M 2015

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

CLOCK SYNCHRONIZATION FOR MODERN MULTIPROCESSORS

ANDRÉ DOS SANTOS OLIVEIRA
DISSERTAÇÃO DE MESTRADO APRESENTADA
À FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO EM
ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

MIEEC - MESTRADO INTEGRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES | **2014/2015**

A Dissertação intitulada

"Clock Synchronization for Modern Multiprocessors"

foi aprovada em provas realizadas em 21-07-2015

o júri [illegible]

Presidente Professor Doutor António José de Pina Martins
Professor Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

[illegible]

Professor Doutor Paulo Manuel Baltarejo de Sousa
Professor Adjunto do Departamento de Engenharia Informática do Instituto Superior de Engenharia do Porto

[illegible]

Professor Doutor Pedro Alexandre Guimarães Lobo Ferreira Souto
Professor Auxiliar do Departamento de Engenharia Informática da Faculdade de Engenharia da Universidade do Porto

O autor declara que a presente dissertação (ou relatório de projeto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extratos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são corretamente citados.

[illegible]

Autor – André dos Santos Oliveira

Faculdade de Engenharia da Universidade do Porto

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Clock Synchronization for Modern Multiprocessors

André dos Santos Oliveira

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Pedro Alexandre Guimarães Lobo Ferreira Souto

July 25, 2015

Abstract

Multi-core architectures, in which multiple processors (cores) communicate directly through shared hardware to performs parallel tasks and that way increasing the execution performance, are becoming very present in computer systems, and are increasingly relevant in Real-Time (RT) applications, from data-centers to embedded-systems, not to mention desktops. Clock synchronization is a critical service on many of these systems.

This dissertation focuses on clocks in synchronous digital systems, in particular Intel architectures, from the distribution and analysis of the clock signal in order to coordinate data paths, to software methods and hardware interactions to maintain a time base accurately across a plurality of interconnected processors with possibly different notions of time.

In addition, a clock synchronization algorithm in a shared-memory multi-core architectures is proposed, as if it's a distributed system, applying a model to filter outlet samples, resulting in a random process that achieves a null offset after a couple of corrections. Some experimental measures were made as well to characterize the speed reading the clock and the communication methods used between different cores.

Resumo

As arquiteturas *multi-core*, em que múltiplos processadores (cores) cooperam entre si através de hardware partilhado para realizar tarefas paralelas e assim aumentar a performance de execução, são cada vez mais presentes nos sistemas computacionais, e cada vez mais relevantes em aplicações de tempo real, desde em centros de dados a sistemas embarcados, para não falar dos desktops. A sincronização de relógios é um serviço crucial em muitos destes sistemas.

Esta dissertação concentra-se em relógios de sistemas digitais síncronos, em particular arquiteturas multi-core, desde a distribuição e análise do sinal de relógio para coordenar as interfaces de dados, a métodos de software e interações com o hardware para manter uma base de tempo precisa e síncrona numa pluralidade de processadores interligados com possivelmente diferentes noções de tempo.

Para além disso, é proposto um algoritmo de sincronização de relógios em sistemas multi-core com memória partilhada, como se de um sistema distribuído se tratasse, aplicando um modelo de filtragem de falsas amostras de tempo, resultando num processo aleatorio que coloca o offset nulo após algumas correções. Foram também feitas algumas medidas experimentais para caracterizar a velocidade de leitura do relógio e os mecanismos usados de comunicação entre diferentes cores.

Acknowledgments

Esta dissertação representa o culminar de um percurso académico de trabalho, aprendizagem, e crescimento pessoal junto de pessoas inspiradoras.

Quero agradecer ao meu orientador, Pedro Ferreira Do Souto, pela motivação que me deu nos momentos de maior dificuldade e insegurança, pelo espirito desafiador nos momentos de maior confiança, e pela paciência que teve comigo na focalização do que era essencial.

À equipa técnica do CISTER, por me acolher nas suas instalações e me fornecer material, em especial ao professor Paulo Baltarejo Sousa (DEI-ISEP) por ser uma das únicas pessoas a quem podia recorrer em momentos dificeis, dada a elevada especificidade desta dissertação, e por toda a ajuda que me deu.

A todos os professores que tive oportunidade de conhecer e trabalhar na licenciatura no ISEP, e no mestrado na FEUP, que me formaram e me moldaram para me tornar aquilo que sou hoje, e o Engenheiro que serei no futuro.

Quero agradecer a toda a minha família, em especial aos meus pais, por tudo o que me proporcionaram, pela confiança que sempre tiveram nas minhas capacidades, pela motivação e inspiração que me deram nos momentos mais difíceis deste percurso, e pela paciência que tiveram a lidar comigo e com as minhas atitudes às vezes injustas. Tudo o que sou deve-se ao que me ensinaram e mostraram ao longo da vida.

E por ultimo mas não menos importante, a todos os meus amigos que partilharam comigo este percurso, e me incentivaram a saber mais e aprender mais, e me ensinaram a importância de trabalhar em equipa, em especial aos meus companheiros de mestrado Pedro Medeiros, Luís Duarte, João Lima e André Sá, assim como às pessoas que me acompanharam na fase final, em especial à Fátima Airosa, que me deram a motivação que tanto precisei e a força para manter a confiança necessária à conclusão desta etapa com sucesso. Também devo tudo isto aos meus amigos de longa data que estiveram sempre presentes e pelos momentos inesquecíveis que vivemos e que vão sempre definir parte da minha pessoa.

André Oliveira

”A man with a watch knows what time it is.
A man with two watches is never sure”

Segal’s law

Contents

List of Figures

List of Tables

Abbreviations and Symbols

ACPI	Advanced Configuration and Power Interface
API	Application Programming Interface
APIC	Advanced Programmable Interrupt Controller
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
DAC	Delay Asymmetry Correction
EOI	End-Of-Interrupt
FLOPS	FLoating-point Operations Per Second
FTA	Fault-Tolerant Average
GALS	Globally Asynchronous Locally Synchronous
HPET	High Precision Event Timer
HRT	High Resolution Timer
IC	Integrated circuit
ICR	Interrupt Command Register
IPI	Inter-Processor Interrupt
IRQ	Interrupt Request
IRR	Interrupt Request Register
ISR	In-Service Register
KILL	Kill If Less than Linear
LKM	Loadable Kernel Module
LVT	Local Vector Table
MIC	Many Integrated Core
MPCP	Multiprocessor Priority Ceiling Protocol
MSR	Model Specific Register
NOC	Network On Chip
NTP	Network Time Protocol
PC	Personal Computer
PIT	Programmable Interrupt Timer
PLL	Phase-Locked Loop
PM	Power Management
PTP	Precision Time Protocol
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
TAI	International Atomic Time
TSC	Time Stamp Counter
UTC	Universal Time Coordinated
VLSI	Very Large Scale Integration

Chapter 1

Introduction

1.1 Contextualisation

The use of computer systems is 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. So the time that operations take to run need to be controlled effectively in the design of real-time systems.

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, as the lack of synchronization between the multiple cores can for example degrade the quality of the scheduling algorithms that rely on cooperation.

1.2 The problem, 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 clock source, there is an upper bound on the precision with which clocks can be read on different processors. 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.

This dissertation has the main purpose of estimating the quality of a clock service provided by Linux on common-of-the-shelf multi-core processors such as those of the x64 architectures, and if possible to design and implement a clock synchronization algorithm that would minimize the clock differences, characterizing its implications and limitations.

As an example of the advantage of having access to a synchronized clock in a multi-core system, we describe next its application to a hard real-time semi-partitioned scheduling algorithm for multiprocessors.

In hard real-time systems, application processes, usually referred to as tasks in the literature, can be analysed as if they were periodic. The execution of such processes in each period is often denoted as a job. A critical aspect in a hard-real time system is to ensure that each job of a process is executed before its deadline, which is assumed to be known at design time. Therefore it is essential that the system uses an appropriate scheduling algorithm, allocating cores to each job.

A class of hard real-time scheduling algorithms is known as semi-partitioned. In these algorithms, some jobs may migrate from one core to another to improve core utilization. Therefore, the execution of a migrating job may be described as a sequence of sub-jobs executing in different cores. This means that a sub-job, which is allocated to one core, may not execute before the previous sub-job, which executes on another core, terminates. These precedence constraints can be satisfied by releasing a sub-job only after the previous sub-job has completed.

A straightforward way to do that is to use some inter-processor communication mechanism, such as the inter-processor interrupt (IPI) on the Intel 64 and IA-32 architectures. I.e. when a sub-job terminates, the scheduler running on the same core may generate an IPI to the core that will run the next sub-job. The handling of this IPI will release the next sub-job, which will eventually be scheduled. The problem with this approach is that this communication is on the critical path, with respect to the response time, and, because IPIs are sent via shared bus, the delays incurred may be rather large. Because in hard-real time one must ensure that deadlines are satisfied, usually by carrying out a timing analysis, this implementation may lead to an overly pessimistic estimation of the job response time and therefore to a low CPU utilization.

By relying on high-resolution timers, i.e. timers that are able to measure time with a resolution of 1 microsecond or better, one can reduce the overhead caused by all IPIs but the first. The idea is as follows. Through a timing analysis it is possible to determine the latest time, relative to its release, at which each sub-job will complete. Therefore, one way to ensure the precedence constraints of the different jobs is to schedule the release of each sub-job to the earliest time one can ensure the previous sub-job will be completed. One possible implementation of this approach is as follows: upon release of a migrating job, the scheduler on the core where this happens, will send an IPI to each of the cores that execute the other sub-jobs of that job. The handler of this IPI in each of the cores will then program a high-resolution timer to release the sub-job to a time by which the previous sub-job has terminated. The value with which each high-resolution timer is programmed, must take into account the variability of response time to IPIs. As mentioned earlier, IPIs are sent via a shared bus, which may delay the sending of the IPI. In addition, at each core, received IPIs may be queued behind other interrupts. So, again, this may lead to some pessimism.

One way to reduce further this pessimism is to use synchronized clocks. If each core has access to a global high-resolution clock, the core where the first sub-job is released may timestamp that event, and send an IPI to all cores where the other sub-jobs will be executed. Upon receiving this IPI, the release time of the first sub-job can be read rather than estimated. Therefore in the programming of the high resolution timers used to release the different sub-jobs, rather than using the maximum IPI delay, one can use the actual delay in the delivery of the IPI. In terms of the timing analysis, the IPI delay and its jitter is not in the critical path anymore. Instead, we need to take into account the accuracy of the clock readings at the different cores.

This algorithm has triggered this research work, but we believe that clock synchronization can be used in other scheduling algorithms, as well as in other resource management algorithms.

1.3 Document Structure

In addition to the introduction, this dissertation has 5 more chapters. Chapter 2 presents a review of the state of the art, focusing on the theoretical background necessary to understand the concepts in the scope of this dissertation. Chapter 3 states the description of the clock to be later synchronized as well as its implications, and chapter 4 shows how to synchronize these clocks. Chapter 5 describes the data export mechanism and an analysis on the results of the experiments carried out, and finally chapter 6 presents the final appreciations on the work as well as improvement suggestions to be made in the future.

Chapter 2

State of the art

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

2.1 Clock Source Distribution

In this section, a review on the clock signal distribution technologies is presented, in order to understand the relevant parameters in a characterization of the clocks.

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 [1].

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.

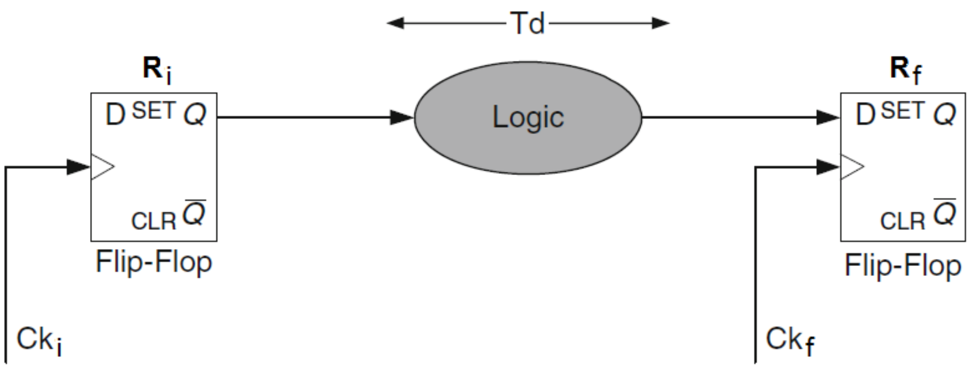


Figure 2.1: Local Data Path

2.1.2 Synchronous Systems

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

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_i and Ck_f 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_i, 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_f, so the minimum allowable clock period $T_{CP}(min)$ between any two registers in a sequential data path is given by equation 2.1.

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

where f_{clkMAX} is the maximum clock frequency, and $T_{PD(max)} = T_{C-Q} + T_d + T_{int} + T_{setup}$.

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 $T_{Skew_{if}}$ can be positive or negative depending on whether Ck_i lags or leads Ck_f, respectively, as shown in figure 2.2.

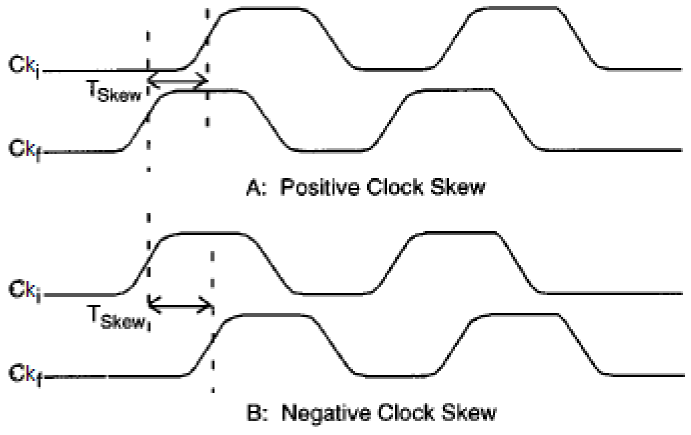


Figure 2.2: Positive and Negative clock Skew

2.1.3 Clock Skew

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

The clock delays of the initial clock signal Ck_i and the final clock signal Ck_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.

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) [2].

Setup specifies whether the digital signal from one stage of the sequential structure has sufficient time to travel 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, where T_{hold} is the hold time requirement for the receive flip-flop.

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

In sum, 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 GALS (Globally Asynchronous and Locally Synchronous).

Table 2.1 lists distribution topologies commonly encountered in synchronous systems.

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

2.1.4.1 Unconstrained Tree

A very common strategy for distributing clock signals used in the history of VLSI (Very Large Scale Integration) 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 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 (Register Transfer Level) 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 the process parameters.

Figure 2.3 shows alternative balanced tree topologies: the tapered X-tree, the H-tree and the binary tree. In a tapered H-tree, the trunk are designed to be wider towards the root of the distribution network to maintain impedance matching at the T-junctions, as it can be seen in the figure.

These three topologies, called *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

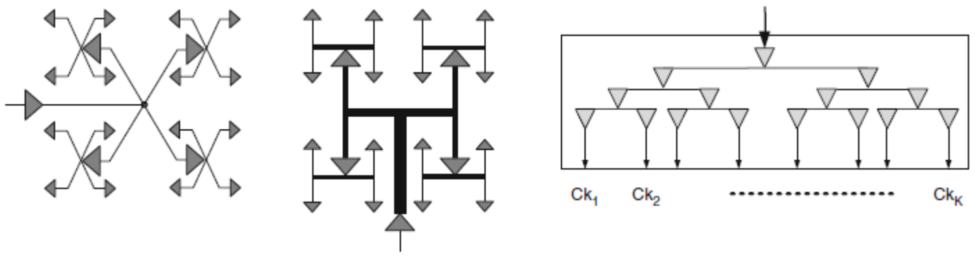


Figure 2.3: Variations on the balanced tree topology
(left) X-tree; (center) H-tree; (right) Binary tree;

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.

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.

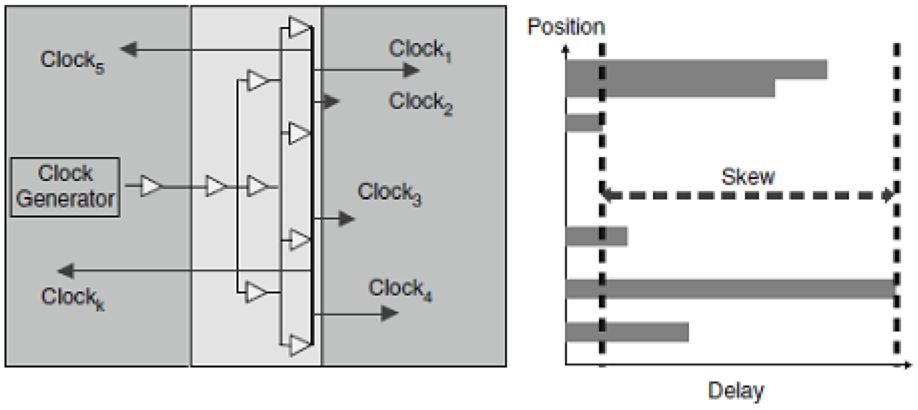


Figure 2.4: Central clock spine distribution

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.

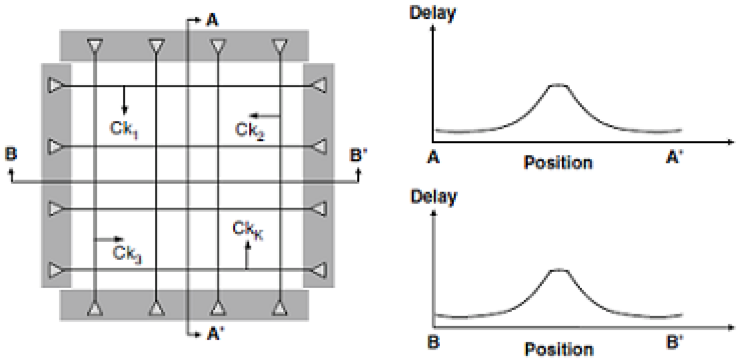


Figure 2.5: Clock grid with 2-dimensional clock drivers

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, that employs a multi-level H-tree driving a common grid that includes 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, because of the need for accurate and exhaustive analysis for all process effects

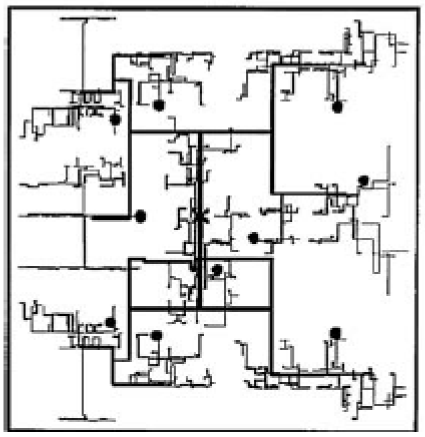


Figure 2.6: Asymmetric clock tree distribution

The evolution of the processor clock distribution designs eventually incorporated adaptive clock compensation. The table 2.2 [2] summarizes clock distribution characteristics of various commercial processors.

