Master of Science Thesis

Enhancing the Monitoring of Real-Time Performance in Linux

Author: Nima Asadi
nai10001@student.mdh.se

Supervisor: Mehrdad Saadatmand
mehrdad.saadatmand@mdh.se

Examiner: Mikael Sjödin
mikael.sjodin@mdh.se

Sweden, April 2014
This thesis is dedicated to my parents for their constant love, support and encouragement
Acknowledgements

I would like to thank my supervisor Mehrdad Saadatmand for his guidance and help during the helpful discussions that we had in our meetings. Furthermore, I would like to express my thankfulness to Mikael Sjödin for his help in publishing a scientific paper based on this thesis work with his comments and feedback. Also, I want to thank Nima Moghaddami Khalilzad for his very insightful advice on the design and modification of the real-time module.
Abstract

There is a growing trend in applying Linux operating system in the domain of embedded systems. This is due to the important features that Linux benefits from, such as being open source, its light weight compared to other major operating systems, its adaptability to different platforms, and its more stable performance speed. However, there are upgrades that still need to be done in order to use Linux for real-time purposes. A number of different approaches have been suggested in order to improve Linux’s performance in real-time environment. Nevertheless, proposing a correct-by-construction system is very difficult in real-time environment, mainly due to the complexity and unpredictability of them. Thus, run-time monitoring can be a helpful approach in order to provide the user with data regarding to the actual timing behavior of the system which can be used for analysis and modification of it. In this thesis work, a design for run-time monitoring is suggested and implemented on a real-time scheduler module that assists Linux with real-time tasks. Besides providing crucial data regarding the timing performance of the system, this monitor predicts violations of timing requirements based on the current trace of the system performance.

Keywords: Real-time, Monitoring, Linux, Operating system, Scheduling, Latency
Contents

Acknowledgements ii

Abstract iii

Contents iv

List of Figures vi

1 Introduction 1
  1.1 Problem Statement ...................................... 1
  1.2 Goals .................................................. 2
  1.3 Thesis Structure ....................................... 3
  1.4 Report Structure ....................................... 3

2 Background 4
  2.1 An Introduction to Real-Time Systems .................. 4
     2.1.1 Real-time Operating System Kernel ................. 4
     2.1.2 Tasks ............................................. 5
     2.1.3 Scheduling ........................................ 7
  2.2 A Short Overview of the Commonly Used RTOSs .......... 9
     2.2.1 Interrupt Abstraction ............................ 11
     2.2.2 RTLinux .......................................... 12
     2.2.3 RTAI ............................................... 13
     2.2.4 Xenomai ........................................... 14
     2.2.5 Reducing the Latencies ........................... 14
     2.2.6 Preemptive Kernel Patch .......................... 15
     2.2.7 Low latency Patch ................................ 17
     2.2.8 Monta Vista ....................................... 17
     2.2.9 TimeSys .......................................... 18
     2.2.10 Preempt_rt Patch ................................ 18
     2.2.11 LITMUSRT ........................................ 19
     2.2.12 Enhancing the Timing Resolution .................. 21
  2.3 An Overview of the Common Approaches Toward a Real-time Linux .... 11
     2.3.1 Interrupt Abstraction ................................ 11
     2.3.2 RTLinux ........................................... 12
     2.3.3 RTAI ............................................... 13
     2.3.4 Xenomai ........................................... 14
     2.3.5 Reducing the Latencies ........................... 14
     2.3.6 Preemptive Kernel Patch .......................... 15
     2.3.7 Low latency Patch ................................ 17
     2.3.8 Monta Vista ....................................... 17
     2.3.9 TimeSys .......................................... 18
     2.3.10 Preempt_rt Patch ................................ 18
     2.3.11 LITMUSRT ........................................ 19
     2.3.12 Enhancing the Timing Resolution .................. 21
  2.4 Monitoring ............................................. 23
     2.4.1 Linux Native Monitoring Tools ..................... 24
        2.4.1.1 Sysstat Project ............................... 24
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.1.2</td>
<td>Mpstat</td>
<td>25</td>
</tr>
<tr>
<td>2.4.1.3</td>
<td>Pid Stat</td>
<td>26</td>
</tr>
<tr>
<td>2.4.1.4</td>
<td>Top</td>
<td>27</td>
</tr>
<tr>
<td>2.4.1.5</td>
<td>Vmstat</td>
<td>29</td>
</tr>
<tr>
<td>2.4.1.6</td>
<td>Gnome Desktop Environment Tool</td>
<td>30</td>
</tr>
<tr>
<td>2.4.1.7</td>
<td>KDE System Guard</td>
<td>30</td>
</tr>
<tr>
<td>2.4.1.8</td>
<td>Proc File System</td>
<td>33</td>
</tr>
<tr>
<td>2.4.1.9</td>
<td>Summary</td>
<td>34</td>
</tr>
<tr>
<td>2.4.2</td>
<td>An Overview of Related Work on Monitoring of Real-Time Performance</td>
<td>35</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Hardware Monitors</td>
<td>36</td>
</tr>
<tr>
<td>2.4.3.1</td>
<td>Noninterference Method</td>
<td>36</td>
</tr>
<tr>
<td>2.4.3.2</td>
<td>PASM</td>
<td>37</td>
</tr>
<tr>
<td>2.4.3.3</td>
<td>ART</td>
<td>37</td>
</tr>
<tr>
<td>2.4.4</td>
<td>Software Monitors</td>
<td>38</td>
</tr>
<tr>
<td>2.4.4.1</td>
<td>PMMS</td>
<td>38</td>
</tr>
<tr>
<td>2.4.4.2</td>
<td>Halsall-Hui</td>
<td>38</td>
</tr>
<tr>
<td>2.4.4.3</td>
<td>OSE Monitor</td>
<td>39</td>
</tr>
<tr>
<td>2.4.5</td>
<td>Hybrid Monitors</td>
<td>39</td>
</tr>
<tr>
<td>2.4.5.1</td>
<td>Hybrid Monitor</td>
<td>39</td>
</tr>
<tr>
<td>2.4.5.2</td>
<td>ZM4</td>
<td>40</td>
</tr>
<tr>
<td>2.4.5.3</td>
<td>SoC-based Monitor</td>
<td>40</td>
</tr>
<tr>
<td>2.4.6</td>
<td>Summary</td>
<td>40</td>
</tr>
</tbody>
</table>

