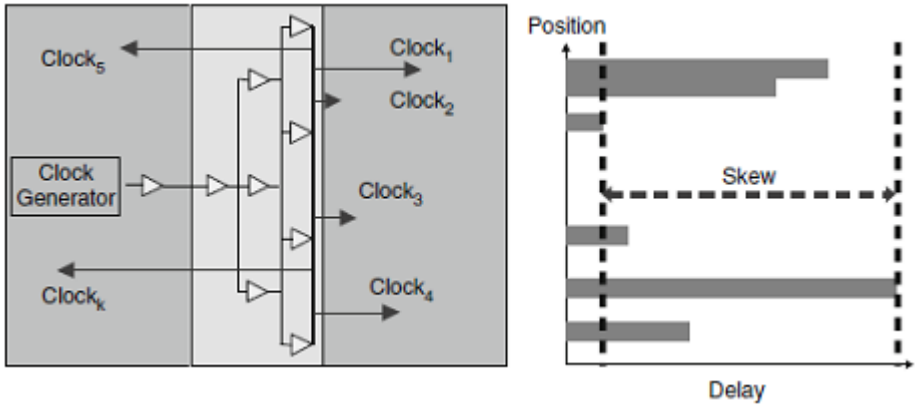

Figure 2.4: Central clock spine distribution.

The binary tree is shown to have embedded shorting at all distribution levels and unconstrained routing to the local loads at the final branches. In this configuration, the clock can be transported in a balanced fashion across one dimension of the die with low structural skew. The unconstrained branches are simple to implement although there will be residual skew due to asymmetry, as the figure 2.4 shows.

Multiple central spines can be placed to overcome this issue, dividing the chip into several sectors to ensure small local branch delays.

2.1.4.4 Grid

A processor will have a large number of individual branches to deliver the clock to the local points, and therefore a deep distribution tree is needed, degrading the clock performance. A superior solution can be subdividing the die into smaller clock regions and applying a grid to serve each region. The grid effectively shorts the output of all drivers and helps minimize delay mismatches, resulting in a more gradual delay profile across the region.

2.1.4.5 Hybrid Distribution

In a processor design, the most common design technique is the hybrid clock distribution. It incorporates a combination of other topologies, providing more scalability. A common approach is the tree-grid distribution, taht employs a multi-level H-tree driving a common grid that includes

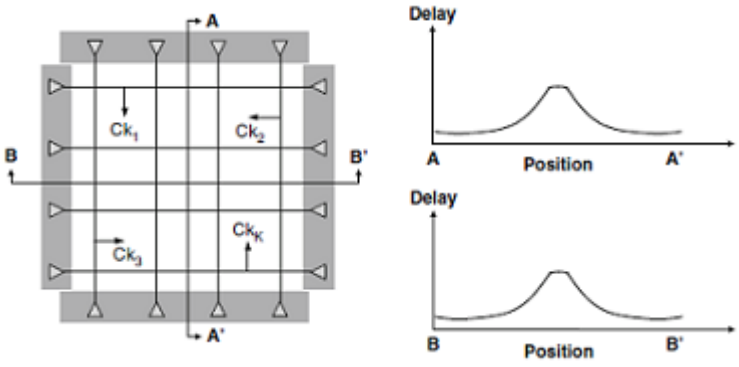

Figure 2.5: Clock grid with 2-dimensional clock drivers.

all local loads. Figure 2.6 shows an example of a processor clock distribution with a first level H-tree connected to multiple secondary trees that are asymmetric but delay balanced.

Several clock distribution topologies have been presented. The primary objective is to deliver the clock to all corners of the die with low skew. Possible improvements of the original tree distribution system consist in providing the clock generator with a skew compensation mechanism [3].

Even if the adaptive design may exhibit higher initial skew, the physical design resource needs for a clock network with adaptive compensation are expected to be lower, since the need of accurate and exhaustive analysis for all process effects

The evolution of the processor clock distribution designs eventually incorporated adaptive clock compensation. The table 2.2 summarizes clock distribution characteristics of various commercial processors. The prevalence of adaptive skew compensation techniques is evident.

Table 2.2: Clock distribution characteristics of commercial processors [1].