Table 2.2: Clock distribution characteristics of commercial processors.

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

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

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.

of the system clock, at different phases or frequencies. These areas are also known as isochronous zones or synchronous clocking areas (SCAs). Dedicated on-die global interfaces are needed to manage data transfer among the domains.

As digital designs move towards multicore and SOC (Systems-on-Chip) 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 GALS class of designs. Table 2.3 summarizes synchronization categories within the GALS class and figure 2.7 shows a generic illustration of the GALS design style.

A plesiochronous clock distribution example is the 65nm dual core Xeon processor, wich consists of two domains for the two cores and the uncore and I/O domain with 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 NoC (Network-on-Chip) 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, i.e. PLLs .

Many modern multicore processors and SoCs only adopt the loosely synchronous styles of the

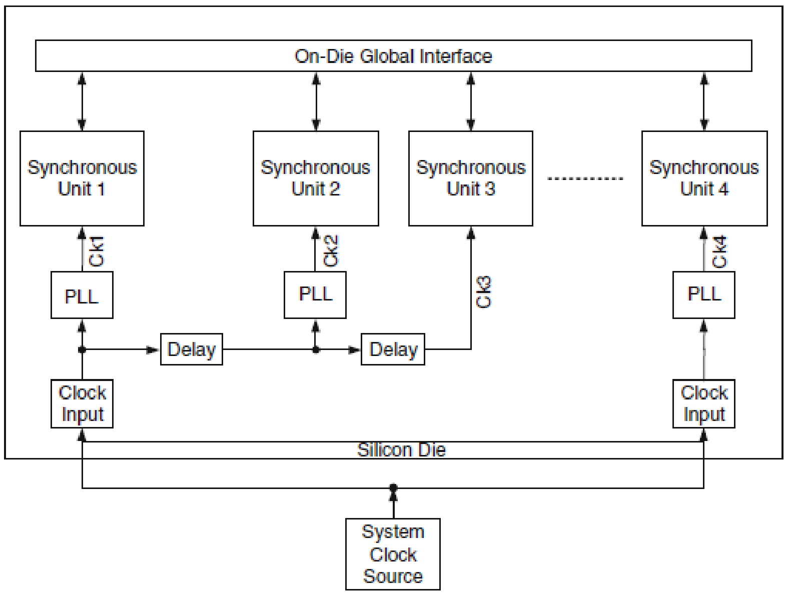


Figure 2.7: Globally synchronous and locally synchronous architecture

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 these 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].

In this section, the fundamental clocking requirements and key design parameters that affect clock performance were understood, as well as different clock distribution topologies, their trade-off, and design techniques needed to overcome their limitations. In conclusion, it became clear that the tendency relies on the parallelism of the resources usage, namely the clock signal source, or even the clock frequency.

2.2 Software Time Management

In this section, we review the time services provided by the Linux kernel.

Many computer activities are driven by timing measurements, that provide keeping the correct time and date for timestamps, and timers to notify the kernel or a software application that a certain interval of time has passed.

2.2.1 Clock and Timer Circuits in the PC

Depending on the architecture, the kernel must interact with some programmable hardware circuits based on oscillators and counters. These circuits provide a free-running counter that issues an interrupt at a fixed frequency, which the kernel handles with an appropriate interrupt handler, implementing the software timers to manage the passing of time [11] [12].

A typical system has several devices that can serve as clocks. Which hardware is available depends on the particular architecture, but the Linux kernel provides a generic interface to all hardware clock chips, with the `clocksource` software abstraction. Essentially, read access to the current value of the running counter provided by a clock chip is granted [13], as can be seen in section 2.2.2.2.

2.2.1.1 Real Time Clock (RTC)

Present in all PCs (Personal Computers), the RTC is a battery-backed CMOS chip that is always running, keeping track of the time even when the system is turned off, storing counters of the year, month, day, hour, minute and the seconds. Some technologies have these counters with respect to the Epoch, i.e. January 1^{st}, 1970, 00:00:00 +00 (UTC) [14], and other maintain certain independent counters time units, like seconds, minutes, hours, months or years [15].

The RTC is capable of issuing periodic interrupts at frequencies between 2 and 8,192 Hz, and can also be programmed to trigger an interrupt when the RTC reaches a specific value, thus working as an alarm clock.

At boot time, the Linux kernel calculates the system clock time from the RTC, using the system administration command *hwclock* and from that point the software 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 [16] [17].

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 time each second.

The RTC and the system clock will drift at different rates, and this drift value can be estimated using the difference of their values when setting the hardware clock upon boot, 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)* [18].

This time source is intended to monitor human timescale units. At a process level timescale, in order to synchronize processes for example, other time sources must be used.

2.2.1.2 Programmable Interval Timer (PIT)

Generically, a programmable interval timer (PIT) is a counter that issues a special interrupt when it reaches a programmed count. The Intel 8253 and 8254 CMOS devices go on issuing timer interrupts forever at a fixed frequency, notifying the kernel that more time intervals have elapsed.

The time interval is called a tick, and its length is controlled by the HZ macro in the kernel code, explained in section 2.2.2.1. The periodic interrupt used to keep the "wall clock" is commonly generated by PIT.

2.2.1.3 Time Stamp Counter (TSC)

Starting with the Pentium, every x86 processors support a counter representing the number of positive edge triggers of the clock signal pin. This counter is available through the 64-bit Time Stamp Counter (TSC) register, that can be read with the assembly instruction `RDTSC`.

Being the clock signal the most basic notion of time of every computational system, the TSC is usually the finest grained, most accurate and convenient device to access on the architectures that provide it [19]. To use it, Linux determines the clock signal frequency at boot time with the `calibrate_tsc()` function, that counts the number of clock signals that occur in a time interval of approximately 5 milliseconds, which is measured with the aid of another clock source, the PIT or the RTC timer.

2.2.1.4 Advanced Programmable Interrupt Controller (APIC) Timer

The local Advanced Programmable Interrupt Controller (APIC) provides yet another time-measuring device, the APIC timer or "CPU Local Timer".

The great benefit of Local APIC is that it's hardwired to each CPU core in Symmetric Multiprocessor (SMP) systems [20]. It provides two primary functions [21] : 1) It receives interrupts from the processor's interrupt pins, from internal sources or from an external I/O APIC and sends these to the processor core for handling; and 2) It sends and receives Inter-Processor Interrupt (IPI) messages to and from other logical processors on the system bus.

Out of all the interrupts the local APIC can generate and handle, the APIC timer is one of them and it consists of a 32 bits long programmable counter that is available to software to time events or operations. Note that the local APIC timer only interrupts its local processor, while the PIT raises a global interrupt, which may be handled by any CPU in the system.

The time base for the local APIC timer is derived from the processor's bus clock, divided by the value specified in a memory-mapped divide configuration register. Since the oscillating frequency varies from machine to machine, the number of interrupts per second it is capable of must be determined with another, CPU bus frequency independent, clock source during APIC initialization, measuring the number of ticks from the APIC timer counter in a specific amount of time measured by that clock.

This timer relies on the speed of the front-side bus clock, i.e. the CPU operating frequency divided by the CPU clock multiplier, or bus/core ratio.

2.2.1.5 High Precision Event Timer (HPET)

The High Precision Event Timer (HPET) is a hardware timer incorporated in PC chipsets, developed by Intel and Microsoft. It consists of one central 32 or 64 bit counter that runs continuously at a frequency of at least 10 MHz, typically 15 or 18 MHz, and multiple timeout registers associated with different comparators, to compare with the central counter. This HPET circuit is considered to be slow to read.

When a timeout value matches, the corresponding timer fires, generating a hardware interrupt. If the timer is set to be periodic, the HPET hardware automatically adds its period to the compare register, thereby computing the next time for this timer to fire.

Comparators can be driven by the operating system, for example to provide a timer per CPU for scheduling, or applications.

2.2.1.6 Advanced Configuration and Power Interface (ACPI) Power Management Timer

The ACPI Power Management Timer is another device that can be used as a clock source, included in almost all ACPI-based motherboards, that is required as part of the ACPI specification.

This device is actually a simple counter increased at a fixed rate of 3.579545 MHz that always roll over (that is, when the counter reaches the maximum, 24-bit binary value, it goes back to zero and continues counting from there) and can be programmed to generate an interrupt when its most significant bit changes value.

Its main advantage is that it continues running at a fixed frequency in some power-saving modes in which other timers are stopped or slowed, but has a relatively low frequency and is very slow to read (1 to 2 μs).

2.2.2 Kernel Timers and Software Time Management

With the several hardware time devices understood, how the Linux kernel takes advantage of them will be discussed.

2.2.2.1 Classical Timers

Classical timers have been available since the initial versions of the kernel. Their implementation is located in *kernel/timer.c*. These timers are also called *timer wheel* and nowadays known as *low-resolution timers* [13].

Essentially, the time base for low-resolution is centered around a periodic tick generated by a suitable periodic source, which happens at regular intervals. Events can be scheduled to be activated at one of these ticks.

With these timers, the software abstraction in the kernel called the "timer wheel", provides the fundamental timeline for the system, measureing 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 [22] [23], assuming that the kernel is defined to work with periodic ticks, situation that change with the creation of *High resolution timers* as discussed later in this section.

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 as 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 because of more frequent timer interrupts whose handler must be executed, resulting in not just less processor time for other work, but also more frequent trashing of the processor's cache [12].

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

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

The architecture-dependent routine is an interrupt handler registered in the allocated interrupt handler list, and its job generically (as the job to be done depends on the given architecture), is to obtain the `xtime_lock` to protect 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 verify if the current task must be interrupted or not;
- 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`, and finally returns. All this occurs every 1/HZ of a second.

The details differ for different architectures, but the principle is nevertheless the same. How a particular architecture proceeds is usually set up in `time_init` which is called at boot time to initialize the fundamental low-resolution timekeeping.

Jiffies provide a simple form of low-resolution time management in the kernel. Timers are represented by `timer_list` struct, defined in *linux/timer.h*.

```
struct timer_list {
    struct list_head entry;             // list head in list of timers.
    unsigned long expires;              // expiration value, in jiffies.
    void (*function)(unsigned long);    // pointer to function upon time-out.
    unsigned long data;                 // argument to the callback function.
    struct tvec_base *base;
};
```

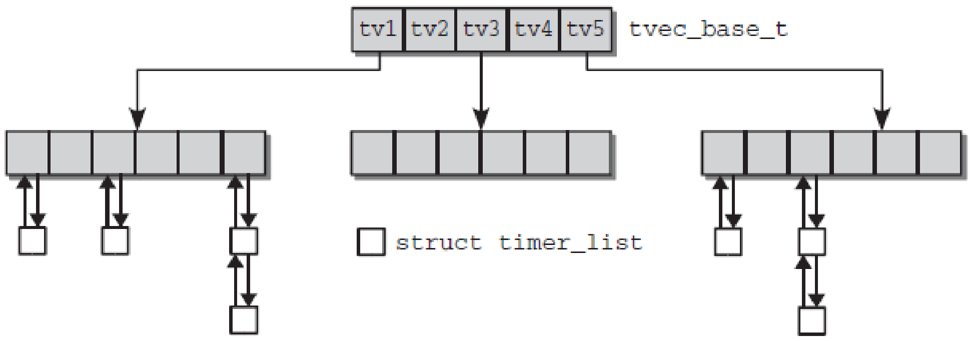


Figure 2.8: Data structures for managing timers

base is a pointer to a central structure in which the timers are sorted on their expiry time. This tvec_base structure exists for each processor of the system; consequently, the CPU upon which the timer runs can be determined using base.

After the creation of the timer and the initial setup, one must specify the fields of the timer_list structure, and activate it with the add_timer function (*timer.h*), adding it to a linked list where all the timers are stored. But because simply stringing together all timer_list instances is not satisfactory in terms of performance, the kernel needs data structures to manage all timers registered in the system to permit rapid and efficient checking for expired timers at periodic intervals and not consume too much CPU time.

The kernel creates 5 different groups within the tvec_base structure, into which the timers are classified according to their expiry time, expires. The basis for grouping is the main array with five entries whose elements are again made up of arrays. The five positions of the main array sort the existing timers roughly according to expiry times, so the kernel can limit itself to checking a single array position in the first group because this includes all timers due to expire shortly. Figure 2.8 shows how timers are managed by the kernel.

The first group is a collection of all timers whose expiry time is between 0 and 255 (or 2^8) ticks. The second group includes all timers with an expiry time between 256 and $2^{8+6}-1=2^{14}-1$ ticks. The range for the third group is from 2^{14} to $2^{8+2*6}-1$, and so on.

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 real-time processing, so timers with accuracy better than 1 jiffy are needed. However *low-resolution timers* are useful for a wide range of situations and deal well with many possible use cases.

2.2.2.2 High-resolution Timers

For many applications, a timer resolution of several milliseconds, typical of low resolution timers, is not good enough. The hardware presented in the previous section provides means of much more precise timing, achieving nominally resolutions in the nanosecond range. During the development

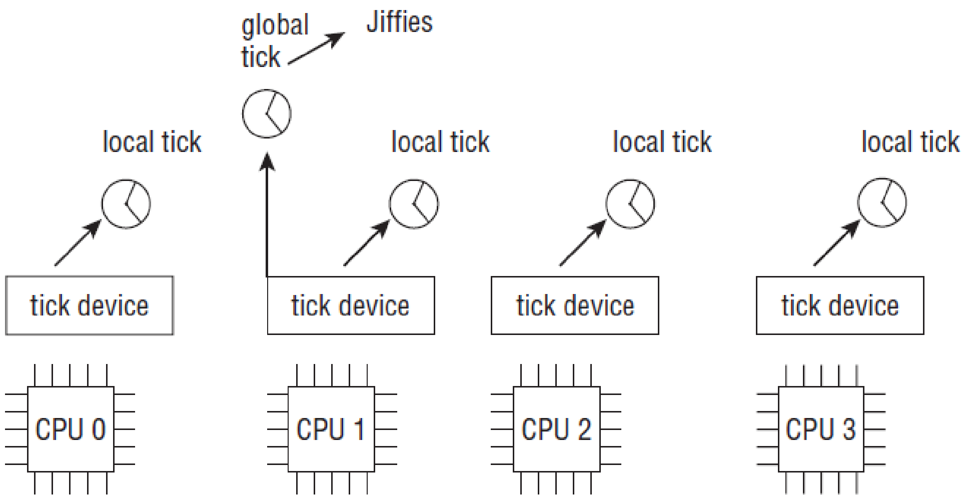


Figure 2.9: Overview of the generic time subsystem

of kernel 2.6, an additional timer subsystem was added allowing the use of such timer sources. The timers provided by the new subsystem are conventionally referred to as *high-resolution timers*.

The mature and robust structure of the old timer subsystem did not make it particularly easy to improve while still being efficient, and without creating new problems. As we've seen before, to program a timer chip to interrupt the kernel at higher frequencies is not feasible due to the tremendous overhead [24]. The core of the high-resolution timer subsystem of the kernel can be found in *kernel/time/hrtimer.c*.

First, some concepts of the generic time subsystem, how does high-precision timekeeping is achieved in the kernel must be understood. The generic time framework provides the foundations for high-resolution timers, and is reused by low-resolution timers. In fact, in recent kernels low-resolution timers are implemented on top of the high-resolution mechanism. The generic timekeeping code that forms the basis for high-resolution timers is located in several files in *kernel/time*. Figure 2.9 [13] provides an overview of the generic time system.

There are three mechanisms that form the foundation of any kernel task related with time, each of them represented by a special data structure [25]:

- **Clock Sources** (defined by `struct clocksource`) - Provides a basic timeline for the system that tells where it is in time. Each clock source offers a monotonically increasing counter that ideally never stops ticking as long as the system is running, with a variable accuracy depending on the capabilities of the underlying hardware.

- **Clock event devices** (defined by `struct clock_event_device`) - Allow for registering an event to happen at a defined point in time in the future. They take a desired time specification value and calculate the values to poke into hardware timer registers.

- **Tick Devices** (defined by `struct tick_device`) - A wrapper around `struct clock _event_device` with an additional feature that provide a continuous stream of tick events that happen at regular time intervals.

The kernel declares the clocksource abstraction, in the *linux/clocksource.h* file, as a mean to interact with one of the hardware counter possibilities present in the machine [26].

```
struct clocksource {
    const char *name;
    struct list_head list;
    int rating;
    cycle_t (*read)(struct clocksource *cs);
    int (*enable)(struct clocksource *cs);
    void (*disable)(struct clocksource *cs);
    u32 mult;
    u32 shift;
    unsigned long flags;
    ...
};
```

name establishes a human-readable name for the source, and list is a standard list element that connect all available clock sources on a standard kernel list.

rating specifies the quality of the clock, between 0 and 499 to allow the kernel to select the best possible one. On modern Intel and AMD architectures, usually the TSC is the most accurate device, with a rating of 300, but the best clock sources can be found on the PowerPC architecture where two clocks with a rating of 400 are available.

read is a pointer to the function to read the current cycle value of the clock. The timing basis of this returned value is not the same for all clocks, so the kernel shall provide means to translate the provided counter into a nanosecond value. Since this operation may be invoked very often, doing this in a strict mathematical sense is not desirable: instead the number is taken as close as possible to a nanosecond value using only the arithmetic operations multiply and shift with the value of mult and shift field members.

The field flags of struct clocksource specifies a number of flags that characterizes the clock with more detail.