### 3 Solution

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Module</td>
<td>43</td>
</tr>
<tr>
<td>3.1.1</td>
<td>ExSched</td>
<td>43</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Communication Between Tasks (API)</td>
<td>44</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Tasks and Queues</td>
<td>45</td>
</tr>
<tr>
<td>3.2</td>
<td>Monitor</td>
<td>46</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Design Features</td>
<td>46</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Design Limitations</td>
<td>48</td>
</tr>
<tr>
<td>3.3</td>
<td>Implementation</td>
<td>49</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Design Features</td>
<td>50</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Design Limitations</td>
<td>51</td>
</tr>
</tbody>
</table>

### 4 Evaluation

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
</table>

### 5 Summary and Future Work

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Summary</td>
<td>64</td>
</tr>
<tr>
<td>5.2</td>
<td>Future Work</td>
<td>64</td>
</tr>
</tbody>
</table>
List of Figures

2.1 A diagram of overall design structure of RTLinux  
2.2 A diagram of overall design structure of Xenomai  
2.3 A diagram of how timers are managed inside Linux kernel.  
2.4 An example of Top command output  
2.5 An example of Vmstat output  
2.6 An example of KDE System Guard output  
2.7 An example of Proc File System output  
3.1 Module  
3.2 A schematic view of the data collection and analysis process in the monitor  
3.3 Accessing Linux Kernel Buffer through applications  
3.4 A schematic view of the monitor in combination with the real-time module.  
3.5 As can bee seen, the completion time of T1 is getting closer to its deadline.  
3.6 Sequence Diagram of the Monitoring Process  
4.1 scheduling of T, T2, T3, and the monitoring task(M) with EDF  
4.2 The effect of a high frequency monitoring task on target system’s behavior  
4.3 T1, with a deadline equal to 1800, has a tardiness of 120 milliseconds.  
4.4 A part of the monitoring output for EDF scheduling of T1, T2, T3, and M  
4.5 scheduling of T1, T2, and the monitoring task (M) with FPS  
4.6 The measured amount of the overhead imposed by the monitoring thread with different periods  
4.7 Rate of deadline misses corresponding to period of the monitoring task
Chapter 1

Introduction

This thesis was carried out at the Intelligent Embedded Systems faculty of the IDT (Innovation, Design, and Engineering) school at Mälardalen University (MDH), a research university in Sweden. Linux is an open source operating system (OS) which is widely used in computers, mobile phones, cars, electrical appliances, and many other systems. While Linux OS has proven to have the advantages such as flexibility towards different systems and stable performance speed, it has also shown low predictability when it comes to real-time applications. This thesis aims to improve the monitoring capability of Linux’s timing behavior in real-time environments.

1.1 Problem Statement

Despite the development of static timing analysis approaches for achieving a correct-by-construction design for systems regarding their timing constraints, violation of those constraints can still happen in real-time environments due to various reasons. One important way to obtain crucial data regarding the actual timing behavior of systems in real world experience is run-time monitoring. However, despite the improvements in designing run-time monitors, significant amount of work needs to be done in optimizing these approaches regarding the main issues with these monitors. Such issues include latency that the monitor adds to the system, accuracy, device limitations, etc.

Linux operating system has proven to benefit from significant features that makes it adaptable to many frameworks. Nonetheless, like any other general-purpose operating systems, it is highly vulnerable to timing constraint violations in real-time environments. The effort in this thesis work is put on the design and
implementation of a timing behavior run-time monitor for Linux operating system with the emphasis of reducing the latency imposed by the monitor while providing comprehensible monitoring facility.

1.2 Goals

The goals of this thesis work are:

- Study of various methods suggested to enhance the real-time capabilities of Linux OS and different approaches suggested to design a mechanism for monitoring of timing behavior in a system
- Suggesting a low-overhead monitoring system to evaluate Linux OS performance in real-time environment
- Implement and evaluate the suggested solution for enhancing the monitoring capabilities of timing performance of Linux OS in real-time environment.

The goals of this thesis were obtained by:

- Study of concepts related to real-time operating systems
- Study of Linux kernel and evaluating its performance when dealing with real-time tasks
- Study of other common real-time operating systems which are especially used in embedded systems
- Study of approaches suggested for enhancing the real-time performance of Linux OS
- Study of suggested methods for monitoring of timing performance in different systems
- Suggestion of a mechanism to enhance the monitoring capabilities of real-time behavior of Linux OS
- Implementation of the monitoring method in the system with the assistance of a real-time module
- Evaluation of the results of the monitoring method regarding accuracy and latency
1.3 Thesis Structure

This thesis consists of two main parts. In the first part, focus has been put on studying the related work regarding different operating systems used in embedded systems, their flaws with handling real-time tasks, the enhancements made to them, and enhancements made to Linux regarding its real-time properties. Moreover, a more in depth survey of various methods and tools for run-time monitoring of timing behavior in real-time and non-real-time systems was conducted regarding this part, which can be found in [1]. In the second section, the design, implementation, and evaluation of a mechanism for run-time monitoring of timing behavior in Linux OS in real-time environment is presented.