Name	**Frequency (MHZ)**	**Skew (ps)**	**Technology (nm)**	**Clock Distribution style**	**Deskew**
Intel Merom	3000	18	65	Tree/Grid	Yes
IBM Power6	5000	8	65	Symmetric H-Tree/Grid	Yes
AMD Quad-Core Opteron	2800	12	65	Tree/Grid	
Intel Xeon processor	3400	11	65	Tree/Grid	Yes
Intel Itanium 2 processor	>2000	10	90	Asymmetric tree	Yes
IBM Power5	>1500	27	130	Symmetric H-Tree/Grid	No
Intel Pentium 4 processor	3600	7	90	Recombinant tile	Yes
Intel Itanium 2 processor	1500	24	130	Asymmetric tree	Yes
IBM Power4	>1000	25	180	Tree/Grid	No
Intel Itanium 2 processor	1000	52	180	Asymmetric tree	No
Intel Pentium 4 processor	>2000	16	180	Spine/Grid	Yes
Intel Itanium processor	800	28	180	H-Tree/Grid	Yes

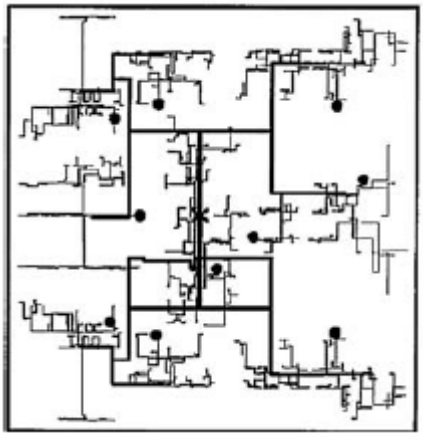

Figure 2.6: Asymmetric clock tree distribution.

2.1.5 Multiclock Domain and Parallel Circuits

As technology scaling comes closer to the fundamental laws of physics, the problems associated with technology and frequency scaling become more and more severe. Technology and frequency scaling alone can no longer keep up with the demand for better CPU performance [4].

In addition, the failure rate in the generation of a global clock began to raise concerns about the dependability of the future VLSI chips [5]. To overcome this problem, VLSI chips came to be regarded not as a monolithic block of synchronous hardware, where all state transitions occur simultaneously, but as a large digital chip partitioned into local clock areas, each area operating synchronously and served by independent clocks within the domain, that can be multiple copies of the system clock, at different phases or frequencies. These areas are also known as isochronous zones or synchronous clocking areas (SCA). Dedicated on-die global interfaces are needed to manage data transfer among the domains.

As digital designs move towards multicores and systems-on-chip (SOC) architectures, this concept of multiple clock domains have become a prevalent design style, and the clock distribution schemes will need to be enhanced to fulfill this need. Each synchronous unit will rely on any of the conventional clock distribution topologies described before to achieve low skew and fully synchronous operation.

This scheme provides functional flexibility for each of the domains to operate at a lower frequency than a single-core processor and to minimize the complexity and power associated with global synchronization.

The multidomain clock distribution architectures for multicore processors and SoCs belong to a class of designs called globally asynchronous and locally synchronous (GALS).

Table 2.3 summarizes synchronization categories within the GALS class and figure 2.7 shows a generic illustration of the GALS design style.

Table 2.3: Clock synchronization categories.

Type	Characteristics of distribution
Synchronous	Single distribution point-of-divergence (POD) with known static delay offsets among all the branches and single operating frequency.
Mesochronous	Single distribution POD but with nonconstant delay offset among the branches.
Plesiochronous	Multiple distribution PODs but with nominally identical frequency among all the domains.
Heterochronous	Multiple distribution PODs with nominally different operating frequencies among the domains.