Finally, enable and disable, as the name suggests, are function pointers to allow the kernel to make the clock source available, or not. The machine's clock sources made available by the kernel can be seen in the */sys/devices/system/clocksource/clocksource0/available_clocksource* Linux file.

The clock_event_device structure deals with the time at which it is supposed to generate an interrupt. It is defined in the *include/linux/clockchips.h* file:

```
struct clock_event_device {
    void  (*event_handler)(struct clock_event_device *);
    int  (*set_next_event)(unsigned long evt,
                           struct clock_event_device *);
    int  (*set_next_ktime)(ktime_t expires,
                           struct clock_event_device *);
    ktime_t  next_event;
    u64  max_delta_ns;
    u64  min_delta_ns;
    u32  mult;
    u32  shift;
    enum clock_event_mode  mode;
    unsigned int  features;
    void  (*broadcast)(const struct cpumask *mask);
    void  (*set_mode)(enum clock_event_mode mode,
                      struct clock_event_device *);
    const char  *name;
    int  rating;
    int  irq;
    const struct cpumask  *cpumask;
    struct list_head  list;
    ...
};
```

The meaning of each field is described in the kernel code, but some key elements are to be highlighted.

max_delta_ns and min_delta_ns specify a range of values in nanoseconds in which the event shall take place with respect to the current time, characterizing the delay at each the event can be generated.

event_handler points to the function that is called by the hardware interface code to pass clock events on to the generic layer.

cpumask specifies for which CPU's the event device works. A simple bitmask is employed for this purpose. Local devices are usually only responsible for a single CPU.

next_event stores the absolute time of the next event. This type of variable, ktime_t, is the data type used by the generic time framework to represent time values. This time representation consists of a 64-bit quantity independently of the architecture, and the manipulation of this objects, as well as the conversion to other time formats, must be made by auxiliary functions defined by the kernel in *ktime.h* file.

features characterizes the event device, as a bit string. For example, CLOCK_EVT_FEAT_PERIODIC identifies a clock event device that supports periodic events, as well as CLOCK_EVT_FEAT_ONESHOT marks a clock capable of issuing one-shot events, that happen exactly once.

Most clocks allow both possibilities, but it can only be in one of them at a time, defined in `mode`, and the `set_mode` function pointer is used to set the current mode of operation.

The next event in which `event_handler` is called to be triggered is configured in the `set_next_event` function, which in turn sets `next_event` using a clocksource delta, or with `set_next_ktime` that uses a direct `ktime_t` value.

But there is no need to call these functions directly to set the mode and next event, as the kernel offers auxiliary functions for these tasks, defined in *kernel/time/clockevents.c*:

```
void clockevents_set_mode( struct clock_event_device *dev,
                           enum clock_event_mode mode)
int clockevents_program_event( struct clock_event_device *dev,
                               ktime_t expires, bool force)
```

Note that each clock event device only has one event programmed, so to manage multiple events, after the execution of the handler of one event, the kernel calculates how much time is left to the next event, and program the clock event device to generate an interrupt at that time. This is analogous to a situation where a family uses only one alarm clock to wake up all people at different times: the person that wakes up first needs to re-program the alarm clock to the next person's wake up time.

Although clock devices and clock event devices are formally unconnected at the data structure level, some time hardware chips support both interfaces.

Besides these two concepts, the kernel distinguishes between two types of clocks, as we can see in figure 2.9:

- A **global clock**, that is responsible to update the *jiffies* value, the wall time and the system load statistics.

- One **local clock** per CPU, that allows process accounting, profiling and most importantly, high-resolution timers.

Note that high-resolution timers only work on systems that provide per-CPU clock sources. The extensive communication required between processors would otherwise degrade system performance too much as compared to the benefit of having high-resolution timers.

After discussing the generic time framework, the implementation of high-resolution timers can be reviewed. These timers are distinguished from low-resolution timers in two aspects: first, the HR (High-Resolution) timers are time-ordered on a red-black tree, and second they are independent of periodic ticks, employing nanosecond time stamps instead of a time specification based on jiffies.

Since low-resolution timers are implemented on top of the high-resolution mechanism, the generic part of the high-resolution timers framework will always be built into the kernel even if support for them is not explicitly enabled. Nevertheless the supported resolution is not any better. This means that even kernels that only support low resolution contain parts of the high-resolution framework, which can sometimes lead to confusion.

HR timers must be bound to one of two clock bases:

- **CLOCK_MONOTONIC**, maintained by the operating system from the system's boot timer, it resembles the tick count and it is guaranteed to always run monotonously in time. It is the preferred clock for calculating the time difference between events [27].
- **CLOCK_REALTIME** can jump forward and backwards. It represents the system's best guess of the real time-of-day and it can be modified by a user with the right privileges.

Currently there are two more clock bases in the kernel: the CLOCK_BOOTTIME that is idential to CLOCK_MONOTONIC, except it also includes any time spent in suspend [28], and CLOCK_TAI, since 3.10, which was managed by the NTP code before, and was moved into the timekeeping core to provide a TAI (International Atomic Time) based clock [29].

Each of these clock bases is an instance of `struct hrtimer_clock_base`, which is equipped with a red-black tree that sorts all pending HR timers, and specifies its type, resolution, and the function to read its current time. The function to read the CLOCK_MONOTONIC is `ktime_get()` and to read the CLOCK_REALTIME, the `ktime_get_real()` function is used, both returning a `ktime_t` time stamp value. There is a data structure with all these clock bases for each CPU in the system, named `struct hrtimer_cpu_base`. Both these structures are in *hrtimer.h* file.

The HR timer itself is specified by `hrtimer` data structure provided by the kernel, defined in the same file:

```
struct hrtimer {
    struct timerqueue_node  node;
    ktime_t  _softexpires;
    enum hrtimer_restart  (*function)(struct hrtimer *);
    struct hrtimer_clock_base  *base;
    unsigned long  state;
#ifdef CONFIG_TIMER_STATS
    int  start_pid;
    void  *start_site;
    char  start_comm[16];
#endif
};
```

`struct timerqueue_node` specifies the node used to keep the timer on the red-black tree, and also the absolute expiry time in the hrtimers internal representation. `base` points to the timer base associated.

`state` is the currently state of the timer, that can be inactive, waiting for expiration (enqueued), executing the callback function, pending, i.e. has expired and is waiting to be executed, or migrated to another CPU.

`start_comm` and `start_pid` are respectively the name and the pid (processor identification) of the task which started the timer, for timer statistics.

The most important field is obviously the timer expiry callback `function` that can return two possible values:

```
enum hrtimer_restart {
    HRTIMER_NORESTART, /* Timer is not restarted */
    HRTIMER_RESTART,   /* Timer must be restarted */
};
```

If the callback returns `HRTIMER_NORESTART`, the timer will simply eliminated from the system after expires. In order to the timer to be restarted, the callback must set the new expiration time on the hrtimer parameter and the return value must be `HRTIMER_RESTART`. The kernel provides an auxiliary function to forward the expiration time of a timer:

```
extern u64
hrtimer_forward(struct hrtimer *timer, ktime_t now, ktime_t interval);
```

Usually `now` is set to the value returned by `hrtimer_clock_base->get_time()`, and for that reason, `hrtimer_forward_now(struct hrtimer *timer, ktime_t interval)` was created, that already does that.

The new expiration time must lie past `now`, so `interval` is added to the old expiration time the times necessary to make that true. The functions returns the number of times that `interval` had to be added to the expiration time to exceed `now`. This makes possible to track how many periods were missed if periodic execution of the function is desired, and respond to the situation.

To actually set and use the timers, there is a well defined interface provided by the kernel in *hrtimer.h* [30].

In order to use `ktime_t` time values, `ktime_set(long secs, long nanosecs)` is used to declare and initialize them. Several other auxiliary functions exist to handle this kind of variables, like to add or subtract time values and convert to other time representations.

A new `struct hrtimer` is initialized with `hrtimer_init(struct hrtimer *timer, clockid_t which_clock)`, in which `clock` is the clock base to bind the timer to, and `mode` specifies if time values are to be interpreted as absolute or relative to the current time, with the constants `HRTIMER_MODE_ABS` and `HRTIMER_MODE_REL`, respectively.

The next step is to define the `function` pointer to the callback, since this is the only field that is not set with the API funtions, and `hrtimer_start` is used to set the expiration time of a timer, declared before, and starts it. The `hrtimer` code implements a shortcut for situations where the sole purpose of the timer is to wake up a process on expiration: if the function is set to `NULL`, the process whose task structure is pointed to by the `data` will be awakened.

If periodic execution of the callback function is desired, after the application code to be executed in it, `hrtimer_forward` must be used, processing the returned overrun eventually, and return `HRTIMER_RESTART`. Timers can be canceled and restarted with `hrtimer_cancel` and `hrtimer_restart` respectively. `hrtimer_try_to_cancel` may also be used with the particularity that it returns -1 if the timer is currently executing and thus cannot be stopped anymore. `hrtimer_cancel` waits until the handler has executed in this case.

When an interrupt is raised by the clock event device responsible for HR timers, the event handler that is called is `hrtimer_interrupt`. Assuming that the high-resolution timers will run based on a proper clock with high-resolution capabilities that is up and running, and that the transition to high-resolution mode is completely finished (since only low-resolution will be available at boot), this function selects all expired timers in the tree, calls the handler function associated with it, and reprograms the hardware for the next event depending on the return value of the handler function, while changing dynamically the `state` of the timer. This is done for each clock base, iteratively.

Concluding, in this section the high resolution timers were reviewed, as well as their implementation. In the next section, the last one of this overview of the Linux time management, shows some top-level application interfaces to use high resolution timer in user-space.

2.2.2.3 Timer APIs

These HR timers are used for heavily clock-dependent applications. In order to support user-level applications, such as animations, audio/video recording and playback, and motor controls, the Linux kernel provides different APIs, i.e. system call interfaces, for using high resolution timers. The most important are the following [31]:

- **timerfd** - An interface defined in *fs/timerfd.c* that presents POSIX timers as file descriptors and waiting for the timer to expire consists of read from it, always returning an unsigned 64-bit value representing the number of timer events since the last read, which should be one if all is going well. If it is more than one then some events have been missed.

- **POSIX timers** - Implemented in *kernel/posix-timers.c*, these timers generate signals indicating the expiration time, and the start of the next period in case of periodic operation. One must wait for the signal to arrive with `sigwait()` and the missed events can be detected using the function `timer_getoverrun()`, which returns zero if none were missed.

- **setitimer** - A system call whose implementation rest in *kernel/itimer.c* that installs interval timers similar to POSIX clocks except that it is hard coded to deliver a signal at the end of each period. The time out is passed in a struct `itimerval` (*time.h*) which contains an initial time out, `it_value`, and a periodic time out in `it_interval` which is reloaded into `it_value` every time it expires.

When *itimers* are used there are three options to distinguish how elapsed time is counted or in which time base the timer resides, specified in `which` parameter in `setitimer`:

- `ITIMER_REAL` fires a `SIGALRM` signal after a specified real time measured between activation of the timer and time-out.
- `ITIMER_VIRTUAL` measures only time consumed by the owner process in user mode, and draws attention to itself by triggering a `SIGVTALRM` signal upon time-out.
- `ITIMER_PROF` calculates the time spent by the process both in user and kernel mode,and the signal sent at time-out is `SIGPROF`.

2.3 Clock synchronization algorithms

This section focus on how processes can synchronize their own clocks.

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

But with multiple CPU's, each with its own clock, it is impossible to guarantee that the crystal oscillators don't have a drift, and 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 system model. 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 per second [32].

Figure 2.10 show a slow, a perfect, and a clock with constant offset. The most important clock parameters to measure a clock synchronization, are:

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

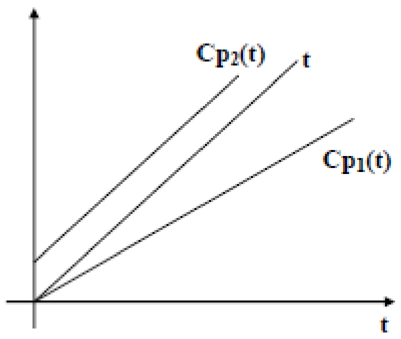


Figure 2.10: The relation between clock time and UT when clocks tick at different rates

2.3.1 Network Time Protocol

A common approach in many protocols, is for a client to read a server's clock and compensate for the error introduce by message delay.

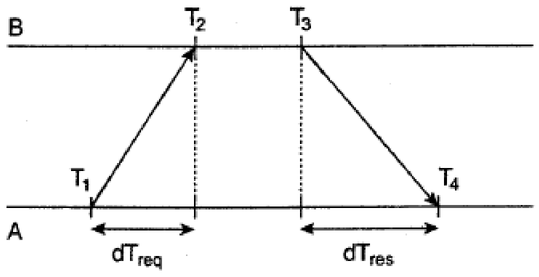


Figure 2.11: Getting the current time from a time server.

In this protocol, (see figure 2.11), 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 stated in expression 2.5.

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

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 [33].

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 2.12 we can see the time server (actually, a time daemon) telling the other machines its own 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 algorithm is more suitable for systems that pursue only that all machines agree on the same time, and not a correct absolute value of the clocks, i.e. internal clock synchronization rather than external clock synchronization.

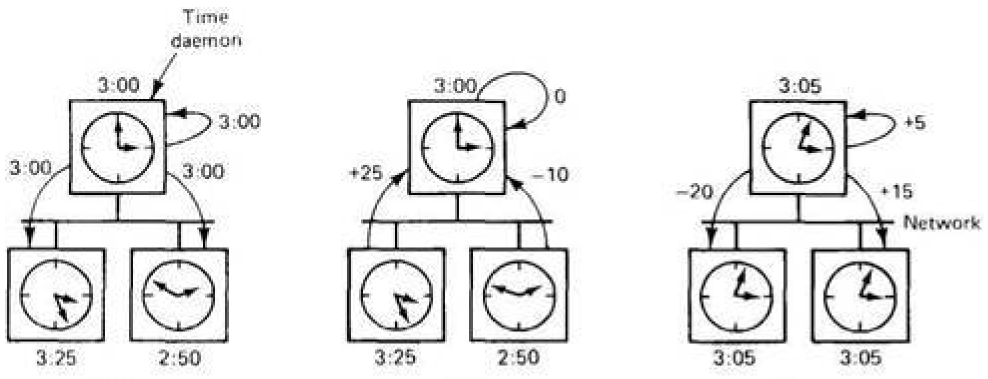


Figure 2.12: The Berkeley Algorithm

2.3.3 Precision Time Protocol

The Precision Time Protocol (PTP) is currently defined in the *IEEE 1588-2008* standard. This standard defines a network protocol to achieve precise and accurate synchronization of real-time clocks in devices of a distributed system that communicate using a network [34] [35].

The target of this algorithm is groups of relatively stable entities, locally networked, cooperating on a set of well defined tasks. It allows multicast communication, unicast communication or both.

The protocol enables heterogeneous systems that include clocks of various inherent precision, resolution, and stability to synchronize to a grandmaster clock, that can be synchronized to a source of time external to the system. It supports synchronization in the sub-microsecond range with minimal network bandwidth and local clock computing resources.

It is intended to measurement and control systems, typical of industrial automation and test environments. It was made to be simple and minimalist in resource requirements on networks and host components.

The IEEE 1588 standard defines the states of a clock, their allowed state transitions, network messages, fields, and semantics. It also describes the datasets maintained by each clock, as well as the actions and timing for all network and internal events.

It is a master-slave protocol and the synchronization and management of the whole process is done by a message exchange between the master and the different slaves.

During the synchronization, two main parameters are estimated:

- Offset: Represents the time difference between two clocks.
- Delay: The time that a message takes to reach its destiny.

Figure 2.13 illustrates all the message exchanges between the master and a slave that are necessary for a correct synchronization.

Periodically, the master node sends a multicast *Sync* message. All the slaves that receive this message save the time (t_2) in which it received it and waits for the master to send a *Follow_up* message containing the time (t_1) that the first message was sent.

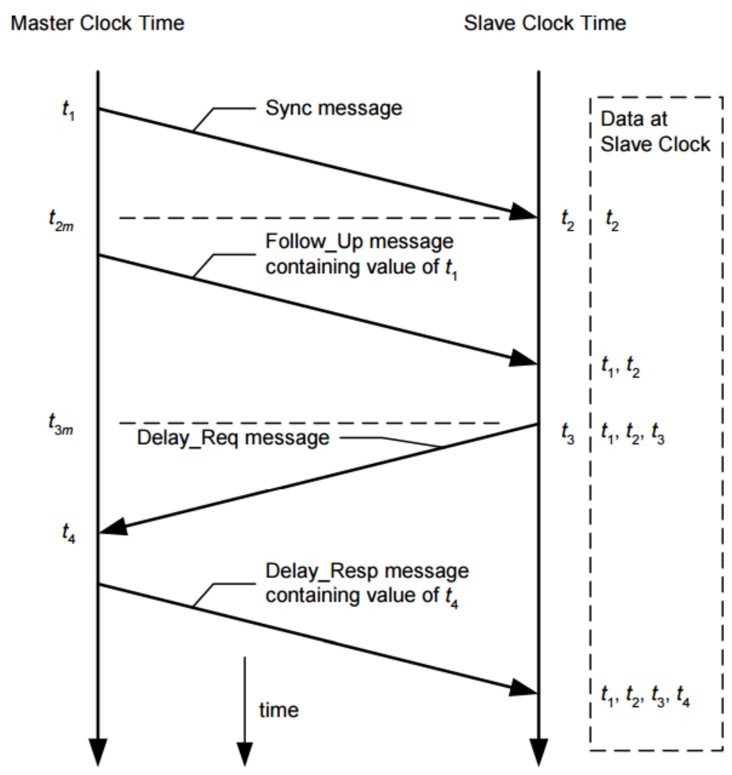


Figure 2.13: PTP message exchange diagram [tutorial]

Afterwards, the slave sends to the master a *Delay_Req* message, recording its sent time (t_3) and then waits for the master's response containing the time in which it received it (t_4).

With all the time references acquired, the slave can estimate the offset value with expression 2.6 and the communication delay with the expression 2.7. The derivation of these expressions can be found in [35].

$$offset = \frac{(t_2 - t_1) - (t_4 - t_3)}{2} \tag{2.6}$$

$$delay = \frac{(t_2 - t_1) + (t_4 - t_3)}{2} \tag{2.7}$$

Knowing the offset of the clocks, the slave can correct its local clock to be synchronized with the master, but the IEEE 1588 standard says nothing about how to do this.