1.4 Report Structure

The thesis report starts with an introduction in chapter 1. Chapter 2 can be divided in to three main parts; in the first part, an introduction to the necessary concepts and definitions in the field of real-time systems is presented. This introduction briefly covers different scheduling methods, task types, and other definitions related to this thesis work. In second part of chapter 2,a very brief overview of some commonly used real-time operating systems as well as common approaches suggested to enhance the real-time performance of Linux is presented. Monitoring of timing behavior is the focus of the third part of chapter 2, which covers a selected number of Linux native monitoring tools as well as a number of tools designed for monitoring of timing behavior in different systems. Chapter 3 is dedicated to the design of a real-time module developed for enhancing the real-time performance of Linux OS, and design and implementation of a mechanism for monitoring of timing behavior of Linux in real-time environments. In chapter 4, an evaluation of the suggested method is provided. The report is finished by a summary and discussion about the future work in chapter 5.
Chapter 2

Background

2.1 An Introduction to Real-Time Systems

The difference between a real-time operating system (RTOS) and a non-real-time operating system is that an RTOS is specially designed to run time-crucial applications within their timing constraints. For example, a vehicle engine control system needs to perform controlling actions on exact times in order to prevent catastrophic consequences. A medical heart signal displayer, an anti-lock brake, and a function controller in a factory are other examples of timing crucial tasks, also known as real-time tasks. Thus, it is clear that the maximum time used for an operation is a necessity for the RTOSs to have. These timing constraints are used and implemented in rules called scheduling policies that mainly prioritize tasks based on how crucial their timing features are. The real-time systems are commonly divided into two major groups: hard, and soft. In hard real-time systems the time an application is supposed to be executed plays the main role. In fact, if the system fails to perform a hard real-time application in its specified timing constraints, the consequence can be a catastrophe. On the other hand, a timing failure in performing a soft real-time application does not result in a disaster. An example of hard real-time applications can be the air bag system in a car or the auto-pilot system in an airplane, which must perform decisions on time otherwise the result will be disastrous. An example of a soft real-time applications is an application in a mobile phone, or a game, or live video stream.

2.1.1 Real-time Operating System Kernel

As the main component of an operating system, the kernel is responsible for connecting the hardware data process mechanism of the system to the application layer. Like most operating systems, the RTOS kernel is responsible for managing
different operations on tasks such as creation, execution, and deletion, according to the policy that the developers set for it. Also, like general-purpose kernels, the tasks of memory management, Interrupt handling, and managing I/O communications are performed by the RTOS kernel. However, the main difference between an RTOS kernel and a general-purpose kernel lies in the fact that the non-real-time kernels do not guarantee that they can perform a real-time application within its timing limitations. Typically, the main constituents of a kernel include the scheduler, interrupt handler, memory manager, and other services. A brief description of each of the mentioned components is given below.

- **Scheduler**: the scheduler has the managing role in the kernel. It allocates processor time to different tasks based on the scheduling policy (or policies) that are designed for it. A description of some of the common scheduling policies mentioned in this report is given later.

- **Interrupt handler**: the interrupt handler is responsible for managing the requests from different hardware tools as well as system calls. An interrupt handler can either be initiated by interrupts coming from the hardware or execution of interrupt codes in software.

- **Memory manager**: memory management section provides users of kernel resources with the location in system memory (memory allocation).

- **Other Services**: depending on the application requirement and type, and also the kernel design, a number of background activities occur when the kernel is operating. These activities include synchronization of shared resource services, supervision of the task resource allocation, and timing synchronization.

In next sections a description of basic concepts of the field of real-time systems is provided.

### 2.1.2 Tasks

A task represents a single process or multiple processes that the operating system must execute. A process is a single instance of a task. Processes receive their necessary resources from the operating system. A job is a set of tasks. In real-time world, each task is assigned a set of timing constraints such as release time, period, execution or computation time, response time, and deadline. In the sections below, some of the basic concepts used in real-time system analysis are explained.

- **Release Time**: the instance that a task gets ready for scheduling is called the release time of that task.
• Period: a task period is a time span after which the next task instance is released. This parameter is used for periodic tasks, i.e., the tasks in which each interval is started in regular interval.

• Execution Time: the time spent by the system to execute a task is called the task execution time.

• Response Time: the time span between a task release and its execution completion is called response time of that task.

• Deadline: a very important task parameter in real-time systems is task deadline. There are two terms regarding to deadline that are used commonly in real-time analysis: absolute deadline, and relative deadline. A task’s absolute deadline is the instance by which the task execution must be completed, and the relative deadline is defined as the maximum allowed response time of a task.

• Periodicity: a single instance of a task execution is called task instance. Some tasks show periodic behavior, meaning that the task instances execute within fixed time intervals. A monitor display update can be an example of a periodic task. However, some tasks do not show the regular behavior of periodic tasks. The reason for such behavior can stem from external interferences. For example, an external signal can be received by the system at any time, whether it comes from the environment or a pressed button. When there is no regular timing between the instances of a task, it is called an aperiodic task. Since because of the irregular nature of aperiodic tasks different instances of such tasks can enter the system in very short time intervals, meeting the deadline cannot be fully guaranteed. Consequently, such tasks are assigned soft deadlines. However, for another type of non-periodic tasks where the minimum time interval between the occurrences of two consecutive instances (Minimum Inter-Arrival Time, also known as MIAT) is given, a hard deadline can be used. Such tasks are called sporadic tasks.

• Task state: since a CPU time must be allocated to all of the tasks, most, or some of the tasks need to stay in a line to wait for their running time to come, where the amount of waiting depends on the tasks. The tasks in the waiting line are in a state called suspended state. Depending on the kernel structure, a task has different types of states. Three general states that a task could have are: ready, running, and suspended.
• Ready state: a task is placed in the ready state when it receives all the resources and data it needs for execution, and is waiting for the scheduler to let it run in its assigned CPU time.

• Running state: when the time reaches for the task to get executed, it is placed in running state.

• Suspended state: a task is in suspended state when it is waiting for the shared resources to be available, or an event to happen.