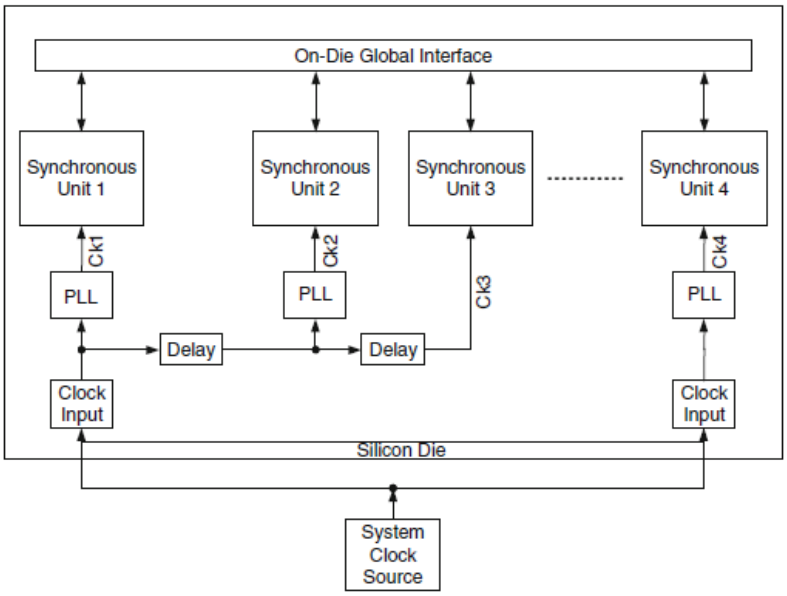

Figure 2.7: Globally synchronous and locally synchronous architecture.

A plesiochronous clock distribution example is the 65nm dual core Xeon processor, wich consists in two domains for the two cores and the uncore and I/O domainwith the interface operating at the same frequency. It uses three independent distribution PODs for the cores and the uncore.

An example of mesochronous clocking cheme, i.e. using the same frequency but with unknown phase [6], is the network-on-chip (NoC) teraFLOPS processor. [7]

The 90nm 2-core Itanium and the 65nm quad-core Itanium processors are examples of a heterochronous clock distribution, supporting nominally different operating frequencies across the domains with multiple clock generators (PLLs).

Many modern multicore processors and SoCs only adopt the loosely synchronous styles of the above to avoid the significant complexity associated with truly asynchronous design, which is an intrinsically analog system, since the time is continuous, and the risk of metastability because the clocks of SCAs that are not really fully independent, is not negligible.

For this reasons, reliability is difficult to guarantee in truly asynchronous systems, and synchronous circuits may be desirable in applications with high reliability requirement [3].

It is clear that multicore processors will be with us for the foreseeable future, trading less single thread performance against better aggregate performance. For many years, increases in clock frequency drove increases in microprocessor performance, but there seems to be no alternative way to provide substantial increases of microprocessor performance in the coming years, considering the KILL rule (Kill If Less than Linear), meaning that any architectural feature for performance improvement should be included if and only if it gives a relative speedup that is at least as big as the relative increase in cost (size, power or whatever is the limiting factor) of the core.

While processors with a few (2–8) cores are common today, this number is projected to grow as we enter the era of manycore computing [8].

This category of chips — with many, but simpler cores — is usually represented by processors targeting a specific domain. The best known representatives of this category are Tilera's TILE-GX family, that consists of a mesh network expected to scale up to 100 cores, picoChip's 200-core DSP as well as Intel's Many Integrated Core (MIC) architecture, that have broken the petaFLOPS barrier (FLoating-point Operations Per Second) [9] [10].

The road ahead for multicore and manycore hardware seems relatively clear, although multicore software is however much less mature, with fundamental questions of programming models, languages, tools and methodologies still outstanding.

2.2 Software Time Management

2.2.1 Real Time Clock and system clock in Linux

A computer has two timepieces that need to be considered: a battery-backed one that is always running, commonly called RTC (Real Time Clock) or Hardware Clock, and the System Clock, also known as software clock, a software counter maintained by the operating system running on the computer.

The RTC consists in a CMOS chip that keeps track of the time even when the system is turned off, storing the values of year, month, day, hour, minute and the seconds [11].

The system clock keeps track of time, time zone and daylight saving Time as the number of seconds since the UNIX epoch, January 1^{st}, 1970, 00:00:00 +00 (UTC).

On the boot of the system, the Linux kernel calculates the system clock from the hardware clock, using the system administration command *hwclock* and from that point the system clock runs independently of the RTC, keeping track of time by counting timer interrupts monotonically, until the reboot or shutdown of the system, when the hardware clock is set from the system clock [12] [13].

But there is no such thing as a perfect clock. Every clock keeps imperfect time with respect to the real time, although quartz-based electronic clocks maintain a consistent inaccuracy, gaining or losing the same amount of time each day. This base "inaccuracy" is known as "time skew" or "time drift".