2.3.4 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 [36].

One example of this kind of algorithm is the *RBS* (*Reference Broadcast Synchronization*), 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 on the communication delay estimation: the time to prepare the massage to be sent, 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 clocks 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.

This chapter reviews the literature and the state of the art related to clock synchronization.

The approach is bottom up: we start by reviewing the clock signal distribution issues at the level of the hardware, next we reviewed the various time management services available at the Linux kernel and finally some clock synchronization algorithms used in distributed systems were described.

Chapter 3

Per-core high resolution clock

This chapter describes the clock to be later synchronized, and its implications. First a presentation on the desired local clock is given. Later the clock implementation is described, where a mechanism to evaluate the available hardware in any machine, CPUID, is presented, as well as the process of choosing the hardware clock source to use. Finally the chosen clock source is characterized regarding its I/O access and its native synchronization in its multiple local instances.

3.1 Clock definition

During system's boot, the kernel interacts with the machine's hardware to discover the available clock sources, and selects the best one relying on their rating field on the `clocksource` structure instance of each clock. In order to list the available clock sources made available by the kernel, one must dump the content of */sys/devices/system/clocksource/clocksource0/available_clocksource* file.

With a chosen per-core `clocksource`, this can be used to build a per-core logical clock in software. It was agreed that this clock should be the time value provided by the `clocksource`, counted since the beginning of the algorithm execution. For that, the `clocksource` value must be marked to represent the initial value of each core, and a read to the local clock would return the subtraction of the current time value and the initial value.

The clocks is to be stored globally, permitting a shared memory space for the synchronization, and a generalization of the operations to deal with them. Each clock, CLK_N, at a given point in time, either to be used by the clock synchronization algorithm to get timestamps, or by posterior user applications that want to read it, is given by the expression 3.1, in which N represents the ID of the core, $Time_N$ represents the current time of the CPU N, and $Time_N^0$ is the initial time value, defined in the beginning of the process to be discussed later. The correction applied by the synchronization algorithm to be presented lies in the α_N and β_N values. The α_N is used to control the time rate of the clock, and β_N is an offset corrector.

$$CLK_N = \alpha_N \left(Time_N - Time_N^0 \right) - \beta_N \tag{3.1}$$

3.2 Clock implementation

3.2.1 CPUID

In this dissertation, several functions used are supported by hardware, directly or indirectly, so the hardware features of the machine in use need to be characterized.

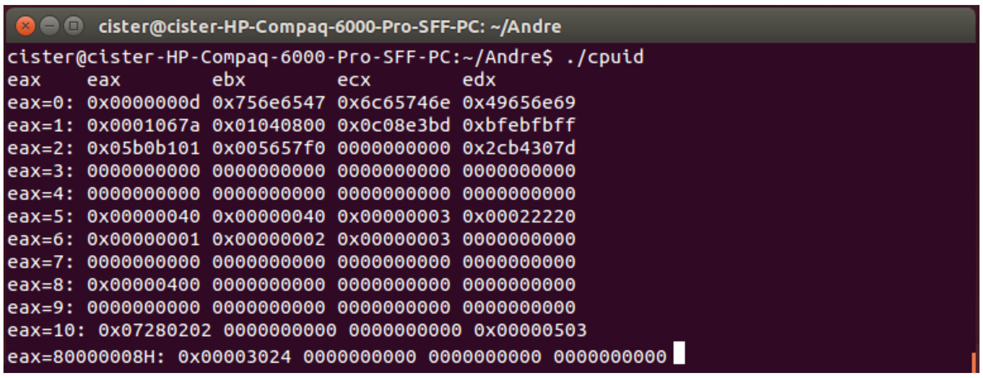
As architectures evolve, the hardware must provide means to enable the software to identify the features available in the CPU. After implementing code sequences and the processor signature identification of processor generations and models, Intel integrated all the information about the features and features supported, creating the CPUID instruction, in a extensible way to allow evolution. With this assembly instruction, software developers can create software applications that can execute compatibly across a wide range of Intel processor generations and models. Although it was created by Intel, other architectures often provide on-chip registers that can be read with this same instruction to obtain the same sort of information.

Standard values in EAX are defined to generate different information to the output registers EAX, EBX, ECX and EDX. For example, with function 01h (EAX=1), the information provided will be with respect to feature information, like processor type or family code.

All the outputs are documented in [37]. In this document, it is used the notation found there. For example "CPUID.01h.ECX[21] = 1" means that if the bit 21 of ECX output, with the EAX = 01h input, is equal to 1 then the particular feature is present in the CPU.

There is a Linux shell command "cpuid [options...]" that processes all the CPUID instruction outputs and dumps detailed information in plain text for each CPU in the system. But to know the value of a particular bit or set of bits, a small C program was created that calls CPUID for several input `eax` values. The output of this program is shown in figure 3.1. This data will serve as a characterization of the machine's features used for the development of this dissertation.

```
void cpuid(unsigned info, unsigned *eax,
           unsigned *ebx, unsigned *ecx, unsigned *edx) {
   __asm__("cpuid;" :"=a" (*eax), "=b" (*ebx), "=c" (*ecx), "=d" (*edx)
           :"a" (info) );
}
```

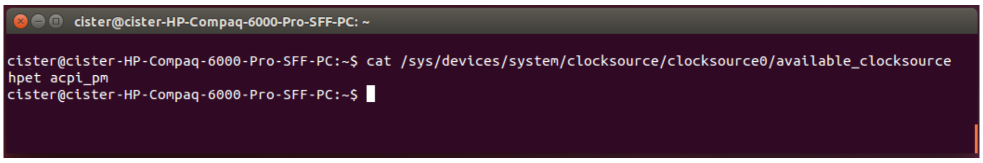


```
cister@cister-HP-Compaq-6000-Pro-SFF-PC: ~/Andre
cister@cister-HP-Compaq-6000-Pro-SFF-PC:~/Andre$ ./cpuid
eax    eax        ebx        ecx        edx
eax=0: 0x0000000d 0x756e6547 0x6c65746e 0x49656e69
eax=1: 0x0001067a 0x01040800 0x0c08e3bd 0xbfebfbff
eax=2: 0x05b0b101 0x005657f0 0000000000 0x2cb4307d
eax=3: 0000000000 0000000000 0000000000 0000000000
eax=4: 0000000000 0000000000 0000000000 0000000000
eax=5: 0x00000040 0x00000040 0x00000003 0x00022220
eax=6: 0x00000001 0x00000002 0x00000003 0000000000
eax=7: 0000000000 0000000000 0000000000 0000000000
eax=8: 0x00000400 0000000000 0000000000 0000000000
eax=9: 0000000000 0000000000 0000000000 0000000000
eax=10: 0x07280202 0000000000 0000000000 0x00000503
eax=80000008H: 0x00003024 0000000000 0000000000 0000000000
```

Figure 3.1: CPUID raw output data

3.2.2 The choice of the local clock source

A said before, in order to list the available clock sources made available by the kernel, one must dump the content of */sys/devices/system/clocksource/clocksource0/available_clocksource* file. Figure 3.2 shows the default results in the used machine.



```
cister@cister-HP-Compaq-6000-Pro-SFF-PC: ~
cister@cister-HP-Compaq-6000-Pro-SFF-PC:~$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource
hpet acpi_pm
cister@cister-HP-Compaq-6000-Pro-SFF-PC:~$
```

Figure 3.2: Initial available clock sources

The current clock source can be inspected by reading the *current_clocksource* file in the same directory. In this case, HPET was automatically chosen as it is the best available, but this is not desirable given the low resolution of this hardware chip for some tasks, as those envisioned.

3.2.2.1 Local APIC

In APIC based systems, i.e. systems with CPUID.01h:EDX[9] = 1, each CPU has a "local APIC". This controller has a timer functionality with a periodic mode of operation, that can eventually serve as the source for the CLOCK_MONOTONIC and CLOCK_REALTIME clock base abstractions. This led to an analysis on the APIC Timer.

As mentioned in section 2.2.1, the local APIC unit contains a 32-bit programmable timer to time events or operations [21]. If CPUID.06h:EAX[2] = 1, this timer runs at a constant rate regardless of P-state transitions (operational states with different frequencies and voltages) and in deep C-states (idle states). If CPUID.06h:EAX[2] = 0 or CPUID 06h function is not supported, it may temporarily stop, making it unreliable as a clock source.

This timer's speed relies on a clock signal derived from the front-side bus clock, whose operating frequency is determined by dividing the CPU frequency by the CPU clock multiplier, or bus/core ratio. On top of that, the APIC implements another divide value specified in the divide configuration register of the APIC's register space. This "register bank" is memory-mapped into a 4 KByte region of the processor's physical adress space, with an initial starting adress of FEE0_0000H.

Besides configuring this divide value, setting up the APIC timer consists of programming other 3 registers: the *initial-count* and *current-count* registers (adresses _0380H and _0390H), and the LVT (Local Vector Table) timer register (adress _0320H) in which the timer mode of operation, the interrupt vector and the delivery status of the interrupt are specified.

There are 2 or 3 modes of operation: Periodic and one-shot modes, that are supported by all local APICs and a third mode called TSC-Deadline mode, present if CPUID.01h:ECX[24] = 1.

When the initial-count register is written, its value is copied into the current-count register and the timer begins to count down to zero, moment at which a timer interrupt is raised. In one-shot mode, the timer remains at 0 until reprogrammed, and in periodic mode the register is reloaded with the initial-count value to restart the count-down. A write of 0 to the initial-count register effectively stops the local APIC timer, in both one-shot and periodic mode.

The TSC-Deadline mode is very different from the other 2 modes. Instead of using the CPU's front-side bus clock frequency to decrement a count, IA32_TSC_DEADLINE MSR (Model Specific Register) must be set to control the absolute time at which a timer interrupt should occur. The local APIC generates a timer IRQ (Interrupt Request) when the value of the CPU's time stamp counter is greater than or equal to this value. Although a higher precision can be achieved with the TSC deadline, there is an uncertainty defining the absolute values of the deadline.

To use the APIC timer, the operating system must know for sure how many interrupts per second it is capable of, as it varies from machine to machine. To do this, after enabling the APIC timer, the counter must be reset and read after a specified amount of time measured with a different clock, to find the number of ticks elapsed. After adjusting it to a second, the kernel multiplies it by the divide value used to find the true CPU bus frequency.

The kernel has the APIC's register space completely defined in *asm/apicdef.h* file, as well as all the interfaces with it in *asm/apic.h*. The characterization of the machines features regarding the APIC, the initialization, setup and calibration routines are all defined in *arch/x86/kernel/apic/apic.c* file.

In the `calibrate_APIC_clock()` function, after setup and configuration of the local APIC timer with the `__setup_APIC_LVTT()` function, the Linux kernel calibrates the APIC by setting a temporary interrupt handler `lapic_cal_handler()` that measures the time elapsed between a specified number of the local APIC interrupts using the TSC (if present), the ACPI PM (Power Management) timer, and the jiffies.

Analysing this function, one can conclude that the local APIC timer is not intended by the kernel to be used as a clocksource for the whole system, but as a clock event device, that generates interrupts at a given input relative time, as the local APIC's `clock_event_device` structure instance is initialized in it. The clock event devices cannot keep track of more information than the time at which it is supposed to generate an interrupt.

3.2.2.2 Time Stamp Counter

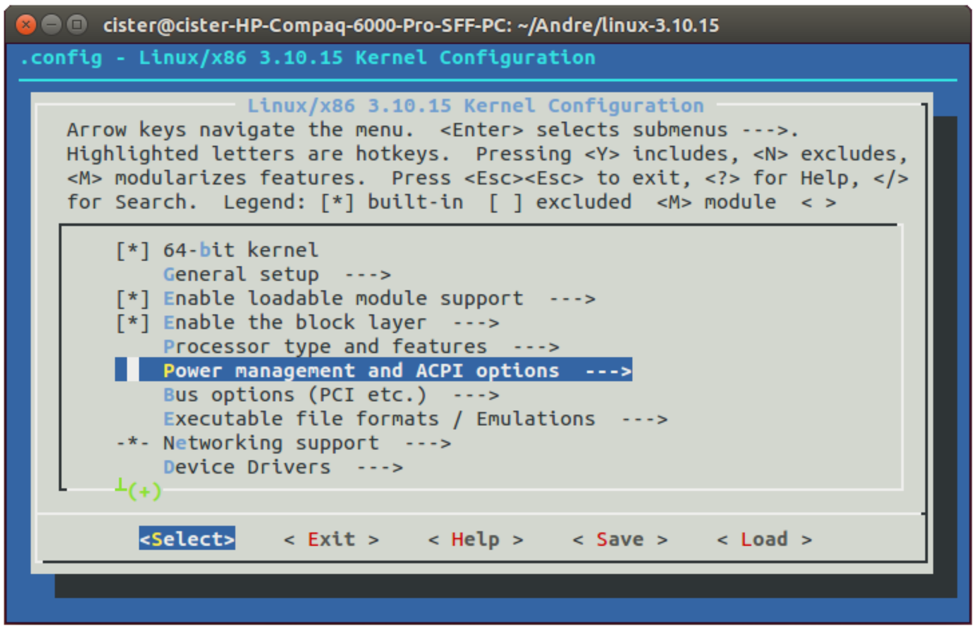
A per-CPU clock source is preferred to the approach of this dissertation, and the clock sources available shown in figure 3.2 are not satisfactory. Since the TSC is supported, i.e. CPUID.01h:EDX[4] = 1, it is expected to be available. This clock source is usually the preferred when it is available, since it has the higher resolution, and is a low-overhead way of getting CPU timing information. However, it can only be used if it is stable. Being directly associated to the processor's clock frequency, it's time rate will vary as the frequency varies dynamically with the power management technology, making it unreliable to manage the passing of time. Also, the processor may lose some cycles from the TSC when entering a halt state.

With the appearance of frequency scaling, systems with multiple CPU's and hibernating operating systems, the TSC ceased to be used as a time reference, until the architectural behaviour moved forward and overcame this problem [38]. With respect to the way the TSC is incremented in current the processor families, 4 TSC types can be distinguished [39]:

- The old type, in which TSC increments with every internal processor clock cycle, so it changes with some deep power management state transitions.
- Constant TSC: The Time Stamp Counter increments at a constant rate. That rate may be set by the maximum core-clock to bus-clock ratio of the processor or may be set by the maximum resolved frequency at which the processor is booted. This behavior ensures that the duration of each clock tick is uniform and supports the use of the TSC as a wall clock timer even if the processor core changes frequency. However it does change on certain power management state transitions.
- Invariant. With this feature, indicated by CPUID.80000007H:EDX[8], the TSC will run at a constant rate in all ACPI deeper states.
- Non-stop. This TSC mode combines the properties of both Constant and Invariant TSC.

The type of TSC present in a processor is identified by the flags entry in the */proc/cpuinfo* file. In the machine used, the `constant_tsc` flag appears in each of the cores. Making a dump of the log messages given by the kernel with the expression "TSC" in it, the results show a message "Marking TSC unstable due to TSC halts in idle", that gives us the reason the TSC is not available. A kernel modification was made, as an attempt to make the TSC stable to be used as the time line of the system.

Having a *.config* file along with the kernel source code, the `make menuconfig` command offers the index of the kernel configuration options to select or deselect, like shown in figure 3.3.



```
cister@cister-HP-Compaq-6000-Pro-SFF-PC: ~/Andre/linux-3.10.15
.config - Linux/x86 3.10.15 Kernel Configuration
                    Linux/x86 3.10.15 Kernel Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >

      [*] 64-bit kernel
          General setup  --->
      [*] Enable loadable module support  --->
      [*] Enable the block layer  --->
          Processor type and features  --->
          Power management and ACPI options  --->
          Bus options (PCI etc.)  --->
          Executable file formats / Emulations  --->
      -*- Networking support  --->
          Device Drivers  --->
      (+)

        <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 3.3: Make menuconfig interface

The kernel can be made lighter disabling some not needed features, and the scheduler jitter can be decreased, increasing the performance.

To make the TSC available as clock source, it is not enough to disable the "frequency scaling" option in Power Management and ACPI Options —> ACPI (Advanced Configuration and Power Interface) Support, will not validate the TSC because it is a "constant TSC". It is not available because of the idle states of the processor.

However, this issue can be worked around by setting the "`idle = poll`" kernel boot parameter in the */etc/default/grub* file. This parameter keeps the clock from entering the idle state, forcing a polling idle loop, and maintaining a continuous time rate on the TSC. This sacrifices low power consumption state when the CPU isn't needed, for reliability in the measurement of time with the TSC.

After updating the grub, dumping the */sys/devices/system/clocksource/clocksource0/available _clocksource* file, the TSC appears in its content, and it is automatically chosen as it has a higher rate in comparison to the others available.

3.2.3 Characterization of the chosen clock source

3.2.3.1 Read access

With TSC being used as the `clocksource` abstraction in the system, the fastest way to read the current time value in expression 3.1, $Time_N$, is by a direct read of the TSC register with the `RDTSC` assembly instruction. The code to generate the needed assembly instructions is presented next.