• Priority: since not all the tasks can be run at the same time, and also, some tasks can be more important than the other ones, each task in real-time systems is assigned a priority. Clearly, the priority of a task depends on its importance. Moreover, the priority of a task depends on the scheduling policy.

2.1.3 Scheduling

As stated before, different tasks are not able to use the CPU time at the same time, thus, there must be an order by which the CPU time is distributed among them. The task of distributing CPU time among all the tasks is performed by the scheduler. The order of execution depends on the policy, or policies, that the programmer designs the scheduler structure in. Below a description of the most common scheduling policies is presented. In general scheduling algorithms are divided into two major groups: offline scheduling and online scheduling. In offline scheduling all the tasks are scheduled before the execution starts. Clearly, in this method all the tasks to be executed by the system as well as the resources they need to be executed and timing constraints are known. While using this scheduling method the CPU time can be managed optimally, the disadvantage is that it does not take care of tasks that are created during the execution. Also, this method can only manage periodic tasks. In online scheduling however planning is done during the system execution time. In this method task characteristics and needs are presented to the scheduler when the task enters the schedule. An advantage of online scheduling is that it can handle all periodic, aperiodic, and sporadic tasks. However, due to run time computation the scheduling overhead is high. There is also another common classification of scheduling algorithms which places them into periodic or non-periodic tasks algorithms. Rate monotonic scheduling (RMS) and earliest deadline first (EDF) are two most famous periodic algorithms that are briefly described below.

• Rate Monotonic Scheduling:
In rate monotonic scheduling the priorities of tasks depend on their period. The shorter the task period, the higher its priority is. Thus, tasks in this method have static priority, meaning that they keep their priority during the whole system execution time.

- Earliest Deadline Frist:

In EDF scheduling policy, whenever a number of task instances are ready to be executed, the scheduler looks for the task instance which is closest to its deadline, and executes it first to make sure that no task loses its deadline. However, the higher overhead of deadline calculations in this method, especially when the system is overloaded, is a disadvantage of this model.

The non-periodic tasks however need different scheduling approaches, because even their release time is not known. The most common scheduling algorithms for non-periodic tasks is presented here:

- Background Scheme:

In background scheme the non-periodic tasks plays the role of a background task, meaning that they are scheduled only if there are no ready periodic tasks. Consequently, only the scheduler idle time is allocated to non-periodic tasks. Also, whenever a periodic task is ready again, the scheduler halts the non-periodic task, and runs the periodic task. While this method is easy to implement, it has the disadvantage that it cannot guarantee that all non-periodic tasks meet their deadlines.

- Polling Server:

In polling server, the periodic task plays the role of a server. Non-periodic tasks are assigned a slice of CPU time, called execution budget. The non-periodic tasks are scheduled to be run in time intervals equal to their execution budgets. This execution budget is consumed by the tasks at rate of 1 per time unit. The non-periodic tasks can be preempted if their execution budget is replenished. When the polling server is released again, the budget is refueled again to its execution time, unless it does not find any non-periodic tasks. In case the polling server does not find any non-periodic tasks its budget is set to zero.

- Deferrable Server:

The aim of using deferrable server is to improve the response time of non-periodic tasks in polling servers. In this method, instead of setting the budget to zero when
there is no non-periodic tasks at the beginning of the server release, the unused budget is retained throughout the server period in order to be used whenever there are aperiodic tasks. Note that the budge in this server is not transferred from one period to the next one.

2.2 A Short Overview of the Commonly Used RTOSs

- Free RTOS

Free RTOS is a famous and simple real-time operating system. This operating system is mainly written in C programming language. Also, making use of a simple and small kernel makes this OS compatible for embedded systems. It also has shown to have low overhead.

Task attributes: priority

The scheduling policy: fixed-Priority preemptive scheduling, Round-robin(RR) for threads with the same priority

- Enea OSE

Enea OSE is a product of the Swedish firm, Enea AB. This compact real-time operating system has been widely used in mobile phones. A compatible platform for this OS is the ARM family. Also, The multicore version of this OS, Enea OSE Multicore, was released in 2009 based on the same kernel architecture as Enea OSE.

Task attribute: priority

Scheduling policy: fixed-Priority preemptive scheduling, periodic scheduling, Round-robin scheduling for tasks with the same priority

- Windows Embedded CE

Windows Embedded CE, also known as WinCE, is a Microsoft product developed for embedded systems and microcontrollers. This OS, which was written in C programming language, was first released in 1996, and currently, the seventh version of it is available. Obviously, since Windows CE is designed for embedded systems, it uses a small memory size. Many later platforms such as AutoPC, Windows Mobile 2003, and Windows Phone were developed based on Windows CE kernel.

Task attribute: priority, 256 priorities

Scheduling: fixed priority preemptive scheduling, it only supports one level of priority inversion.

- QNX

QNX is another real-time OS that aims mainly embedded systems. This operating system which was first released in 1982 by Research In Motion, is a Unix-like OS which executes most of the operations in form of small tasks called servers.
Task attributes: no WCET or non-functional attributes specifiable Scheduling policies: FIFO scheduling, Round-robin scheduling, Adaptive scheduling

- Lynx OS

Another Unix-like operating system written in C, C++, and Ada languages is Lynx OS. Lynx was first released in 1986, and currently, the fifth version of this OS is available for users. Lynx is a closed source OS. Simple Priority based scheduling Publishes accurate real-time performance numbers that are in accordance with the system’s performance in real-world situations such as interrupt latency, context switch time, preemption time, and many others.
Scheduling policy: fixed priority preemptive scheduling

- VxWorks

VxWorks is a 64-bit real-time operating system that can be used on embedded systems. Supporting shared resources services as well as local and distributed message queues and benefiting from a compact design have made this RTOS useful in different industries that make use of embedded systems, such as in airplane and vehicle production.
Task attribute: priority Scheduling policy: preemptive fixed priority scheduling (256 priority levels), Round robin, Can use Posix (FIFO and RR)