The two clocks will drift at different rates, and this drift value can be calculated using the difference values on each day when setting the hardware clock, written to the file */etc/adjtime* making possible to apply a correction factor in software with *hwclock(8)*. The system clock is corrected by adjusting the rate at which the system time is advanced with each timer interrupt, using *adjtimex(8)* [14].

But this is for maintain a correct "wall clock" at a computational system, in human timescale units. At a process level timescale, in order to synchronize processes for example, the Linux Kernel has some other services.

2.2.2 Kernel Timers and Time Management

2.2.2.1 Timer Wheel

The term *Real time* defines a time measured from a fixed point in the past, either the Epoch or some point in the life of a process.

Process Time on the other hand is defined as the amount of CPU time used by a process, divided into user CPU time and system CPU time, which is the time spent by the kernel on behalf of the process.

The clock source counter, also called the "timer wheel" which provides the fundamental timeline for the system, measures time in *jiffies*, a kernel-internal value incremented every timer interrupt. The timer interrupt rate, and so the size of a jiffy is defined by the value of a compile-time kernel constant HZ, and the kernel's entire notion of time derives from it [15] [16].

Different kernel versions use different values of HZ. In fact, on some supported architectures, it even differs between machine types, so Hz can never be assumed has any given value. The i386 architecture has had a timer interrupt frequency of 100 Hz, value raised to 1000 Hz during the 2.5 kernel's development series.

Although higher tick rate means finer resolution, increased accuracy in all timed events, and more accurately task preemption decreasing scheduling latency, it implies higher overhead in the processor since implies more frequent timer interrupts, and it must spend more time executing the timer interrupt handler, resulting in not just less processor time for other work, but also more frequent trashing of the processor's cache [17].

Given this issues, since 2.6.13 the kernel changed HZ for i386 to 250, yielding a jiffy interval of 4 ms.

On x86, the main system timer is the programmable interrupt timer (PIT). It is a simple device with limited functionality, but for most user space applications it serves its purpose. Other x86 time sources include the local APIC timer and the processor's time stamp counter (TSC).

The "tick" timer interrupt handler is divided into two parts: an architecture-dependent and an architecture-independent routine.

The architecture-dependent routine is a interrupt handler registered in the allocated interrupt handler list, and its job generically (as it depends on the given architecture) consists in protecting

the write access to the jiffies counter register (*jiffies_64*) and the wall time value, resetting the system's timer, and call the timer routine that does not depend on the architecture.

This routine, called *do_timer()* performs much more work:

- Increment the *jiffies_64* register count by one, safely;
- Update consumed system and user time, for the currently running process;
- Execute *scheduler_tick()*, the kernel function to schedule new tasks;
- Update the wall time, stored in *xtime* struct, defined in *kernel/timer.c*;
- Calculate the CPU load average;

In some situations, timer interrupts can be missed and ticks fail to be incremented, for example if interrupts are off for a long time, so in each timer interrupt algorithm the ticks value is calculated to be the change in ticks since the last update.

do_timer() then returns to the original architecture-dependent interrupt handler, which performs any needed cleanup, releases the *xtime_lock* lock, and finally returns. All this occurs every 1/HZ of a second.

2.2.2.2 KTimers

Kernel timers or dynamic timers are essential for managing the flow of time in kernel code as they are the solution for the need of a tool for delaying a task for a specified amount of time certainly no less and not much longer. These timers are not cyclic, they are created and destroyed after they expire, hence the dynamic nomenclature.

Timers are represented by *timer_list* struct, defined in *linux/timer.h*.

```
struct timer_list {
    struct list_head entry;           /* entry in linked list of timers */
    unsigned long expires;            /* expiration value, in jiffies */
    spinlock_t lock;                  /* lock protecting this timer */
    void (*function)(unsigned long);  /* the timer handler function */
    unsigned long data;               /* lone argument to the handler */
    struct tvec_t_base_s *base;       /* internal timer field */
};
```

After the creation of the timer and the initial setup, one must specify the fields of the structure above, and activate it, adding it to a linked list where all the timer are stored. The kernel then runs the timer handler when the tick count reaches the specified expiration. Typically, timers are run fairly close to their expiration, however they might be delayed until the first timer tick after their expiration.

Consequently, timers cannot be used to implement any sort of hard real-time processing, so timers with accuracy better than 1 jiffy are needed.