```
static __inline__ unsigned long long rdtsc(void) {
    unsigned hi, lo;
    __asm__ __volatile__ ("RDTSC" : "=a"(lo), "=d"(hi));
    return ( (unsigned long long)lo)|( ((unsigned long long)hi)<<32 );
}
```

The resulting value, or a difference between two values read, may be converted to a time value, as long as the CPU frequency is known. There may be multiple methods to get the CPU frequency, namely getting access to the `mult` and `shift` parameters of the `clocksource` abstraction. To accelerate this process, this value can be taken from the kernel log regarding the TSC, with the command *dmesg | grep "TSC"*, and hard-coded in a *#define* macro, in KHz. So in another machine, this parameter would have to be redefined, making this approach totally non-portable. Other solution to get the current time is with `ktime_get()`. This function is assigned by the kernel to point to the `read` function field of the CLOCK _MONOTONIC `clock_base` structure, and returns its value in a `ktime_t` format. It consists of a read from a `base_mono` field present in the `struct timekeeper`, a structure holding internal timekeeping values, defined in the *linux/timekeeper_internal.h* file. This `base_mono` value is updated with `timekeeping_update()` function of the *time/timekeeping.c* file, called in the initialization routine of the `clocksource` and timekeeping values, in the suspend/resume methods, and in all functions that change components of the timekeeper or the time of day, and in each update, the value of the `clocksource` is stored in the `cycle_last` field of the `tk_read_base` structure.

The kernel does not have a consistent value of the clocksource in every instant of time, as this is maintained by the hardware, so the ktime_get() function besides reading the updated base_mono value, it gets the nanoseconds passed since the last update, with the function timekeeping _get_ns() that compares the cycle_last value with the current clocksource value, saved in cycle_now.

It should be expected that the resulting value of the RDTSC operation, after a conversion to a nanosecond value given the CPU frequency, could be compared to the result of the ktime_get() function.

A small program was made to compare absolute timestamps of these two operations. A first execution showed an offset of approximately 14.4 seconds, meaning that the TSC and the clock_base abstraction values don't have the same starting point. Later in time this offset appears to have decreased a little, meaning that the TSC and the timekeeper of the system also do not run at the same claimed frequency.

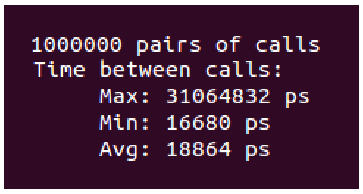
The RDTSC instruction is an instruction to fetch a register maintained by the hardware that counts every clock cycle occurred, and the clock_base is a software abstraction which is initiated later after the RESET signal and after calibration routines at the system's boot to be able to manage time correctly.

From this we can conclude that reading the Time Stamp Counter, or other hardware-maintained register, only makes sense to compare multiple values and get relative time values or intervals.

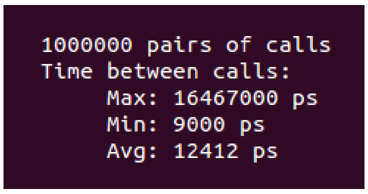
To compare these two instructions, it was made a statistic study about the time it takes to read the TSC directly and a execution of the ktime_get() function, based on [40].

```
(...)
for (i = 0; i < 1000000; i++) {
    tsc1 = rdtsc();
    tsc2 = rdtsc();
    delta = ( (tsc2 - tsc1) * tsc_period_ps);
    if (i == 0 || delta < min) min = delta;
    if (delta > max) max = delta;
    avg += delta;
} avg /= ITERATIONS;
(...)
```

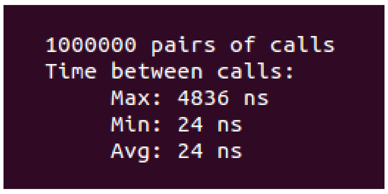
The same approach was used with ktime_get(), except the result comes with a nanosecond resolution. This loop was integrated in a user-space application and in a kernel module. In figure 3.4 we can see the results of these 3 experiments.



```
1000000 pairs of calls
Time between calls:
     Max: 31064832 ps
     Min: 16680 ps
     Avg: 18864 ps
```



```
1000000 pairs of calls
Time between calls:
     Max: 16467000 ps
     Min: 9000 ps
     Avg: 12412 ps
```



```
1000000 pairs of calls
Time between calls:
     Max: 4836 ns
     Min: 24 ns
     Avg: 24 ns
```

Figure 3.4: Results from experiments on read latency
(1) RDTSC from user-space; (2) RDTSC from kernel-space; (3) Ktime_get from kernel-space

As it can be seen in the outputs of the algorithms, the values from user-space are different from the ones in kernel space. This can be due to the scheduling policies applied to application code and to kernel code. These 3 algorithms were not performed binded to a single processor, and a switch of the executer core in the middle of the two consecutive reads can lead to greater values, something that can occur more frequently in user-space.

Also, the TSC is faster to read when compared to the `ktime_get()` function. This is expected due to the nature of each operation. The RDTSC is an assembly instruction itself, and the `ktime_get()` has some code to execute and some structures to access, like described earlier.

With the clock source access reviewed and software methods to read it compared, other issue is to be characterized with respect to this clock source. That is the native synchronization in multiple cores, since this is a per-core source of time manager.

3.2.3.2 Synchronization across multiple cores

In previous multi-core CPUs, each core had its own TSC that would slow down or stop according to the frequency of the CPU itself. In order to make two accurate measurements relative to each other, it was necessary to pin the measuring code to a single core.

On recent intel architectures, all the cores on a PC receive an external RESET signal and a common reference clock signal from one external crystal oscillator, so all see RESET at the same time when the motherboard is powered. The internal clocks in the processors are kept in phase with a PLL [41].

The TSC counter is reset to zero at a RESET signal, and all cores in the same package see RESET synchronously. Intel guarantees a TSC with a constant rate, as long as the `constant_tsc`, `invariant_tsc` and `nonstop_tsc` flags appear on the CPU information.

As long as these assumptions hold, all TSC counters in the same package (same clock source and same RESET signal), are synchronized as if there was only one counter in the system.

Dealing with old CPUs, modifying the TSC itself in an attempt to synchronize was never an hypothesis, because this operation is effectively writing to a moving target, the software is never guaranteed the value it could receive. The various software operations to update the TSC value for a given core can be delayed indefinitely if it has to be re-performed.

In most recent CPU’s, Intel proposed in [42] the use of an *uncore* TSC (i.e. in the region of the package that does not belong to any core) which is incremented on every front-side bus clock by the value in [15:8] bits of a machine specific register called *_PLATFORM_INFO*. If this is set to

25, for example, every bus clock the *uncore* TSC increments by 25, maintaining the illusion that the core is running at the stamped frequency, for purposes of various software.

With this *uncore* TSC, a multi-core package can go into a deep power down (C6) state, and when a core is resumed from it, its internal TSC gets initialized to the value of the *uncore* TSC that didn't go to sleep, plus an offset local to the core. Reading and writing directly to a core offset can be performed, bypassing the need for synchronization between multiple cores of a multi-core processor. As all cores/threads within a socket share a single hardware counter, writing directly to an offset for a core effectively allows software to update a core's offset at its leisure.

The synchronization among the TSC of different cores applies only to 64/ IA-32 Intel architectures, from a certain point forward in their history. Old Intel processors and processors from other architectures may not ensure this synchronization. Besides that, in many-core systems the trend is to adopt a GALS type of solution, where different, asynchronous time domains co-exist.

Given that, it was decided that the clock to be used should be based on the TSC, but without assuming that the different TSCs are synchronized. Instead, the synchronization of the local clocks is assured by the execution of an algorithm as the one described in chapter 4.

In this chapter we have described how to define a clock device per-core on top of a hardware clock source to be chosen on a X86-based machine. In the next chapter we show how to synchronize these clocks.

Chapter 4

Clock Synchronization

In this chapter clock synchronization implementation is presented. After a presentation on the algorithm and the adopted approach, the used communication mechanisms are showed along with the implementation of the algorithm with each of these methods. The section that follows describes a filtering model to deal with the delay asymmetry in the communication, and finally we present the kernel module that was implemented to insert correction algorithm into the system.

4.1 The synchronization algorithm

To synchronize the different per-core clocks, we decided to apply an algorithm based on the PTP algorithm, defined in the *IEEE 1588* standard, and described in section 2.3.3.

As this is a master-slave algorithm, the idea is to define one of the CPUs as the master and the rest as slaves, and the timestamp exchanging routine runs periodically between the master and each slave. For that, it was created a high resolution timer to run in the master, with a relatively large period. Two communication methods for the handler of this timer's expiration were designed, to be presented in the next section.

In order to have the sending time and receipt time in both directions, this algorithm doesn't need the *Follow_up* message from the *IEEE 1588* standard because the sending time of the *Sync* message (t_1 in figure 2.13) may be stored in the shared memory, and the slave is free to read it when it receives the message (after t_2). The same applies with the *Delay_resp* message, whose purpose is sending the receipt time of the previously sent *Delay_req* message (t_4). This message is not necessary with this system model. In fact, even if the last message of the algorithm is sent from the slave to the master, i.e. the slave would have to wait to know the last timestamp value, there is no need for another pseudo-message to the slave or a waiting mechanism in the slave, because the master CPU itself can adjust the slave's clock, modifying the respective memory value as long as it knows the slave's CPU ID.

4.2 The communication mechanisms

In this section, the communication methods are presented. In fact, as the memory is shared among the processors in the architectures we are considering, there is no need to send messages with the purpose of passing information to another node. *Message-based* communication can be implemented as follows. To send a message, a process writes the message to shared memory, and then notifies the destination process. Upon receiving this notification, the destination process then reads the message from the shared memory. So all we need is to implement some notification mechanism.

4.2.1 Inter-Processor Interrupts

To notify a given processor, there is a special type of signal called IPI, which can interrupt another processor or a group of processors that share the bus in which the IPIs are propagated. The IPI mechanism is typically used in SMP systems to send fixed interrupts (interrupts for a specific vector number), special-purpose interrupts, and also used for software self-interrupts, interrupt forwarding, or preemptive scheduling [21].

At the lowest level, a processor can generate IPIs by programming the ICR (Interrupt Command Register) in its local APIC. The software must set up this register to indicate the type of IPI message to be sent and the destination processor and its mode (physical or logical). The act of writing to the low double-word of the ICR causes the IPI to be sent through the APIC bus.

In the P6 family processors, as it is the case of the used machine, when an IPI is received, the receiver CPU's APIC examines the IPI data that arrives through the bus, to determine if it is the specified destination for the IPI. Next it examines if the interrupt request is a special-purpose request, like a NMI or INIT, or one of the MP protocol IPI messages, i.e. a BIPI (Boot IPI), FIPI (Final Boot IPI), or a SIPI (Startup IPI), and in case of being a special-purpose request, the interrupt is sent directly to the processor core for handling. If not, the local APIC looks for an open slot in one of its two pending interrupt queues: the IRR (Interrupt Request Register), that contains the active interrupt requests that have been accepted, but not yet dispatched to the processor and ISR (In-Service Register) registers, that as the names says, has the already dispatched interrupts.

When a fixed interrupt has been dispatched to the processor core for handling, the completion of the handler routine is indicated with an instruction in the instruction handler code that writes to the EOI (End-Of-Interrupt). This deletes the interrupt from its queue and (for level-triggered interrupts) send a message on the bus indicating that the interrupt handling has been completed.

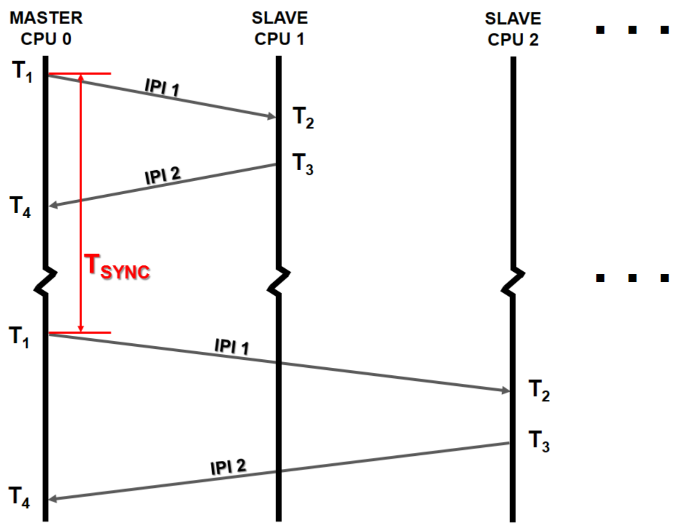


Figure 4.1: Synchronization method with IPIs

In the Linux Kernel the sending and handling of an IPI belongs to the architecture-dependent code, but the *kernel/smp.c* file defines generic helpers for SMP IPI calls, in which we highlight the following:

- **smp_call_function_single**(int cpu, smp_call_func_t func, void *info, int wait)
  - Run a function on a specific CPU.
- **smp_call_function_single_async**(int cpu, struct call_single_data *csd)
  - Like the above, but the call is asynchronous and can thus be done from contexts with disabled interrupts.
- **smp_call_function_any**(struct cpumask *mask, smp_call_func_t func, void *info, int wait)
  - Run a function on *any* of the given CPUs.
- **smp_call_function_many**(struct cpumask *mask,smp_call_func_t func,void *info,int wait)
  - Run a function on a set of other CPUs.
- **smp_call_function**(smp_call_func_t func, void *info, int wait)
  - Run a function on all other CPUs.
- **on_each_cpu**(void (*func) (void *info), void *info, int wait)
  - Call a function on all processors.

The argument `void *info` present in all functions is a pointer to an argument to pass information to the handler of the IPI interrupt. One useful thing to pass is the CPU id number of the IPI sender, for the receiver to know who called for its attention. Note that in `smp_call_function_single_async` this parameter is a field of the call_single_data structure.

Since what we want is to exchange timestamps between the cores to synchronize their clocks, the IPIs serve as a mechanism to mark synchronization instants in the two intervening nodes , and the shared memory is used to pass information about the timestamps.

Figure 4.1 shows the implementation of the timestamp exchange part of the implemented algorithm, based on PTP, with the use of IPIs.

The expiration handler of the periodic high-resolution timer for synchronization, that runs on the master CPU (CPU_0), sends an IPI to a $Slave_N$ (that corresponds to CPU_N), after the read of the IPI sending time. To get an estimate of this time, a read to the current clock, given by expression 3.1 is executed before and after the send of the IPI. This way we know that the IPI is sent in that time interval, and this timestamp, t_1, is assumed to be the average time. Note that in the case of the master, α_N is always one and β_N is always zero.

To get the receipt time in the receiver CPU, the first thing done in the IPI handler is to read its own clock. After getting this timestamp, t_2 according to the standard, the slave sends another IPI to the master (all nodes know that the master is CPU_0) obtaining the clock value before and after the sending, to get t_3. All this is done with disabled preemptions in order to avoid context switching while busy-waiting.

A second IPI handler runs in the master side, in which it gets timestamp t_4. With all four timestamps, the master can correct the slave's clock , with the mathematical expressions in 2.6 and 2.7 of section 2.3.3, taken from PTP's specification.

From the functions presented above to send IPIs, `smp_call_function_single_async` was the chosen one because it can be called from contexts with disabled interrupts. By contrast, `smp_call_function_single`, can deadlock when called with interrupts disabled. Note that the used function has the `void *info` argument as a field in the *csd call_single_data* structure

An alternative to the use of IPIs is to also use a data communication mechanism based in the hardware protocols for memory coherency maintenance.

4.2.2 Multiprocessor Cache Coherency

Cache coherency is intended to prevent data inconsistency arising from storing shared data on private caches, i.e. not shared among cores,

This functionality ensures that any changes to the values of shared variables are propagated throughout the system in a timely fashion. This can be used for *message-based* communication by making the sender processor modify a given memory location, a flag, and the receiver processor to wait for that modification, and mark a timing event when it sees the new value. The only issue is that the receiver must poll the flag in a tight loop to ensure that, with the minimum delay, i.e. it must be prepared to be notified.

A solution to this issue is to use an IPI to begin this process, after which the sender will wait for the receiver to set the flag, and can further on set another flag.

To avoid delays, the flags to implement this mechanism were declared as atomic variables. These type of variables ensure atomic, i.e. indivisible, race-free access even in the presence of concurrent access by processes running on different processors. For reasons of code portability, access to atomic variables in the Linux kernel uses a specific API, listed in [43].

Figure 4.2 illustrates the application of this method in the clock synchronization algorithm. Since it is the master that takes the initiative, an IPI is used to initiate the process. Upon receiving the IPI, the slave "sends" a message, setting the respective atomic flag, and it busy-waits for

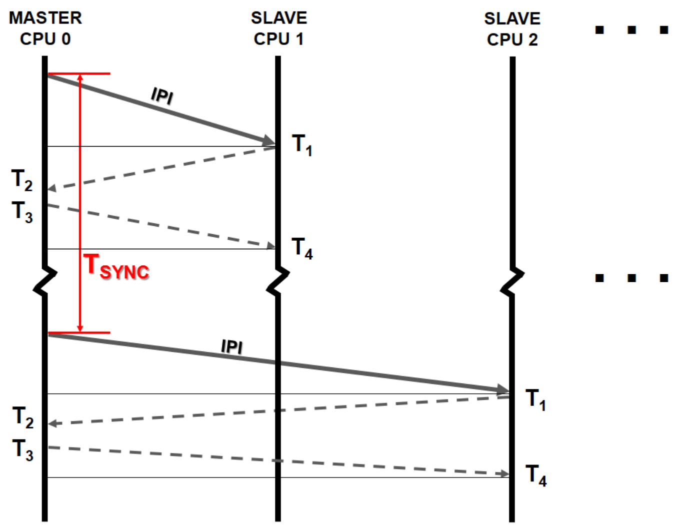


Figure 4.2: Synchronization method with cache coherency

the response. Note that the order of the timestamps is different from the PTP standard and the previously showed method, so offset expression changes to 4.1.

$$offset = \frac{(t_4 - t_3) - (t_2 - t_1)}{2} \tag{4.1}$$

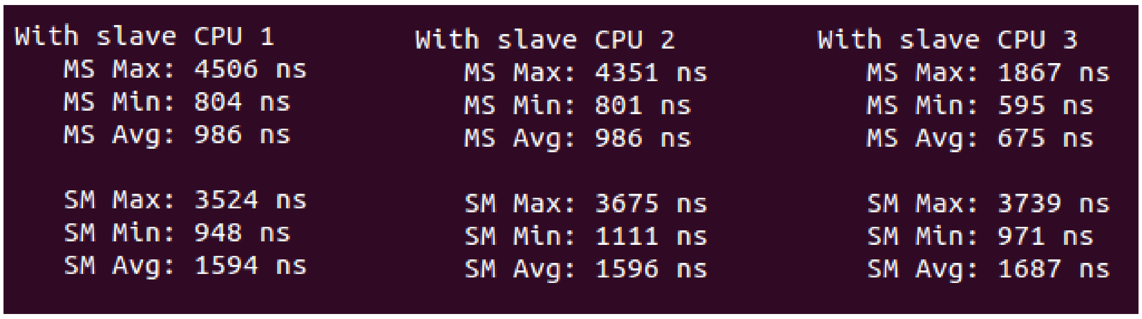
The time stamping follows the same logic as in the previous method, that is, getting the message's exit time is done by reading the clock before and after sending it and calculate its average, and to retrieve the receipt time, a timestamp is taken as soon as the handler method is executed.

After receiving the cache coherency response from the master, the slave node may correct its own clock, based on the expressions 2.6 and 2.7, of the *IEEE 1588* standard. The resulting offset is applied to the β_N of the clock. Note that this parameter is subtracted from the clock, as it can be seen in expression 3.1. If the offset is positive and consequently the slave's clock is ahead of the master's clock, the clock must be set back, and vice-versa.

4.2.3 Comparison

To have an idea on which method is better, it is possible to take advantage of the Intel's guarantee on the TSC's synchronization, and measure the communication latency of each method.

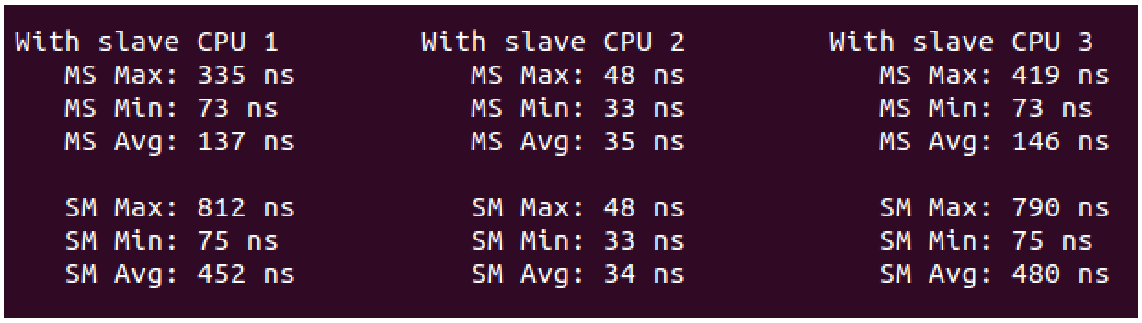
So an experiment was made with timestamps before sending an IPI and after receiving it, in both directions. The experiment consists of making an IPI transaction between the master CPU and a single slave CPU like the implementation on figure 4.1, but with `RDTSC` timestamps, with the given conversion to a nanosecond value. For each slave core, 10 000 iterations were made, presenting the minimum, the maximum and the average latency observed, in each of the directions. The same logic was used for the cache coherency method, like in figure 4.2.