- RT Linux

RT Linux is a modified version of Linux OS that was first released in 2007. The main objective of this kernel is to provide support for hard real-time tasks. Furthermore, transparency, and modularity are important points in the development of RT Linux. Linux RT is written in C programming language.
Scheduling policy: EDF(Earliest deadline first), RM(Rate monotonic), can use Posix(FIFO and RR)

- Xenomai

Xenomai is another project aiming at providing real-time support for Linux OS. There are major difference in goals and design of Xenomai and RT Linux which will be discussed in next chapter. Xenomai is compatible for common platforms such as ARM and X86.
Task attribute: priority at the time of creation Scheduling policy: fixed priority preemptive scheduling. FIFO and Round-robin for tasks with the same priority are supported as well
• RTAI

RTAI (Real-time application interface) was developed as a real-time extension for Linux OS. This project which is compliant to POSIX (Portable Operating System Interface), uses a real-time abstraction layer architecture for handling real-time tasks. Discussion about the architecture of RTAI is provided in next chapter.

Task attribute: priority (16 bit integers with 0 being the highest priority) Scheduling policy: fixed priority preemptive scheduling for both periodic and non-periodic tasks

2.3 An Overview of the Common Approaches Toward a Real-time Linux

A number of solutions have been suggested for making Linux compatible with real-time environments. In this chapter, an overview of the most common solutions for improving the real-time performance of the Linux operating system is presented.

2.3.1 Interrupt Abstraction

The Interrupt abstraction method [2] is based on adding a virtual layer between the kernel and the hardware. This layer, which is called Real-Time Hardware Abstraction Layer (RTHAL), takes the full control of the interrupts and the system timers. This process is done by dividing the interrupt s coming from real-time sources and non-real-time sources. When an interrupt occurs the RHTAL checks if it comes from a real-time source, meaning that it needs to be served immediately. If the interrupt is real-time, it is forwarded to Linux kernel without any hesitations, but if it comes from a non-real-time source, the interrupt is placed in a pending vector along with all other Linux activities, and if there are no real-time tasks, it will be executed. As a matter of fact, all the ordinary Linux threads are given the lowest priority by the abstraction layer, and the real-time events have the high priority. In order to take the full control of the interrupts, the RTHAL renews the cli (disable interrupt) and sti (enable interrupt) function calls so that Linux cannot deactivate the hardware interrupt. When adding an interrupt abstraction layer to the Linux kernel structure provides possibilities such as lowering the latency, it also has disadvantages such as lack of memory protection between the real-time tasks and the Linux kernel due to the fact that the real-time section executes in the same memory space as the Linux kernel codes [3]. In the next sub-sections two main approaches based on Interrupt Abstraction technique, RTLinux and Real-Time Application Interface(RTAI), are discussed.
2.3.2 RTLinux

In RTLinux [2], an interrupt abstraction layer is placed between the Linux kernel and the hardware. A preemptive fixed-priority scheduler takes care of both the real-time tasks and the Linux kernel activities, which run as the lowest priority task. As described before, real-time tasks have the privilege to have direct access to the Linux kernel, meaning that if an interrupt is coming from a real-time source, it is forwarded to the Linux kernel; otherwise it is held in a pending vector. A real-time task in RTLinux is created as a kernel module which can be dynamically loaded, so they do not occupy the virtual memory. When the module is loaded it informs the scheduler about the release time, the deadline, and the period of the task. For starting a real-time task instead of using the system calls, the initialization code is used, which in term gives the information about the three mentioned attributes of a task to the RTLinux kernel [3]. These real-time tasks should be created as simple as possible and the non real-time parts of the code should be left to the user space. Moreover, being compliant with the POSIX 1003.13 the programmers do not have to learn a new API. Though, a disadvantage of the RTLinux is that it is covered by US Patent 5885745, which is not valid outside the USA, which has motivated the programmers to turn to RTAI.

![Figure 2.1: A diagram of overall design structure of RTLinux](image-url)
2.3.3 RTAI

RTAI [2] has proven to be a robust and mature approach which has been developed constantly by the community developers. The RTAI project is almost entirely module-based and the kernel patch changes only a few lines of code in the standard Linux kernel. This is best explained in [4]: “it should be noted that Linux is almost unaffected by RTHAL, except for a slight (and negligible) loss of performance due to calling of cli and sti related functions in place of their corresponding hardware function calls, and due to the use of function pointers instead of directly linked functions.”

That real-time tasks are built as modules has the advantage that the RTOS and the real-time task share the same execution space and the system call, therefore, they are implemented as simple function calls instead of slower software interrupts. Moreover, tasks are run in processor supervisor mode which will give them full access to the hardware.

While the main technique used in RTAI is quite similar to the one used in RTLinux, the major architectural difference between these two projects is how they add the real-time features to Linux. Although both of these two use Linux kernel modules that can be loaded to the memory, some differences exist in how much change they make in the standard Linux. In developing RTLinux, the kernel source files are aimed, consequently many modifications have been made in the source file [5]. On the contrary, the interrupt abstraction layer in RTAI has a structure of pointers to the interrupt vector, which localizes the interrupt management. This abstraction layer is built by fewer than 20 lines of modifications and there are only about 50 lines are added to the original code. Also, by changing the pointers in the real-time abstraction layer back to what it was before implementing RTAI, the structure can be changed back to the original one when there is no real-time operation needed. This makes RTAI a more efficient and elegant approach with less intrusion in Linux codes. RTAI task scheduler work is based on preemptive fixed priority scheduling. Also, RTAI gives the permission to have immediate access to the PC hardware by removing the need to first going through the interrupt handling layers of the original Linux kernel [3]. Also by using a loadable module, RTAI executes a subset of POSIX 1003.1.c. As a matter of fact, the POSIX support in RTAI in quite similar to standard Linux except the fact that RTAI does not support signal handling and child parenting function in standard Linux (which are inappropriate for real-time purposes, because all threads are considered as parts of a single process).
2.3.4 Xenomai