2.2.2.3 Scheduler Clock

In addition to the clock sources and clock events there is a special weak function in the kernel called *sched_clock()*. This function shall return the number of nanoseconds, in a 64 bit register, since the system was started and it is used for scheduling the system, determining the time-slice dedicated to a given task. It is also used for time-stamping an event [18].

An architecture may or may not provide an implementation of *sched_clock()* on its own. If a local implementation is not provided, the system jiffy counter will be used as the scheduler clock, making its maximum resolution 1/HZ of the jiffy frequency for the architecture. The scheduler clock is accessed much more often than the clock source, hence it must very fast and it should also be monotonic, as the clock source.

On a symmetric multiprocessor system (SMP), *sched_clock()* should be called independently on each CPU unit due to race conditions and synchronization issues.

Some hardware will cause clock drifts between the CPUs' scheduler clocks, but most kernels are compiled with the "CONFIG_HAVE_UNSTABLE_SCHED_CLOCK=y" configuration. This tells the kernel to mix several sources into the scheduler clock, including the monotonic wall clock and TSC.

2.2.2.4 High-resolution Timers

After a lot of work trying to integrate high-resolution and high-precision features into the existing timer framework, and after testing various such high-resolution timer implementations in practice, Kernel engineers came to the conclusion that the timer wheel code is fundamentally not suitable to be extended for high-res timers, as it is a very tight code around jiffies and has been optimizes for its usage.

Higher-resolution timing subsystems have been investigated over a long time. As we seen before, program the timer chip to interrupt the kernel ar higher frequencies is not feasible due to the tremendous overhead. UTIME, by University of Kansas, proposes a way to add sub-jiffy precision on the base of dynamic ticks, i. e. generating interrupts only when there is some scheduled work that needs to be accomplished .

The HRT project is a fork of the UTIME codebase, and eventually became the current implementation of high-resolution timers in Linux 2.6.16 [19].

Instead of using the "timer wheel" struct, HRT consists in a time-sorted linked list. The *ktime_t*, which is used to store a time value in nanoseconds, is meant to be used as a "black box" type and its definition depends on the architecture. The interface for hrtimers can be found in *linux/hrtimer.h*.

A timer, represented by struct *hrtimer*, must be bounded to one of two cloks:

- CLOCK_MONOTONIC, maintained by the operating system from the system's boot, it resembles the jiffies tick count and it is guaranteed to always move forward in time.

- CLOCK_REALTIME can jump forward and backwards, as represents the system's best guess of the real time-of-day

Also, a function must be set by the user to be called when the timer expires, as well as a *void * data parameter. The *hrtimer* code implements a shortcut for situations where the sole purpose of a timer is to wake up a process on expiration: if the function is *NULL*, the process whose task structure is pointed to by the data will be awakened.

These timers are used for heavily clock-dependent applications, such as animation, audio/video recording and playback, and motor controls, as it may be used in user space for the implementation of POSIX timers, the *nanosleep()* call, and as the base for interval timers or *itimers*, and timed futex operations [20] [21] [22] [23].

2.3 Clock synchronization algorithms

This section focus on how processes can synchronize their operations in order to, for example, manage the mutual access to a shared resource or in situations where the multiple processes need to agree on the ordering of events.

In a centralized system time is unambiguous, as the centralized server will dictate the system time and it does not matter much if this clock is off by a small amount. Since all processes will still be internally consistent.

With multiple CPUs, each with its own clock, its impossible to guarantee that the crystals don't differ after some amount of time even when initially set accurately. In practice, all clocks counters will run at slightly different rates. This clock skew brings several problems that can occur and several solutions as well, some more appropriate than others in certain contexts.

All the algorithms have the same underlying model of the system. Each processor is assumed to have a timer that causes a periodic interrupt H times a second, but real timers do not interrupt exactly H times a second [24].

Figure 2.8 show a slow, a perfect, and a clock with constant offset. The most important definitions in a clock that the protocols must aim to improve, are:

- Accuracy $\alpha \rightarrow |Cp_i(t) - t| \leq \alpha$ for all i and t
- Precision $\delta \rightarrow |Cp_i(t) - Cp_j(t)| \leq \delta$ for all i, j, t
- Offset $\rightarrow$ Difference between $C_p(t)$ and t
- Drift $\rightarrow$ Difference in growing rate between $Cp(t)$ and t

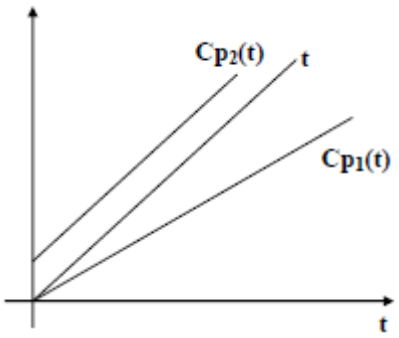

Figure 2.8: The relation between clock time and UT when clocks tick at different rates.