```
With slave CPU 1          With slave CPU 2          With slave CPU 3
   MS Max: 4506 ns           MS Max: 4351 ns           MS Max: 1867 ns
   MS Min: 804 ns            MS Min: 801 ns            MS Min: 595 ns
   MS Avg: 986 ns            MS Avg: 986 ns            MS Avg: 675 ns

   SM Max: 3524 ns           SM Max: 3675 ns           SM Max: 3739 ns
   SM Min: 948 ns            SM Min: 1111 ns           SM Min: 971 ns
   SM Avg: 1594 ns           SM Avg: 1596 ns           SM Avg: 1687 ns
```

Figure 4.3: IPI latency experiment results



```
With slave CPU 1          With slave CPU 2          With slave CPU 3
   MS Max: 335 ns            MS Max: 48 ns             MS Max: 419 ns
   MS Min: 73 ns             MS Min: 33 ns             MS Min: 73 ns
   MS Avg: 137 ns            MS Avg: 35 ns             MS Avg: 146 ns

   SM Max: 812 ns            SM Max: 48 ns             SM Max: 790 ns
   SM Min: 75 ns             SM Min: 33 ns             SM Min: 75 ns
   SM Avg: 452 ns            SM Avg: 34 ns             SM Avg: 480 ns
```

Figure 4.4: Cache Coherence latency experiment results

Figures 4.3 and 4.4 show the results of the test for each method. The "MS" notation denotes for "Master to Slave" and "SM" stands for "Slave to Master".

As expected, the IPI mechanism is more subject to scheduling uncertainties, because the interrupt callback is enqueued as we can see in `generic_exec_single()` function in *smp.c* file. Besides that, the IPI propagation is through the APIC bus, which is shared for both IPIs between all the processors and used by the I/O APIC to forward external interrupt requests to a local APIC for handling. The maximum values show the significant delays that they can take due to contention in the bus or context switching by the operating system.

The cache coherency method is expected to be more subject to context switching by the scheduling algorithm of the operating system, as its execution blocks in a busy loop, waiting for the atomic variable's modification, but with the preemptions disabled this method showed much better results, as we can see in figure 4.4. These results regarding the cache coherency method, came from an experiment with disabled preemptions in the master's and slave's timestamp exchange routines. Without it, several samples showed very high values, some of them reaching up to seconds of delay.

These measurements show that the IPI method has a higher propagation delay. In a way this was to be expected, as the cache coherency is a hardware-only solution, but at the same time it presents a lower deviation. Also, a very important issue to conclude from this experiment, is that the IPI method shows a much less balance in the direction of the communication, becoming a worse method to apply in the clock synchronization algorithm, as discussed next.

4.3 Delay Asymmetry Correction

The PTP protocol assumes that the communication delay is symmetric. The presence of asymmetric message latencies in the forward and reverse communication path between master and slave clocks will limit the achievable accuracy. A way to deal with this problem is by filtering out "bad" clock update samples. In this section, a DAC (Delay Asymmetry Correction) Model for asymmetric communication links based on [44] is proposed to achieve high synchronization accuracy.

After the exchange of the basic timing messages defined in the standard, this algorithm calculates not only the offset but also an estimated delay asymmetry between the master and the slave, called R, that consists of a ratio between the latencies from the master to the slave and from the slave to the master. The detail flowchart of the fist stage of the DAC model is illustrated in figure 4.5 [44].

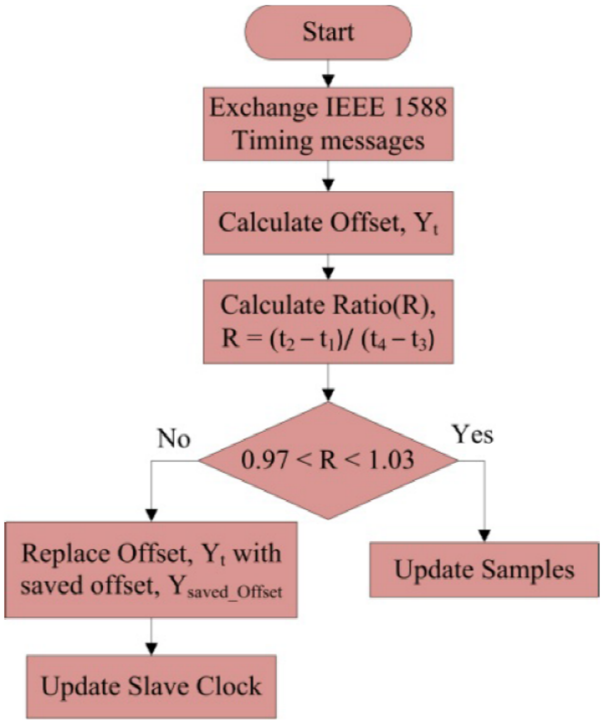

Start
Exchange IEEE 1588 Timing messages
Calculate Offset, Y_t
Calculate Ratio(R), $R = (t_2 - t_1)/(t_4 - t_3)$
$0.97 < R < 1.03$
No
Yes
Replace Offset, Y_t with saved offset, Y_{saved_Offset}
Update Samples
Update Slave Clock

Figure 4.5: Flowchart of DAC model

Only offsets that were calculated with a set of timestamps that pass the first condition of the algorithm ($0.97 < R < 1.03$) are potentially deemed good samples. Otherwise, when the calculated offset values passed the first test, are fed into a 2nd stage filter, named Update Sample Filter. As the name suggests, the second stage filter, not shown in figure 4.5, is implemented in distributed systems to ensure that only good samples are used to update the slave clock to keep the synchronization accuracy high. Since R is an estimate of the unknown asymmetry ratio, there is a possibility that some clock packets may pass the first test with higher or lower offset values than the anticipated offset value. So, those timing packets are considered as outliers and can severely affect the slave accuracy if applied. If the slave realizes that the latencies are highly asymmetric, the recently calculated offset value is discarded and replaced with a previously stored offset value, that has previously passed both filtering stages.

As the PTP is used in distributed systems, the clocks in different machines may have an unknown drift, no matter how small it is, due to different clock sources in the different machines, and this algorithm takes that into consideration. When it does have a successful update, it estimates the drift per synchronization interval, based on the new calculated offset and the number of intervals that passed since the last successful update, and saves this offset. This is the applied offset in each synchronization interval when the values do not pass the filtering stages, independently of the acquired time samples. In sum, the filtering process of the DAC model does not only filter out bad samples, but it also saves a notion of good updates to apply a correction when the time samples are not trustworthy.

The flowchart of this second filtering stage is illustrated in [44] but in our case that doesn't apply. In a multi-core system, even if it is a GALS design (see section 2.1.5), the clock domain of the isochronous zones are fed from the same hardware clock source. This means that the clock rate is the same in every CPU, being the offset the only source of error, therefore the second stage of the DAC model would grow the drift instead of minimize it, and only the first step was implemented.

When R passes the condition, the returned offset is applied to the clock, or else it is discarded. Further offsets obtained from trustworthy time samples are added to the current β_N as this new value results from lack of accuracy of previous calculated offsets. A adaptation was made to the DAC model to never accept worst samples than the ones already applied. To do that, the initial interval of acceptance is higher $\left(\left[90\%, 110\%\right]\right)$ but when a set of samples put the ratio within this interval, the limits are adapted, in order to accept further values that are only better, and achieve a convergence to an ideal null offset. Two or three iterations should be enough, and from a certain point further only values with ratio 100% are accepted to be added to the β_N correction factor of expression 3.1.

4.4 Kernel Module

The clock synchronization was implemented as a kernel module, given the kernel structures that are being dealt with, and working at this level gives the developer more certainties regarding task scheduling and the latencies of timer events.

A LKM (Loadable Kernel Module) is a way to add code to a Linux kernel while it is running, without adding source files to the kernel tree and recompile de kernel. These modules typically are used for three purposes: 1) Device drivers, designed for a specific piece of hardware; 2) Filesystem drivers, that interprets the contents of files and directories, and 3) System calls, that offers services from the kernel to user space programs.

The goal of the developed module is to create a high resolution timer in a single CPU, to execute the synchronization periodically. This CPU acts as the master in the algorithm. Since there is no API to create a `hrtimer` associated with a specific core, i.e. the timer will be associated with the core that executes `hrtimer_init()`, the solution adopted was to create a kernel level thread bound to that core, and create a timer within its execution. In addition, the master CPU should send an IPI to every other CPU in order to mark their initial clock value.

A kernel thread, or *kthread*, is a process that exists only in kernel space and does not have access to user address space. Being an integral part of a kernel and running in a kernel address space, they have access to kernel data structures. Linux implements threads as processes that share resources among themselves, it does not have a separate data structure to represent a thread.

Each thread is represented with `task_struct` and the scheduling of these is the same as that of a process. It means the scheduler does not differentiate between a thread and a process, they are as schedulable and pre-emptable as any other process.

Typically *kthreads* are lightweight processes which perform a certain task asynchronously in background. To see these threads, the *ps -ef* shell command shows a full list of processes running in memory, in which the processes between square brackets are kernel threads.

The kernel thread is created with `kthread_create()`, that returns a `task_struct` structure pointer, and then `wake_up_process()` (*linux/sched.h*) is used to run the thread function passed as argument in `kthread_create()`. But first we want to bind it to a given cpu, and that is accomplished with `kthread_bind()` function. These functions regarding *kthreads* are defined in *linux/kthread.h*.

This process of creating and initiating the *Kthread* is executed when the compiled module is inserted in the kernel with the *insmod* bash command. In the source file, a function is indicated to run at insertion with `module_init()`, as well as at the removal of it (*rmmod command*), with `module_exit()`.

The *kthread* function, `thread_master_fn()`, to be executed in the master CPU, sends a broadcast IPI to every slave, takes its own $Time_N^0$ and registers a new `hrtimer` structure, initializing it with a given periodic expiry time, and starts its countdown to execute a handler. Note that this thread function is not the function to be executed periodically, so after this, the job of the thread is done, but since we need to cancel the local `hrtimer` created in this function, a mechanism was implemented to make it wait until the module is removed.

Upon removal of the module, a function is executed and eliminates all the created *kthreads*, with `kthread_stop()`, so the thread function will wait for it, using `kthread_should_stop()` call, that returns *true* only after `kthread_stop()` is called [45].

```
(...)
set_current_state(TASK_INTERRUPTIBLE);
while(!kthread_should_stop()) {
    schedule();
    set_current_state(TASK_INTERRUPTIBLE);
}
set_current_state(TASK_RUNNING);
(...)
```

This code changes the state of the self thread from "RUNNING" to "TASK_INTERRUPTIBLE", and then releases the CPU usage with the `schedule()` function. The kernel will schedule this thread again later, making the state "RUNNING" again, hence the use of the `set_current_state(TASK_INTERRUPTIBLE)` again. When the thread is stopped, the task is set to "RUNNING" again, and then returns, after cancelling the local `hrtimer` with `hrtimer_cancel()`.

After the register of the `hrtimer`, its local interrupt service routine, is executed every configured period. As pointed out in section 2.2.2.2, if periodic execution of this callback function is desired, after the work to be done in it, `hrtimer_forward()` must be used, processing the returned overrun eventually, and return `HRTIMER_RESTART`. The application code to be executed is the synchronization routine itself, described in the previous sections of this chapter.

In this chapter the developed kernel module implementation was presented, focusing in the communication solutions to exchange timestamps among the CPUs, a mechanism to deal with the delay asymmetry that affects the synchronization quality and the implementation of the algorithm itself, based on PTP.

Chapter 5

Evaluation of the synchronization

In this chapter a tool to export the data to user-space in order to plot the results of the algorithm is presented, as well as an analysis on the results of the experiments carried out.

5.1 Data Export to User-space

In order to evaluate the quality of clock synchronization from a user-space application, we decided to create a file system interface associated with the developed kernel module.

There is a special structure defined in the kernel's file system for this purpose called `file _operations`, that defines a handler for the `open`, `read`, `write`, and `release` requests from a user-space program, the owner module, and other possibilities.

It was decided to associate this struct to a */proc* file, a special kind of file input that belongs to a virtual filesystem, which is sometimes referred to as a process information pseudo-file system. It doesn't contain 'real' files but runtime system information (e.g. system memory, devices mounted, hardware configuration, etc). For this reason it can be regarded as a control and information centre for the kernel [46].

To create a */proc* file, `proc_create()` is used, receiving as relevant arguments the name of the file to be created, the `file _operations` struct instance and the file system's permissions mode. It returns an instance of the `proc_dir_entry` structure that defines the interface to the */proc* file abstraction.

The idea is to make the kernel module to export data values to a queue associated with the */proc* file, and to make a user-space application to read the virtual file when there are new exported values. Since old values that were already exported don't need to be in the queue, an efficient solution is to create a circular queue, with the corresponding read and write item "pointers". The user-space application does not write to the file, it only collects new values, so the `write` request handler doesn't need to be defined.

The `proc_read()` method is executed when the user does a `read` operation on the */proc/ clock_cpuN*. This function dequeues an element from the circular buffer, it validates the size of the returned data so it is not bigger than the requested size in the `read()` function, and it places the element in the buffer whose pointer is passed in the `read()` function as well as in `proc_read()`. To make that available to the `read()` caller process, the function `copy_to_user` is used and finally the length of the data read is returned, like any other `read` system call.

On the kernel module side, it was decided to make the module to export the offset value when a correction is applied. So when a sample ratio is in the permitted range, the offset is applied to the clock and the offset id enqueued in the buffer.

Note that each access to the queue buffer, either a read or a write operation, is assured with a `spinlock`, an access synchronization mechanism that makes the waiting process to busy-wait in a polling loop for the resource that is trying to access.

To manage the data export in the kernel module, there is more than one way. We could have a single */proc* file, and therefore a single queue, the `spinlock` would manage the concurrency among multiple exports, and each buffer element would have to be a structure with the cpu ID attached as well, for the user-space application to be able to know from which clock the values correspond. Being blocked by the queue `spinlock` may compromise the temporal execution of multiple export operations, something that grows as the number of cores grow. This is not a scalable solution.

Instead, it was decided that a */proc/clock_cpuN* file for each core N should be created, as well as a queue, parallelizing the export task to the *proc* file system. The problem with this solution is that the `file_operations` defines the same `proc_read()` for every *proc* file, and the user-space program can't pass an argument in the `read()` system-call. The module should take care of this, i.e. be able to, inside `proc_read()`, know which file was requested to be read, and fetch the corresponding CPU's clock value to return the to user.

The `proc_dir_entry` structure has a field to define a pointer to any position, called `data`. Instead of `proc_create()`, we used `proc_create_data()` that has an additional argument to define that pointer. For that, an array containing all the IDs of the cores must be created to maintain the content of the pointers passed to all `proc_create_data()` calls.

To pass this data to the `proc_read()`, first it has to pass to the `proc_open()`.

```
int proc_open(struct inode *inode, struct file *filp);
```

The `struct file`, defined in *linux/fs.h*, is one of the most important data structures used in device drivers and other virtual files. It represents an open file, and is created by the kernel on `open` and is passed to any function that operates on the file, until the last `close` [47]. This structure has a pointer to `void` called `private_data`, than can hold some information for every operation on the file.

The `inode` structure, on the other hand, is used by the kernel internally to represent files. Therefore, it is different from the file structure that represents an open file descriptor. There can be numerous `file` structures representing multiple open descriptors on a single file, but they all point to a single `inode` structure. This structure contains a great deal of information about the file, and the associated `proc_dir_entry` structure can be accessed from it with the function `PDE()`. This function is not accessible to use in the model, but on top of this function there is even a better one, `PDE_DATA()` that returns the `void* data` of the `proc_dir_entry` associated with the `inode` passed as argument. In sum, the `proc_open()` function only does one thing:

```
int proc_open(struct inode *inode, struct file *filp){
    filp->private_data = PDE_DATA(inode);
    return 0;
}
```

From this point forward, every function that deals with the created */proc* files has the information of which CPU it refers to. In the `proc_read` function, the first thing to be done is to get the respective CPU.