Xenomai [6] is a real-time subsystem which can be integrated with Linux kernel in order to increase the predictability of the system. In its latest improved version, it employs a small side kernel that cooperates with the Linux kernel. A major difference that Xenomai makes is that it allows the real-time tasks to work on the user space. In fact, Xenomai makes use of the concept of introducing two domains for executing the tasks, a primary domain, which is controlled by the real-time section, and a secondary domain which is controlled by the Linux kernel. Real-time threads are handled in the primary domain unless the Linux standard API is called. It is worth mentioning that Xenomai has the ability to pull out the real-time processes from the normal tasks which are created by Linux standard POSIX API. This means that the real-time threads in Xenomai kernel inherit the ability of invoking the linux functions when they are not being run in real-time mode [6]. Now when the real-time task calls the Linux standard API, it is transported to the primary domain, where the Linux kernel schedules it using the SCHED_FIFO (first-in-first-out) or SCHED_RR (Round-robin) policies, with regards to the task’s fixed priority. After the function invocation is completed, the real-time task can go back to the primary domain. This way, Linux OS is used more efficiently. The other advantage of this applied isolation is that real-time tasks do not use the same memory space as the other tasks, therefore, the possibility of the occurrence of a crash due to existence of errors in a real-time task is reduced. A problem with Xenomai is the latency that it will still experience when dealing with real-time tasks in primary domain. This latency is longer than the latencies imposed by RTAI. Also, the secondary domain, which is in control of the Linux kernel scheduler, imposes another latency to the system. Thus, the developers in Xenomai project are working on enhancing these problems with the real-time performance of the system.

2.3.5 Reducing the Latencies

A set of other suggestions has been presented to make the Linux kernel itself more predictable. In order to achieve this goal, focus has been on improving the non-deterministic parts of the Linux kernel. The main sources of unpredictability in Linux kernel are latencies, the scheduling method, and the timing resolution of the Linux kernel. The approaches suggested to fix these problems have resulted in an operating system more responsive to the applications without having the need to change those applications. Consequently, now we can expect that pre-emptibility can be a standard specification of Linux kernel. However, according
Figure 2.2: A diagram of overall design structure of Xenomai

to the evaluations and tests a preemptive kernel can reduce the kernel latency for
a few milliseconds in some applications, depending on the application type. A
number of suggestions have been given to reduce the latencies without changing,
or adding to, the standard API of the Linux OS. Two major works in this field
were proven to be more efficient: the preemptive kernel patch, and the low latency
patch [7] [8]. This section discusses these two methods. The first method was first
suggested by Robert Love in Monta Vista’s preemptible kernel patch, which was
later used in the next versions of Linux, and the second one was proposed by Ingo
Molnar. Other solutions were suggested as well by researchers in order to improve
the timing resolution of the Linux kernel, and will be explained in this part. The
last solution which will be discussed in this section will be the PREEMPT-RT
patch, again suggested by Ingo Molnar.

2.3.6 Preemptive Kernel Patch

The main idea behind the Preemptive Kernel Patch [9] is to provide the scheduler
with the opportunity to execute more frequently. This happens by changing the
interrupt return codes and spinlock macros, thus the scheduler is called when it is
appropriate for the preemption to happen [10]. In fact, this approach overcomes
the limitation of a single execution flow inside the Linux kernel. Consequently,
there is no need to disable preemption when an execution enters the kernel. In
order to support kernel preemptability, the kernel code must be completely pro-
tected by using mutexes or spinlocks. Preemptive Kernel Patch disables kernel
preemption by holding a spinlock. Spinlocks are the basic synchronization tools
in the Linux kernel for symmetrical multiProcessor systems mode(SMP). To avoid
race condition, spinlocked regions are chosen as critical sections in the Linux SMP
kernel because basically every processor can use a system call in this mode. In
fact, what spinlock does is to make sure that at each time only one process can
enter the critical section [10]. This function makes the spinlocks similar to binary
semaphores or mutexes with the difference that unlike mutexes, the process that
wants to use the spinlock is not put in a waiting queue; it is put in a busy wait con-
dition called spinning around until the current process unlocks the spinlock [10].
The point here is that applying spinlocks in uniprocessors is not possible because
busy waits are not allowed in uniprocessors. On the Contrary, on single processor
OSs there is no need to use spinlocks because the processor executes only one
process flow at a time. Another major change that this approach makes in Linux
kernel is introducing a new counter semaphore called the preemption lock counter
(PLC) which lets the preemption of the Linux kernel happen. Preemption can
actually happen when the value of this counter is zero, and it is forbidden when
the value is any value bigger than zero. It is important to know that preemp-
tion cannot happen in 5 cases:(1) while handling interrupts (2) while processing
Soft-IRQs (3) while holding a spinlock, write, or read lock. (4) while the kernel
is executing the scheduler (5) while initializing a new process in the fork() system
call [10]. Thus, when the processor enters in any of the above conditions the PLC
is incremented, so no preemption is allowed. When the scheduler is leaving the
critical section, meaning that all the locks that a task uses are released, the PLC
is decremented this time, and a test is made to see if preemption has been asked
for during the time that spinlock was held, and if so, the preemption will happen
immediately. This process usually can happen tens of thousands of times in a
second. However, one problem that this approach can face is dealing with long
spinlocks, when a protected region is too long and thus preemption is forbidden
for a long time. Robert Love suggested a feasible solution for this problem by
introducing Monta Vista’s preemption patch. This will be discussed in the next
section.
2.3.7 Low latency Patch