2.3.1 Network Time Protocol

A common approach in many protocols, that lets clients contact a time server, estimating the message delays.

This protocol, like in figure 2.9, A will send a request to B, timestamped with value T_1. B, in turn, will record the time of receipt T_2 (taken from its own local clock), and returns a response timestamped with value T_3, and piggybacking the previously recorded value T_2. Finally, A records the time of the response's arrival, T_4. Let us assume that the propagation delays from A to B is roughly the same as B to A, meaning that $T_2 - T_1 \approx T_4 - T_3$. In that case, A can estimate its offset relative to B as:

$$\theta = T_3 - \frac{(T_2 - T_1) + (T_4 - T_3)}{2} = \frac{(T_2 - T_1) + (T_3 - T_4)}{2} \tag{2.5}$$

There are many important features about *NTP*, of which many relate to identifying and masking errors, but also security attacks. *NTP* is known to achieve worldwide accuracy in the range of 1-50 msec. The newest version (*NTPv4*) was initially documented only by means of its implementation, but a detailed description can be found in [25].

2.3.2 The Berkeley Algorithm

In contrast, in *Berkeley UNIX*, a time server is active, polling every machine from time to time to ask what time it is there, computing the answers to tell the other machines to advance their clocks to the new time or slow their clocks down until some specified reduction has been achieved, since it is not allowed to set the clock backwards.

In figure 2.10 we can see the time server (actually, a time daemon) telling the other machines its time and asks for theirs. They respond with how far ahead or behind they are and the server computes the average and tells each machine how to adjust its clock.

This is more suitable for systems where a radio clock is not present, and there's no way to know the actual real time, and it is sufficient that all machines agree on the same time.

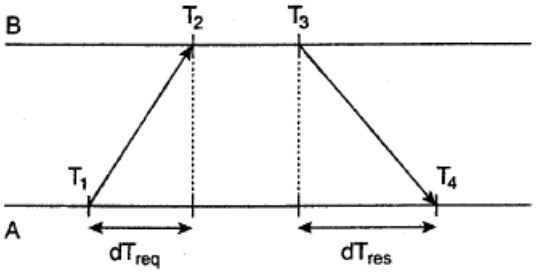

Figure 2.9: Getting the current time from a time server.

2.3.3 Distributed clock synchronization

When there is no master clock, all nodes have to exchange their clock values among themselves. A virtual reference clock can be created averaging all clocks but there is no assumption that there is a single node with an accurate account of the actual time available.

There are two basic classes of algorithm for clock synchronization: those based on averaging and those based on events.

Averaging works by each node measuring the skew between its clock and that of each other node (e.g., by comparing the arrival time of each message with its expected value) then setting its clock to some "average" value.

Event-based algorithms rely on nodes being able to sense events directly on the interconnect: each node broadcasts a "ready" event when it is time to synchronize and sets its clock when it has seen a certain number of events from other nodes [26].

One example of this kind of algorithm is the *Reference Broadcast Synchronization* (*RBS*), where a sender broadcasts a timestamped reference message to adjust the receivers clocks as in NTP, but RBS also allows the packet's time of arrival to be used as a reference point for clock synchronization. To do that, propagation time is measured from the moment that a message leaves the network interface of the sender, eliminating two sources of variation in estimating the communication delays: the time to prepare the massage to be send, and the time to interface with the network. What remains is the delivery time at the receiver, but this time varies considerably less than the network-access time.

Fault-Tolerant Average algorithm (FTA) is another example, in which a node gathers all clocks and eliminates the clocks with the k highest and k lowest skew to use the average of the remaining clock as the virtual reference clock. This method reduces the sensitivity to clock that diverge a lot from others, and clocks with byzantine errors, i. e. arbitrary values.