```
    int cpu = *((int *)filp->private_data);
```

Also, the enqueue and dequeue routines had to be modified to receive as argument the CPU, to know the correct circular queue to deal with, to access the respective position indexes for read and write control, as well as to lock and unlock the right `spinlock_t`.

All the conditions are met to develop the user-space application. It is a very simple multi-threaded program with a thread for each core. Each thread opens a single */proc* file and creates a *.txt* file for the respective core and enters an infinite loop reading the */proc/clock_cpuN* file (operation that blocks when there is no data to be read), writing the resulting content to the corresponding *clock_cpuN.txt* file. This solution was created because in a single thread reading the files sequentially, a situation could happen where a given CPU is blocking this application when not generating values whereas other CPUs are exporting values that would not be read. Another solution would be to use the `select()` to monitor the multiple file descriptors.

In the `open` system-call to the */proc/clock_cpuN* file, the program keeps trying to open it until it exists. The idea is to run the user-space application before loading the kernel module, in order to get all the values reported, and have a vision of the offset evolution since the very beginning of the kernel insertion.

In the end, there will be a *.txt* file associated with each core, with one exported element per line, containing a iteration stamp and an offset value, ready to be analysed by an external software to graphically exhibit the evolution of the offset values calculated, in order to measure the quality of the chosen clock synchronization algorithm.

The source code of the kernel module implementation can be seen in appendix A.

5.2 Results

The following section presents the results obtained from the produced kernel module and the implemented clock synchronization algorithm.

To measure the quality of the clock synchronization, the ideal would be to compare the clock values in the same real-time instant, but it is extremely difficult, if possible, for software to execute an operation at the same time in every core.

A single CPU cannot read all the clocks as the `RDTSC` or the `ktime_get()` operations give the clock value of the CPU requesting it.

Also, with a high resolution timer instantiated for every core, it is impossible to assure that the interrupt upon expiration of each `hrtimer` would be handled at the same time, even if they are programmed to expire at the same absolute time value. Besides the fact that the clock event device manages interrupts per core, there is always some delay in the service of the interrupt, with the presence of other tasks with more priority that take the CPU's attention.

Clock values sampled at different points in time cannot be compared to measure the quality of the clock synchronization. So instead, it was decided that it is the offset values that should be exported, i.e. the values that the PTP based algorithm calculates as being the offset of the involved CPU clocks. This is done by the time the correction factor is to be applied, that is when the communication latency ratio lies between the dynamic quality range.

But to analyse the results of the algorithm, getting the offset values applied to the β_N correction factor is not enough to have an idea on the evolution of the algorithm towards the allegedly null offset. It is interesting to know the time elapsed since two adjacent corrections. For that, a counter of iterations was implemented for each core to export in every successful ratio sample achieved, and consequently associate a time value with the offset sample, as long as the synchronization task's period is known.

This iteration value is attached to the offset value to be exported. Since the export is done as a string, a simple `sprintf` function was used to join the two values separated by a comma. This permits an analysis on the time it takes for the algorithm to converge, given the relative small amount of samples that get to be considered as acceptable to be applied.

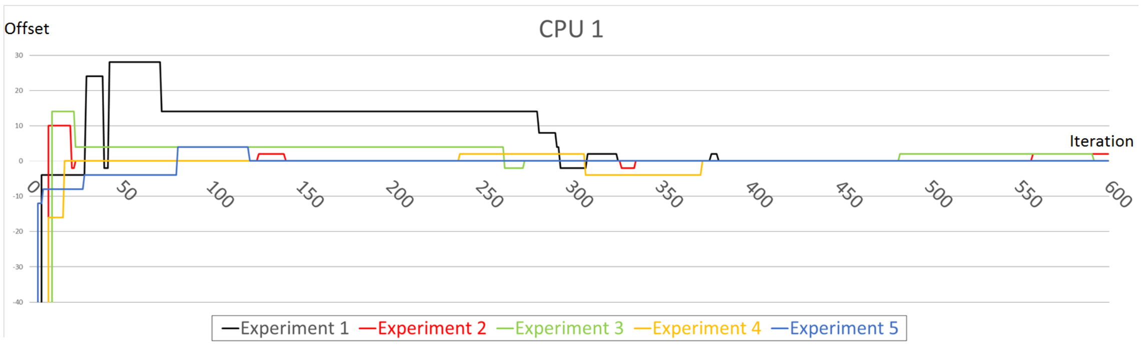
With the *.txt* files, one for each core, containing these two-value entries, one per line, the results of the two implemented methods can be observed in a plot.

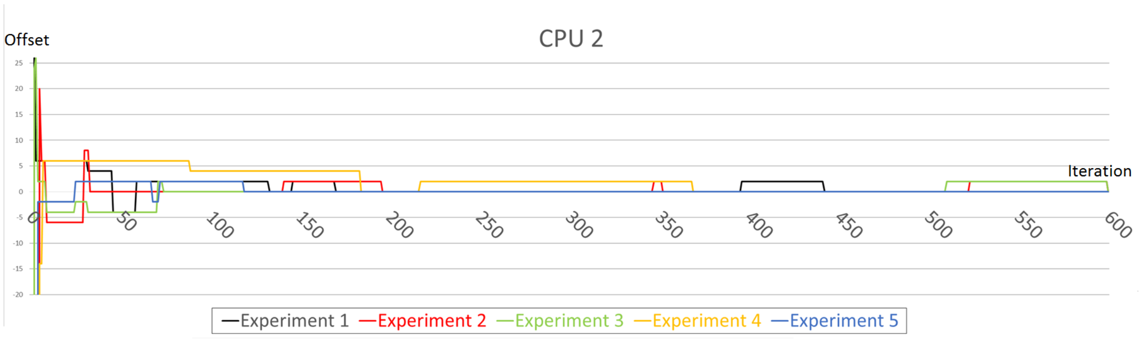
Figure 5.1 shows the results of 5 experiences on the cache coherency method. The plot shows that after a couple of succesfull iterations, i.e. with the latency ratio within the quality range, the offset stabilizes in zero, or close to it.

The first successful timestamp exchange is nearly sufficient to synchronize the clocks. The drift is null, because the same clock signal is fed to all CPUs with a PLL, so the clock difference is constant. Iteratively, the correction factor is accumulated with subsequent samples, that are always better as the ratio quality interval shrinks, and since the offset in further iterations is calculated along with the correction factor of the previous corrections, the resulting offset is the residues from the inaccuracy obtained in previous samples.

In addition, a growth in the time elapsed between applied correction can also be observed. This is due to the increasing difficulty in getting a better balance in the communication latency of each direction. Despite this, the evolution of the process is aleatory, something that can be concluded by comparing different experiments with respect to the same CPU's clock.

Note that the initial offset, achieved in the first successful iteration, is not visible in these plots, because of the magnitude difference. It was noticed that this offset is always negative, i.e. the slave's clock is always behind the master's clock, as expected, because the TSC_0 of the master is always snapshotted before the others.





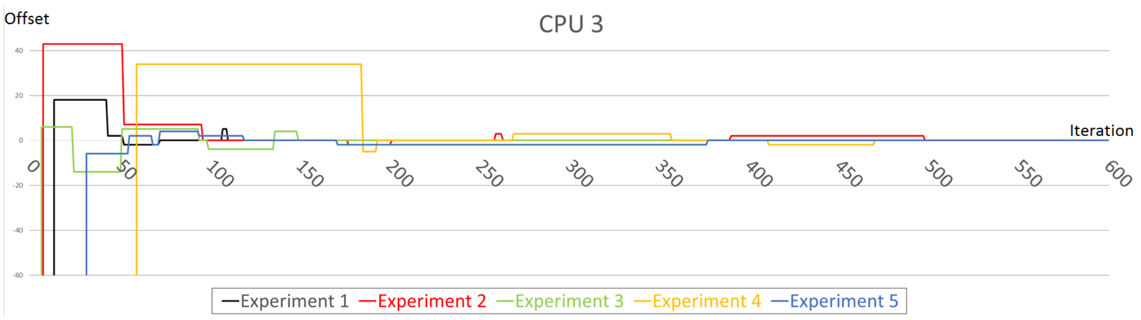


Figure 5.1: Cache Coherence synchronization method results

Besides this algorithm, an isolated test was made to characterize the randomness of the ratio values with the cache coherency method. In a 15 hours of execution, 19,6% of the ratio samples were between 90 and 110% in CPU 1, whereas in CPU 2 10,3% were in the range. In CPU 3, only 3,7% of the ratio calculations were in the defined interval.

Regarding the method with only the IPIs, a strange phenomenon occurred. It turns out the latency was approximately the same in every iteration with respect to one direction, but considerably different comparing one direction to the other. This resulted in a very similar latency ratio amongst iterations, but far from the permitted range to accept the calculated offset, making it impossible to apply corrections for the β_N of the corresponding clock. If samples with unbalanced delay values are considered, the calculated offset values would be misleading and would produce inaccurate correction factors, degrading the clock synchronization.

Chapter 6

Conclusions and Future Work

In this last chapter, we present our conclusions and suggest improvements

6.1 Work carried out and Assessments

The main goal of this dissertation was to design and implement a clock synchronization algorithm that would allow to have a consistent notion of time with high resolution across all cores in a multiprocessor system, characterizing its implications and limitations.

A state of the art review survey is presented on the creation of a time base in a computational system, on time management mechanisms used by the Linux Kernel in Intel architectures and finally on synchronization methods in parallel processing systems.

Various hardware mechanisms acting as clock sources were studied and compared, as well as how the software can interact with them, and how it is currently done in the Linux Kernel in particular. After a careful study the TSC was chosen as the basic clock source for the kernel. Being a per-CPU solution, its characteristics and implications were studied, and its presence in multi-core/multi-processor environments was analysed.

Also, it was expected an implementation of a clock synchronization algorithm. This was made in kernel-space, which required research on working at this level, the kernel modification and re-compilation, the navigation through the kernel source files, and how to develop a kernel module with all the limitations associated with the kernel-space development. The algorithm chosen was based on PTP, used in distributed systems, but with the necessary adaptations, and two methods were presented for the communication between cores: Cache coherency and IPI.

Besides the algorithm, several tests were implemented to characterize the latency of the two communication methods, the delay in obtaining the TSC and the timekeeper of the Kernel with the `ktime_get()`, and a statistical study on the relative amount of ratios that actually hits the quality range defined by the delay asymmetry correction model.

With respect to the algorithm, with the cache coherency communication method, we achieved good results. However, the measurement of the clock synchronization algorithm was based on the offset values reported by the timestamp exchange protocol, not the actual clock differences, which are extremely difficult, if possible, to accurately measure due to the delays in the measurement. The uncertainty of the system's scheduling was a major limitation, because basically a real-time application was implemented on top of a non real-time operating system.

The IPI communication method was shown unreliable. This was a surprising result, as initially it was expected to be used. It was not possible to acquire reliable timestamps to apply trustworthy corrections, resulting in a failed method with no results.

Finally, a method to export values from kernel-space to user-space was implemented, in order to be able to analyse the results obtained . Personally this was a very interesting and fruitful task, as it permitted to understand what's behind the read/write operations on the file-system abstraction.

After a lot of research, implementation, testing, errors and corrections, it is believed that a good review on the possibility of implementing a clock synchronization algorithm in a multi-core system was carried out. It is important to point out that, besides the theoretical interest of this approach, being able to perform an accurate clock synchronization with a nanosecond resolution in a multi-core/multiprocessor system can be extremely challenging, and an important lesson learned with this dissertation is that the limit of the software is the hardware: the software cannot do any better than the best hardware can do.

6.2 Future Work

The major limitation of this dissertation is its portability, as we relied on architecture-dependent kernel functionalities. Intel architectures, in particular multi-core X86-64, were used. Other architectures like used in embedded systems, and current global-purpose architectures like AMD, remain to be scrutinized, as well as new and future architectures - e.g. many-core architectures. The latter is an interesting topic to be studied, as this is the tendency in parallel computing, and it brings the topic of GALS clock distribution, that changes the assumptions made in this work.

Other future work is the study of the time management software mechanisms currently utilized in other operating systems, namely real-time operating systems, that have other scheduling guarantees, decreasing the limitations felt during this dissertation work.

Regarding the synchronization algorithm, it remains to be done a rate correction, with the α_N correction factor of expression 3.1. The drift was not a concern, but on the other hand this algorithm could bring discontinuities when corrections are applied, and within the use of the clock in possible applications this could bring fatal errors in time management. The time rate correction would be the answer to get a monotonically increasing clock value. Also, other clock synchronization algorithms, such as [48], and their shared memory implementation can be evaluated.

Appendix A

Kernel module source code

A.1 Multiprocessor Cache Coherency Method

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/kthread.h>
#include <linux/sched.h>
#include <linux/hrtimer.h>
#include <linux/smp.h>
#include <linux/proc_fs.h>
#include <linux/delay.h>
#include <asm/uaccess.h>


#define CORE_COUNT 4
#define MASTER_CPU 0
#define TSC_KHz 2659983


#define CNTR 0
#define TSC_0 1
#define BETA 2
#define ALPHA 3

#define PERIOD_SYNC_sec 0
#define PERIOD_SYNC_nsec 100000000

#define BUFFER_LEN 100
#define QUEUE_LEN 50


static void increment(int * item);
static int is_full(int r, int w);
static int enqueue (char *buffer, int cpu);


struct task_struct *thread_master;

long long clock[4][CORE_COUNT];



struct call_single_data csd;

unsigned int slave_id;

unsigned long long stamp1_1, stamp1_2,
                   stamp2_1, stamp2_2,
                   stamp4;

signed long long t2_1, t4_3, offset;
long ratio;
long ratio_min[CORE_COUNT], ratio_max[CORE_COUNT];

unsigned long long cntr_iterations[CORE_COUNT];

atomic_t flag1, flag2;



struct queue{
   char buffer[BUFFER_LEN];
};

struct queue queue[QUEUE_LEN][CORE_COUNT];

int write_item[CORE_COUNT];
int read_item[CORE_COUNT];

spinlock_t lock[CORE_COUNT];

struct proc_dir_entry *proc_entry[CORE_COUNT];

int cpu_data[CORE_COUNT];
```

```
static __inline__ unsigned long long rdtsc(void){
   unsigned hi=0, lo=0;

   __asm__ __volatile__ ("rdtsc" : "=a"(lo), "=d"(hi));
   return ( (unsigned long long)lo)|( ((unsigned long long)hi)<<32 );
}


unsigned long long read_clock(int cpu){

   unsigned long long TSC = rdtsc();
   return clock[ALPHA][cpu] * (TSC - clock[TSC_0][cpu]) - clock[BETA][cpu];
}



void slave_sync(void *info){

   char buffer[BUFFER_LEN];

   preempt_disable();

   stamp1_1 = read_clock(slave_id);
   atomic_set(&flag1,1);
   stamp1_2 = read_clock(slave_id);

   while(!atomic_read(&flag2));


   stamp4 = read_clock(slave_id);


   ////Synchronization:

   t2_1 = stamp2_1 - ((stamp1_1/2) + (stamp1_2/2));
   t4_3 = stamp4 - ((stamp2_1/2) + (stamp2_2/2));

   offset = (t4_3 - t2_1)/2;

   ratio = abs(100 * ( (double)t4_3/t2_1 ));

   cntr_iterations[slave_id]++;
   if( ratio >= ratio_min[slave_id] && ratio <= ratio_max[slave_id] ){

      //Correction
      clock[BETA][slave_id] += offset;

      //Export offset value
      sprintf(buffer, "%lld,%lld", cntr_iterations[slave_id], offset);
      enqueue(buffer, slave_id);
   }
   if ( (ratio < 100) && (ratio > ratio_min[slave_id]) ){
      ratio_min[slave_id] = ratio;
      ratio_max[slave_id] = 100 + (100 - ratio);
   }
   if ( (ratio > 100) && (ratio < ratio_max[slave_id]) ){
      ratio_max[slave_id] = ratio;
      ratio_min[slave_id] = 100 - (ratio - 100);
   }
   if(ratio == 100)
      ratio_min[slave_id] = ratio_max[slave_id] = 100;


   atomic_set(&flag1,0);
   atomic_set(&flag2,0);

   if(slave_id == (CORE_COUNT-1) ){
      slave_id = 1;
   }
   else slave_id++;

   preempt_enable();
}


enum hrtimer_restart sync(struct hrtimer *timer) {

   ktime_t period_sync;

   preempt_disable();
   //int this_cpu = smp_processor_id();

   smp_call_function_single_async(slave_id, &csd);

   while(!atomic_read(&flag1));
   stamp2_1 = read_clock(MASTER_CPU);
   atomic_set(&flag2,1);
   stamp2_2 = read_clock(MASTER_CPU);



   period_sync = ktime_set(PERIOD_SYNC_sec, PERIOD_SYNC_nsec);
   hrtimer_forward_now(timer, period_sync);

   preempt_enable();
   return HRTIMER_RESTART;
}



void get_init(void *info){

   int this_cpu = smp_processor_id();
   clock[TSC_0][this_cpu] = rdtsc();
}



int thread_master_fn(void *data) {

   struct hrtimer timer_sync;
   ktime_t period_sync;


   //Broadcast IPI to set TSC_0 for each cpu
   smp_call_function(get_init, NULL, 0);
   clock[TSC_0][MASTER_CPU] = rdtsc();


   period_sync = ktime_set(PERIOD_SYNC_sec, PERIOD_SYNC_nsec);
   hrtimer_init(&timer_sync, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
   timer_sync.function = &sync;
   hrtimer_start(&timer_sync, period_sync, HRTIMER_MODE_REL);


   set_current_state(TASK_INTERRUPTIBLE);
   while (!kthread_should_stop()) {
      schedule();
      set_current_state(TASK_INTERRUPTIBLE);
   }
   set_current_state(TASK_RUNNING);

   hrtimer_cancel(&timer_sync);
   return 0;
}






static void increment(int * item) {
   *item = *item + 1;
   if(*item >= QUEUE_LEN){
      *item = 0;
   }
}



static int is_empty(int r, int w) {
   //empty if r == w, otherwise  r != w
   return !(r ^ w); //xor
}


static int is_full(int r, int w) {
   int write = w;
   increment(&write);
   return write == r;
}


static int dequeue (int cpu, char *buffer) {
   int ret = 0;
   spin_lock(&lock[cpu]);
   if( !is_empty(read_item[cpu],write_item[cpu]) ){
      strcpy(buffer, queue[read_item[cpu]][cpu].buffer);
      increment(&read_item[cpu]);
      ret = 1;
   }
   spin_unlock(&lock[cpu]);
   return ret;
}
```

```
static int enqueue (char *buffer, int cpu) {
   spin_lock(&lock[cpu]);
   if(is_full(read_item[cpu], write_item[cpu]))
      increment(&read_item[cpu]);

   strcpy(queue[write_item[cpu]][cpu].buffer,buffer);

   increment(&write_item[cpu]);
   spin_unlock(&lock[cpu]);
   return 1;
}



int proc_open(struct inode *inode, struct file *filp) {
   //gets void* .data field of respective proc_entry
   filp->private_data = PDE_DATA(inode);
   return 0;
}


ssize_t  proc_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
   char buffer[BUFFER_LEN];
   int ret = 0, len = 0;

   int cpu = *((int *)filp->private_data); //Get cpu from file data

   if(!dequeue(cpu, buffer))
      return 0;

   len = strlen(buffer);
   if(len <= 0)
      return -EFAULT;

   if(count < len)
      return -EFAULT;

   ret = copy_to_user(buf,buffer,len);
   if(ret != 0)
      return -EFAULT;

   *f_pos += count - len;
   return len;
}
```

```
int proc_close(struct inode *inode, struct file *filp) {
   //printk(KERN_INFO "[%d] release\n", current->pid);
   return 0;
}



static const struct file_operations proc_fops = {
   .owner  = THIS_MODULE,
   .open   = proc_open,
   .read   = proc_read,
   .release   = proc_close,
};






int mod_init(void) {

   char name[12];
   int id, j;

   slave_id=1;

   csd.info = NULL;
   csd.func = slave_sync;

   atomic_set(&flag1,0);
   atomic_set(&flag2,0);

   printk(KERN_INFO " INIT \n");


   for (id = 0; id < CORE_COUNT; id++) {
      for(j=0; j<3; j++)
         clock[j][id] = 0;
      clock[ALPHA][id] = 1;
   }
   for (id = 1; id < CORE_COUNT; id++) {
      cntr_iterations[id] = 0;

      ratio_min[id] = 80;
      ratio_max[id] = 120;

      // /proc entrys to export each clock rate
      sprintf(name, "clock_cpu%d", id);
      cpu_data[id] = id;
      proc_entry[id] = proc_create_data(name, 0666,
                                        NULL, &proc_fops,
                                        (void*) &(cpu_data[id]) );
      if(proc_entry[id] == NULL)
         return -ENOMEM;
      spin_lock_init(&lock[id]);
      write_item[id] = 0;
      read_item[id]  = 0;
   }

   thread_master = kthread_create(thread_master_fn, NULL, "Kthread_sync");
   if (thread_master) {
      kthread_bind(thread_master, MASTER_CPU); //bind to the cpu Master
      wake_up_process(thread_master);  //start thread
   }
   return 0;
}

void mod_end(void) {

   int id;

   if(!kthread_stop(thread_master))
      printk(KERN_INFO "Master Kthread TERMINATED \n");

   for (id = 1; id < CORE_COUNT; id++) {
      spin_unlock(&lock[id]);
      proc_remove(proc_entry[id]);
   }
}


MODULE_LICENSE("GPL");

module_init(mod_init);
module_exit(mod_end);
```