Another solution for reducing scheduler latency, called the Low Latency Patch [10], was suggested by Ingo Molnar. This project is maintained by Andrew Morton now. In this method, focus is directed on introducing preemption points (also called rescheduling points) in blocks of code that may execute for longer periods of time. The basic idea is to find blocks of code that iterate over large data structures and find out how to assign a call to the scheduler safely if the loop has gone over a certain threshold and a rescheduling is needed. This sometimes implies a lock breaking to happen, which means a spinlock has to be canceled, and activated again, after the rescheduling process is finished. A preemption point can be placed only where no data inconsistencies are possible due to read/write or write/write conflicts [10]. Most of these preemption points are placed where long lists have to be changed which usually happens in memory management or in drivers. One tool for finding the places where a preemption is necessary is Andrew Morton’s rtc_debug_patch that modifies the real-time clock driver in order to look for the latencies bigger than a specified threshold. Then the programmer should evaluate the reason why the latency is created by the help of looking at the routines that happen to be the reasons for the long latency sections. Also, the programmer can find the preemption places by applying the patches and comparing the source files before and after applying them. Although Low Latency Patches are based on a simple idea, the implementation is not very easy. Finding and modifying the code sections that cause latencies despite the development of useful device such as Andrew Morton’s suggested tool is a time consuming debugging task. This can be even more difficult considering the dynamic nature of the Linux kernel state. Nevertheless, low latency patches have shown to reduce the maximum Linux kernel latency to a much lower value than what the preemptive kernel patch offers [10]. This method is implemented and used in some real-time versions of Linux such as REDLinux.

2.3.8 Monta Vista

As mentioned before, long spinlocks caused by long sections that need to be protected can turn in to a major problem in Preemptive Kernel Patch approach. A solution for this problem is to combine the preemption patch with parts of the Andrew Morton’s Low Latency Patch [10]. This solution, suggested by Ingo Molnar, was used in Monta Vista’s patch which was later accepted and used in Linux 2.4.2 version. The idea is that the preemption points of the Low Latency Patch are used to solve the problem of long spinlocks seen in the Kernel Preemption
Patch, by adding preemption points to the long blocks of code that need protection. As a matter of fact, when the Kernel Preemption Patch makes the Linux kernel preemptible in general, the use of Low Latency Patch makes the spinlock areas preemptible. Monta Vista also provides measurement tools so that developers can measure the preemptibility performance with their actual applications and environment. Monta Vista has proven to reduce the average latency in Linux kernel. However, it does not guarantee to reduce the maximum latency [10].

2.3.9 TimeSys

The idea behind the design of TimeSys [11] is the same as Monta Vista, meaning that spinlock calls are used to manage the preemptions. Though, the main difference between TimeSys and Monta Vista is that in TimeSys, mutexes are used instead of interrupt counters [11]. Basically, a mutex is a binary semaphore which is used in a system to ensure the mutual access to the same resource. When in the interrupt counter approach preemption is forbidden in the spin locked sections, in mutex approach preemption is available for a higher priority process when the lower priority process is using different resource. This means that it is possible to give the real-time tasks a higher priority than some interrupts. The mutex also uses priority inheritance protocol (PIP) to solve the priority inversion problem. In practice, implementing this protocol is difficult, but it gives the kernel the possibility to further reduce the kernel latency even to the level of the latency caused by the interrupt abstraction method. Beside the extension made in the scheduler, the interrupt systems are also changed in TimeSys. In fact, the interrupt service routines (ISR), and IRQs are transformed to kernel threads, meaning that they will be scheduled under the real-time preemptive scheduler too, which is how the real-time threads are able to have a higher priority than the interrupts. Moreover, since all the interrupt handlers are treated as scheduler threads, it is possible to put an interrupt handler to sleep. This way it is possible to substitute the spinlock interrupts with mutexes. TimeSys approach enhances the hard real-time features in Linux, but still it is not possible to preempt the Kernel process at any arbitrary point safely.

2.3.10 Preempt_rt Patch

The Preempt_rt patch [2] has attracted developers attention with its neat and efficient design, and parts of its design were employed by Linux. This patch also uses spinlocks to protect the critical sections from preemption in general, and employs
mutexes to allow preemption in some parts of the protected area in order to remove the problem of long locks. Priority inheritance protocol is also used to handle the spinlocks and semaphores. The other feature of this patch, similar to the successful features of the above-mentioned approaches, is that it treats the interrupt handlers as a user space thread which is managed by the kernel scheduler. The Preempt_rt patch converts the standard Linux timers to separate infrastructures to obtain a higher resolution timer, and a separate one for the time outs, which helps obtaining a high resolution user space POSIX timer [12]. A comparison between the performances of the Preempt_rt patch, the standard Linux, and a system that employs both RTAI and Xenomai approaches has been made in [2]. According to the test, the maximum interrupt latency in the preempt_rt patch is quite similar to the system using both Xenomai and RTAI patches, but this latency is higher in the standard Linux kernel. These results indicate the improvement made by the Preempt_rt patch.