Another example is the *Interactive Consistency*, where all nodes send a vector with their view of all other clocks and locally build a local matrix with all views of all clocks, remove the byzantine clocks and generate a virtual reference, but with the price of high over-head in the communication.

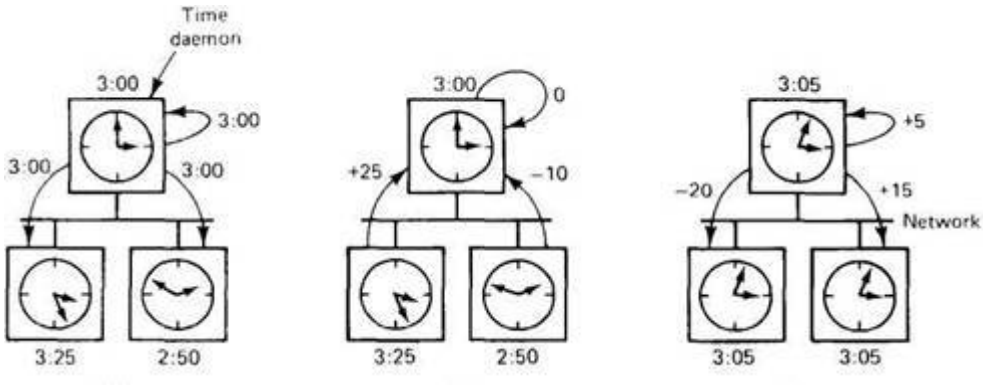

Figure 2.10: The Berkeley Algorithm.

2.3.4 Logical Clocks

In 1978, Leslie Lamport showed that although clock synchronization due to real time is possible, it need not to be absolute in order to two processes interact correctly [27].

Often, what matters is not that all processes agree on what time it is, but rather agree on the order of an event, so Lamport defined a relation called happens-before, that states the conditions that make the statement $a \sim b$ true, that is read "a happens before b".

For that, Lamport timestamps $L(e)$ are assigned to events e.

- If a and b are events in the same process, and $L(a) < L(b)$, then $a \sim b$ is true.
- If a is the event of a message being sent by one process, and b is the event of the message being received by another process, then $a \sim b$ is true.
- If $a \sim b$ and $b \sim c$, then $a \sim c$ is true.

If two processes don't interact, nothing can de said (or need be said) about the order of two events a and b, making these events to be concurrent.

When messages are exchanged, Lamport's solution follows directly the rules, making adjustments to the timestamp of events to make maintain the happens-before relation. If a message arrives and the receiver's clock shows a value prior to the time the message was sent, the receiver fast forwards its clock to be one more than the sending time.

But the main problem of *Lamport clocks* is that $L(a) < L(b)$ does not necessarily imply that a indeed happened before b, since they do not capture causality.

This led to the idea of *vector clocks*. Each process Pi maintain a vector $VCi[n]$ with the number of events occurred in all n processes, including its own, incrementing $VCi[j]$ at the occurrence of each new event that happens at process Pj, giving Pi knowledge of the local time at Pj.

By comparing each other vector timestamps, two processes can determine whether the happens-before relation holds between two pairs of events [28].

Chapter 3

Work Plan

This dissertation divided into 4 components:

- 1. Design of a method to measure the quality of clock synchronization;
- 2. Implementation of the clock synchronization algorithms chosen;
- 3. Instrumentation of the code;
- 4. Experiments, results analysis and interpretation;
- 5. Thesis writing.