A.2 Inter-Processor Interrupts

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/kthread.h>
#include <linux/sched.h>
#include <linux/hrtimer.h>
#include <linux/smp.h>
#include <linux/proc_fs.h>
#include <linux/delay.h>
#include <asm/uaccess.h>


#define CORE_COUNT 4
#define MASTER_CPU 0
#define TSC_KHz 2659983


#define CNTR 0
#define TSC_0 1
#define BETA 2
#define ALPHA 3

#define PERIOD_SYNC_sec 0
#define PERIOD_SYNC_nsec 100000000

#define BUFFER_LEN 100
#define QUEUE_LEN 50


static void increment(int * item);
static int is_full(int r, int w);
static int enqueue (char *buffer, int cpu);


struct task_struct *thread_master;

long long clock[4][CORE_COUNT];


struct call_single_data csd1, csd2;

unsigned int slave_id;
unsigned long long stamp1_1, stamp1_2,
                   stamp2_1, stamp2_2,
                   stamp4;

signed long long t2_1, t4_3, offset;
long ratio;
long ratio_min[CORE_COUNT], ratio_max[CORE_COUNT];

unsigned long long cntr_iterations[CORE_COUNT];

struct queue{
   char buffer[BUFFER_LEN];
};

struct queue queue[QUEUE_LEN][CORE_COUNT];

int write_item[CORE_COUNT];
int read_item[CORE_COUNT];

spinlock_t lock[CORE_COUNT];

struct proc_dir_entry *proc_entry[CORE_COUNT];

int cpu_data[CORE_COUNT];



static __inline__ unsigned long long rdtsc(void){
   unsigned hi=0, lo=0;

   __asm__ __volatile__ ("rdtsc" : "=a"(lo), "=d"(hi));
   return ( (unsigned long long)lo)|( ((unsigned long long)hi)<<32 );
}


unsigned long long read_clock(int cpu){

   unsigned long long TSC = rdtsc();
   return clock[ALPHA][cpu] * (TSC - clock[TSC_0][cpu]) - clock[BETA][cpu];
}
```

```
void sync2(void *info){ //IPI handler 2 -> in master

   char buffer[BUFFER_LEN];

   stamp4 = read_clock(MASTER_CPU);


   ////Synchronization:

   t2_1 = stamp2_1 - ((stamp1_1/2) + (stamp1_2/2));
   t4_3 = stamp4 - ((stamp2_1/2) + (stamp2_2/2));

   offset = (t2_1 - t4_3)/2;

   ratio = abs(100 * ( (double)t4_3/t2_1 ));

   cntr_iterations[slave_id]++;

   if( ratio >= ratio_min[slave_id] && ratio <= ratio_max[slave_id] ){

      //Correction
      clock[BETA][slave_id] += offset;

      //Export offset value
      sprintf(buffer, "%lld,%lld", cntr_iterations[slave_id], offset);
      enqueue(buffer, slave_id);
   }

   if ( (ratio < 100) && (ratio > ratio_min[slave_id]) )
      ratio_min[slave_id] = ratio;

   if ( (ratio > 100) && (ratio < ratio_max[slave_id]) )
      ratio_max[slave_id] = ratio;

   if(ratio == 100)
      ratio_min[slave_id] = ratio_max[slave_id] = 100;


   if(slave_id == (CORE_COUNT-1) ){
      slave_id = 1;
   }
   else slave_id++;
}
```

```
void sync1(void *info){ //IPI handler 1 -> in slave

   stamp2_1 = read_clock(slave_id);
   smp_call_function_single_async(0, &csd2);
   stamp2_2 = read_clock(slave_id);
}



enum hrtimer_restart sync(struct hrtimer *timer) {

   ktime_t period_sync;
   //int this_cpu = smp_processor_id();

   stamp1_1 = read_clock(MASTER_CPU);
   smp_call_function_single_async(slave_id, &csd1);
   stamp1_2 = read_clock(MASTER_CPU);



   period_sync = ktime_set(PERIOD_SYNC_sec, PERIOD_SYNC_nsec);
   hrtimer_forward_now(timer, period_sync);

   return HRTIMER_RESTART;
}



void get_init(void *info){

   int this_cpu = smp_processor_id();
   clock[TSC_0][this_cpu] = rdtsc();
}




int thread_master_fn(void *data) {

   struct hrtimer timer_sync;
   ktime_t period_sync;


   smp_call_function(get_init, NULL, 0);
   clock[TSC_0][MASTER_CPU] = rdtsc();
   period_sync = ktime_set(PERIOD_SYNC_sec, PERIOD_SYNC_nsec);
   hrtimer_init(&timer_sync, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
   timer_sync.function = &sync;
   hrtimer_start(&timer_sync, period_sync, HRTIMER_MODE_REL);


   set_current_state(TASK_INTERRUPTIBLE);
   while (!kthread_should_stop()) {
      schedule();
      set_current_state(TASK_INTERRUPTIBLE);
   }
   set_current_state(TASK_RUNNING);

   hrtimer_cancel(&timer_sync);
   return 0;
}



static void increment(int * item) {
   *item = *item + 1;
   if(*item >= QUEUE_LEN){
      *item = 0;
   }
}



static int is_empty(int r, int w) {
   //empty if r == w, otherwise  r != w
   return !(r ^ w); //xor
}



static int is_full(int r, int w) {
   int write = w;
   increment(&write);
   return write == r;
}
```

```
static int dequeue (int cpu, char *buffer) {
   int ret = 0;
   spin_lock(&lock[cpu]);
   if( !is_empty(read_item[cpu],write_item[cpu]) ){
      strcpy(buffer, queue[read_item[cpu]][cpu].buffer);
      increment(&read_item[cpu]);
      ret = 1;
   }
   spin_unlock(&lock[cpu]);
   return ret;
}
```

```
static int enqueue (char *buffer, int cpu) {
   spin_lock(&lock[cpu]);
   if(is_full(read_item[cpu], write_item[cpu]))
      increment(&read_item[cpu]);

   strcpy(queue[write_item[cpu]][cpu].buffer,buffer);

   increment(&write_item[cpu]);
   spin_unlock(&lock[cpu]);
   return 1;
}
```

```
int proc_open(struct inode *inode, struct file *filp) {
   //gets void* .data field of respective proc_entry
   filp->private_data = PDE_DATA(inode);
   return 0;
}
```

```
ssize_t  proc_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
   char buffer[BUFFER_LEN];
   int ret = 0, len = 0;

   int cpu = *((int *)filp->private_data); //Get cpu from file data

   if(!dequeue(cpu, buffer))
      return 0;

   len = strlen(buffer);
   if(len <= 0)
      return -EFAULT;

   if(count < len)
      return -EFAULT;

   ret = copy_to_user(buf,buffer,len);
   if(ret != 0)
      return -EFAULT;

   *f_pos += count - len;
   return len;
}




int proc_close(struct inode *inode, struct file *filp) {
   //printk(KERN_INFO "[%d] release\n", current->pid);
   return 0;
}




static const struct file_operations proc_fops = {
   .owner  = THIS_MODULE,
   .open   = proc_open,
   .read   = proc_read,
   .release   = proc_close,
};
```

```
int mod_init(void) {

   char name[12];
   int id, j;
   slave_id=1;

   csd1.info = NULL;
   csd1.func = sync1;
   csd2.info = NULL;
   csd2.func = sync2;

   printk(KERN_INFO " INIT \n");

   for (id = 0; id < CORE_COUNT; id++) {

      for(j=0; j<3; j++)
         clock[j][id] = 0;
      clock[ALPHA][id] = 1;
   }

   for (id = 1; id < CORE_COUNT; id++) {

      ratio_min[id] = 90;
      ratio_max[id] = 110;

      // /proc entrys to export each clock rate
      sprintf(name, "clock_cpu%d", id);
      cpu_data[id] = id;
      proc_entry[id] = proc_create_data(name, 0666,
                                        NULL, &proc_fops,
                                        (void*) &(cpu_data[id]) );
      if(proc_entry[id] == NULL)
         return -ENOMEM;
      spin_lock_init(&lock[id]);
      write_item[id] = 0;
      read_item[id]  = 0;
   }

   thread_master = kthread_create(thread_master_fn, NULL, "Kthread_sync");
   if (thread_master) {
      kthread_bind(thread_master, MASTER_CPU); //bind to the cpu Master
      wake_up_process(thread_master);  //start thread
   }
   return 0;
}
```

```
void mod_end(void){

   int id;

   if(!kthread_stop(thread_master))
      printk(KERN_INFO "Master Kthread TERMINATED \n");

   for (id = 1; id < CORE_COUNT; id++) {
      spin_unlock(&lock[id]);
      proc_remove(proc_entry[id]);
   }
}


MODULE_LICENSE("GPL");

module_init(mod_init);
module_exit(mod_end);
```

References

[1] E.G. Friedman. Clock distribution networks in synchronous digital integrated circuits. *Proceedings of the IEEE*, 89(5):665–692, 2001.

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

[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*. PhD thesis, Texas A&M University, 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*, pages 87–96, 2006.

[6] D Wiklund. Mesochronous clocking and communication in on-chip networks. *Proc. Swedish System-on-Chip Conf*, pages 3–6, 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):29–41, 2008.

[8] Karl-filip Faxén. Multicore computing — the state of the art. 1:1–36, 2008.

[9] András Vajda. Multi-core and Many-core Processor Architectures. In *Programming Many-Core Chips*, page 241. 2011.

[10] Intel Corporation. Intel® Many Integrated Core Architecture (Intel® MIC Architecture). URL: `http://goo.gl/8sSXBW`.

[11] Daniel P. Bovet and Marco Cesati. Timing Measurements. In *Understanding the Linux Kernel*, chapter 6, pages 227–257. O'REILLY, 3rd edition, 2005.

[12] Robert Love. Timers and Time Management. In *Linux Kernel Development Third Edition*, chapter 11, pages 207–230. Sams Publishing, 3rd edition, 2005.

[13] Wolfgang Mauerer. Time Management. In *Professional Linux ® Kernel Architecture*, chapter 15, pages 893–948. Wiley Publishing, 2008.

[14] STMicroelectronics. Str71X Real Time Clock Application Example. pages 1–9, 2004.

[15] Intel Corporation. Accessing Real Time Clock Registers and NMI Enable Bit, 2009. URL: http://www.intel.com/content/www/us/en/intelligent-systems/software/real-time-clock-nmi-enable-paper.html.

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

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

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

[19] Vmware E S X Esxi and Vmware Workstation. Timekeeping in VMware Virtual Machines. *White papers Latest revision 12 Aug*, pages 1–26, 2008.

[20] "Brendan", "Alexmode", and "Max". OSDev.org - APIC Timer, 2012. URL: http://wiki.osdev.org/APIC_timer.

[21] Intel Corporation. ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER (APIC). In *Intel ® 64 and IA-32 Architectures Software Developer ' s Manual Volume 3A : System Programming Guide , Part 1*, volume 3, chapter 10. 2015.

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

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

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

[25] Kernel.org. Timekeeping.txt - Clock sources, Clock events, sched_clock() and delay timers. URL: https://www.kernel.org/doc/Documentation/timers/timekeeping.txt.

[26] Embedded Linux Experts. Free Electrons. URL: http://free-electrons.com/.

[27] Redhat.com. Red Hat Enterprise MRG 2 - Chapter 15. Timestamping, 2013. URL: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_MRG/2/html/Realtime_Reference_Guide/chap-Realtime_Reference_Guide-Timestamping.html.

[28] John Stultz. LKML.org, 2011. URL: https://lkml.org/lkml/2011/2/15/979.

[29] John Stultz. time: move leap second management into timekeeping core, 2012. URL: http://lkml.iu.edu/hypermail/linux/kernel/1205.2/03005.html.

[30] "corbet". The high-resolution timer API, 2006. URL: http://lwn.net/Articles/167897/.

[31] "csimmonds". Over and over again: periodic tasks in Linux, 2009. URL: http://www.2net.co.uk/tutorial/periodic_threads.

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

[33] David L. Mills. *Network Time Synchronization: the Network Time Protocol on Earth and in Space*. 2nd edition, 2011. URL: `http://www.eecis.udel.edu/~mills/book.html`.

[34] National Institute of Standards and Technology. Introduction to IEEE 1588, 2009. URL: `http://www.nist.gov/el/isd/ieee/intro1588.cfm`.

[35] John Eidson (agilent). IEEE-1588 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems - A Tutorial -, 2005. URL: `http://www.nist.gov/el/isd/ieee/upload/tutorial-basic.pdf`.

[36] John Rushby. Bus Architectures for Safety-Critical Embedded Systems. *Embedded Software*, (October):306–323, 2001.

[37] Intel Corporation. Intel Processor Identification and the CPUID Instruction. (January), 1999.

[38] Intel Corporation. TIME-STAMP COUNTER. In *Intel ® 64 and IA-32 Architectures Software Developer ' s Manual Volume 3B : System Programming Guide , Part 2*, volume 3, chapter 17.13, pages 140–142. 2015.

[39] Sergiu Iordache and Terry Lambert. TSC resynchronization. URL: `https://www.chromium.org/chromium-os/how-tos-and-troubleshooting/tsc-resynchronization`.

[40] Darren Hart. Linux Test Project, 2006. URL: `http://sourceforge.net/p/ltp/git/ci/d2a255e0d8a285431ade06178e6b8edf8e6a9743/tree/testcases/realtime/func/measurement/rdtsc-latency.c`.

[41] Jay ; D. CPU TSC fetch operation especially in multicore-multi-processor environment. URL: `stackoverflow.com/questions/10921210/cpu-tsc-fetch-operation-especially-in-multicore-multi-processor-environment`.

[42] Martin G Dixon, Jeremy J. Shrall, and Rajesh S. Parthasarathy. Controlling Time Stamp Counter (TSC) Offsets For Mulitple Cores And Threads, 2011. URL: `http://www.google.com/patents/US20110154090`.

[43] Robert Love. Kernel Synchronization Methods. In *Linux Kernel Development Third Edition*, chapter 10, pages 175–180. 2005.

[44] Md. Arifur Rahman, Thomas Kunz, and Howard Schwartz. Delay Asymmetry Correction Model for Master-Slave Synchronization Protocols. *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*, pages 1–8, 2014.

[45] Tux Think. Kernel Thread Creation. URL: `http://tuxthink.blogspot.pt/2011/03/kernel-thread-creation-3.html`.

[46] Binh Nguyen. Linux Filesystem Hierarchy, 2004. URL: `http://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html`.

[47] Alessandro Rubini and Jonathan Corbet. Chapter 3 - Char Drivers. In *Linux Device Drivers*, chapter 3, page 564. 2005. URL: `http://lwn.net/images/pdf/LDD3/ch03.pdf`.

[48] Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158, 1989.