These components are represented in the gannt chart of figure 3.1

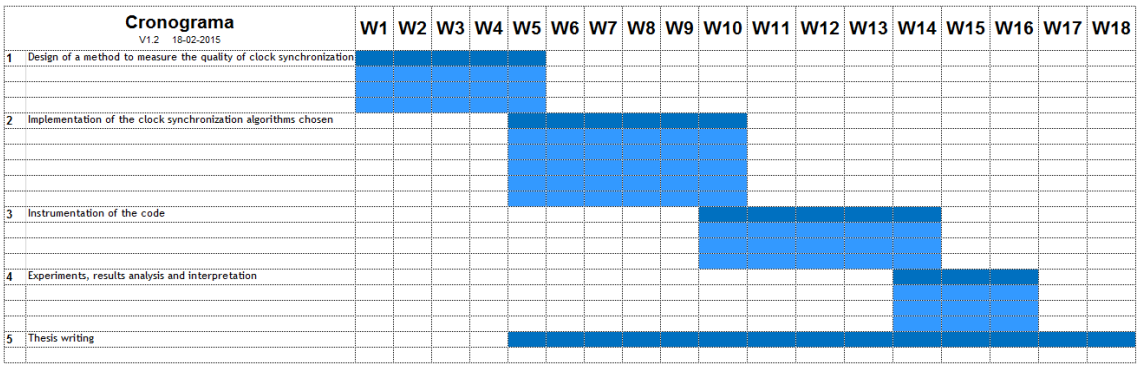

Figure 3.1: Gantt chart of work plan

References

[1] T. Xanthopoulos. Clocking in Modern VLSI Systems. In *Media*, chapter 2. 2009.

[2] E.G. Friedman. Clock distribution networks in synchronous digital integrated circuits. *Proceedings of the IEEE*, 89(5), 2001.

[3] Chuan Shan and Dimitri Galayko. A reconfigurable distributed architecture for clock generation in large many-core SoC. 2014.

[4] XIAOJI YE. Parallel VLSI circuit analysis and optimization. (December), 2010.

[5] Matthias Függer, Ulrich Schmid, Gottfried Fuchs, and Gerald Kempf. Fault-tolerant distributed clock generation in VLSI systems-on-chip. *Proceedings - Sixth European Dependable Computing Conference, EDCC 2006*, 2006.

[6] D Wiklund. Mesochronous clocking and communication in on-chip networks. *Proc. Swedish System-on-Chip Conf*, 2003.

[7] S.R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, a. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, 43(1), 2008.

[8] Karl-filip Faxén. Multicore computing — the state of the art. *Computer*, 1, 2008.

[9] András Vajda. Multi-core and Many-core Processor Architectures. In *Programming Many-Core Chips*. 2011.

[10] Intel Corporation. Intel Many Integrated Core Architecture (Intel MIC Architecture). URL: http://tinyurl.com/dxhjs3g.

[11] STMicroelectronics. Str71X Real Time Clock Application Example. 2004.

[12] Archlinux. Time, 2015. URL: https://wiki.archlinux.org/index.php/Time.

[13] LinuxSA. Linux Tips: Linux, Clocks, and Time, 1998. URL: http://www.linuxsa.org.au/tips/time.html.

[14] Ron Bean. The Clock Mini-HOWTO, 2000. URL: http://tldp.org/HOWTO/Clock.html.

[15] Michael Kerrisk. time(7) - Linux man page, 2012. URL: http://man7.org/linux/man-pages/man7/time.7.html.

[16] Tim Bird. Embedded Linux Wiki - Kernel Timer Systems, 2013. URL: http://elinux.org/Kernel_Timer_Systems.

[17] Robert Love. Timers and Time Management. In *Linux Kernel Development Second Edition*, chapter 10. Sams Publishing, 2nd edition, 2005.

[18] Kernel org. Clock sources, Clock events, scheduler clock and delay timers, 2014. URL: https://www.kernel.org/doc/Documentation/timers/timekeeping.txt.

[19] Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. 30 Seconds Is Not Enough! A Study of Operating System Timer Usage. *ACM SIGOPS Operating Systems Review*, 42, 2008.

[20] Thomas Gleixner and Douglas Niehaus. hrtimers and beyond - transformation of the Linux time(r) system. *History*, 2006.

[21] Kernel org. hrtimers - subsystem for high-resolution kernel timers, 2014. URL: https://www.kernel.org/doc/Documentation/timers/hrtimers.txt.

[22] Tim Bird. Embedded Linux Wiki - High Resolution Timers, 2013. URL: http://elinux.org/High_Resolution_Timers.

[23] Caldera_International. clock(HW) - High-precision system clock, 2003. URL: http://osr507doc.sco.com/en/man/html.HW/clock.HW.html.

[24] Andrew S Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms, 2/E*. 2nd edition, 2006.

[25] David L. Mills. *Network Time Synchronization: the Network Time Protocol on Earth and in Space*. 2nd edition, 2011.

[26] John Rushby. Bus Architectures for Safety-Critical Embedded Systems. *Embedded Software*, 2211(October), 2001.

[27] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

[28] Pedro F. Souto. Lamport Logical Clock and Clock Vectors. 2013.