2.3.11 LITMUSRT

Linux testbed for multiprocessor scheduling in real-time systems (LITMUSRT) [13] is a project based on modifications applied to the Linux 2.6.20 kernel configured for a symmetric multiprocessor (SMP) architecture. In this project a scheduler plug-in interface is created that allows new scheduling algorithms to be added [13]. Another main subsection of the LITMUSRT is the user-space libraries for supporting the creation and execution of real-time tasks, as only real-time tasks are scheduled by LITMUSRT. LITMUSRT operates in two modes: real-time, and non-real-time. The system is basically in non-real-time mode at the starting to set up the complete tasks before the scheduling happens. The core LITMUSRT timer tick function (rt_scheduler_tick()) is responsible for mode switching [13]. For releasing and queuing of real-time tasks, LITMUSRT provides the abstraction of a real-time domain implemented by structure rt-domain-t. This abstraction domain consists of a ready queue and a release queue, which is ordered by ascending release time of tasks. An EDF order function is used to organize the ready queue. Though, a feedback-control EDF (FC-EDF) algorithm was used for order functions in LITMUSR later. Wrapper functions are created for list operations such as queuing and excluding tasks from the queue, and functions such as (list_insert()) (used for putting a task into the queue) and (list_qsort()) (used for sorting the queue) are used to extend the Linux list.h API. Real-time tasks have the highest static priority upon creation. However, Instead of being kept in the standard Linux run queues, the real-time tasks are managed by scheduler plug-in. Each of
these plug-ins is responsible for its own run queue. When schedule() is invoked, the current scheduler plug-in takes the control, and if it chooses a real-time task to be scheduled on the local processor, then the task is placed into the run queue, thus the Linux scheduler is bypassed. There are thirteen scheduling plug-in decision types used in LITMUSRT, each time one of them used. As a matter of fact, each scheduling algorithm is created as a plug-in. During the system booting, the kernel command-line parameter rtsched determines which of the scheduling decisions should be used. A brief description of these steps [13]:

- When a real-time task is going to be added, a call to prepare task() calls the scheduler plug-in which examines the task. If the new task does not match the plug-in requirements, then it is not accepted.

- When the task-blocks() function is called, the schedule plug-ins are notified of task blockings, and thus, can move the blocking task.

- If the new task is a real-time task, the function try-to-wake-up() invokes wake-up-task()

- When a real-time task exits, a call to tear-down is made to notify the scheduler plug-ins, so the resources belonging to that task are freed.

- A call to mode-change() notifies the scheduler plug-ins about a switch to, or from the real-time mode. After switching to the real-time mode, the real-time tasks are put into TASK-RUNNING state in the release queue.

- The scheduler plug-ins are notified of each tick of the timer when a call to scheduler-tick() happens.

- As mentioned before, schedule() is the main scheduling function. After this function is called, a scheduler plug-in should select which real-time task to execute. Only if the scheduler plug-in selects no task, the Linux scheduler takes the responsibility of the scheduling.

- In order to avoid race condition, LITMUSRT re-inserts a preempted real-time task into the ready queue by calling finish-switch() function after the context switch is finished.

- By calling the function sleep-next-period(), the scheduler plug-in is notified about an early completion of a task so that the task will be put back into the release queue.
Chapter 2. Background

- To configure scheduler plug-ins through a unified interface, the function scheduler-setup() is created. This is because some schedulers have different parameters for determining the method of scheduling.

- To examine the condition if a newly unblocked task needs to inherit the priority of the task behind it in FIFO for the FMLP (flexible multiprocessor locking protocol) protocol, the function inherit-priority() is called.

- In case a task has to give up an inherited priority in FMLP The function return-priority() is called.

- If the priority of the locking task has to be raised according to FMLP rules, the function pi block() is called.

Besides the libraries that LITMUSRT uses, a number of system calls are also added to Linux. These system calls are used in managing real-time tasks, asking for an update about the status of the system, notifying the completion of jobs, setting up the schedulers, and to provide support for FMLP. Scheduling decisions are made in timer interrupts. This implies a unified timing system for all of the processors. In order to align the timings, first the value of the Time Stamp Counter (TSC), a 64 bit counter responsible for recording the current time, is recorded by the processors upon booting the system. Based upon this measurement all the processors calculate how much delay they need to align themselves with respect to processor number 0. After that, each processor disables and resets its local interrupt controller timer, and at this time it generates its next interrupt after a full timer period. By resetting its interrupt controller timer, each processor aligns its time to the others. Moreover, task periods are integral with respect to the length of the systems smallest scheduling time span (quantum) [13].

2.3.12 Enhancing the Timing Resolution

The timer mechanism in Linux works based on a timer notion called jiffy (An unknown short period of time). All of the timeouts of the timers are changed in to jiffies, and for managing them, a Linux system timer is used. This system timer is designed by cascading five timer vectors (TV). When scheduling a timer, it is placed in one of the five vectors based on the time the system has until the timer expiration time. The vectors can be considered as arrays of jiffies that work by being moved to a time in the future. All needed to do when inserting a new timer is to place it in a timer regarding to its timeout value in jiffies. Now when a timer interrupt is called, it means that the jiffy’s value should be incremented, which
means all the timer values are changed [2]. The cascade design is built in order to make this change possible in all the timer values. A benefit of this method is that it makes timer insertion and deletion faster, because all the timers are shifted at once. On the contrary, since the cascade function disables preemption and interrupts when running, and each time millions of timers are executing at once, very big latency problems emerge. Another problem is that many of the timer functions include big blocks of code. In order to fix these latency problems and obtain real-time features in Linux, a number of solutions have been presented. Two of the solutions are the high resolution timer (HRT), by George Anzinger, and KTimers, suggested by Thomas Gleixner and Ingo Molnar [2]. In next sections these two approaches are reviewed.

![Diagram of how timers are managed inside Linux kernel.](image)

**Figure 2.3:** A diagram of how timers are managed inside Linux kernel.

- **HRT**

  The idea behind HRT is to employ the Linux system timers to manage all the timers in the timer vectors. This timer system is able to use its own system interrupt, which can be used to trigger each of the timers that are put in timer queues by HRT’s Linux system timer [14]. Unlike Linux standard interrupt system, this system timer does not generate an interrupt after a constant interval of time. In fact, interrupt creation depends on the next event scheduled by the OS. As a matter of fact, HRT is built on top of the Linux System Timers and introduces a separate interrupt to obtain a resolution in microsecond scale [2].
• KTimers

The KTimers approach uses a red-black tree to store all the timers, so there is no need for the cascade operation although it is slower than the standard timeout mechanism of Linux [2]. It also keeps the expiration dates by using function called Ktime t, and uses nanosecond time out specifications. When KTimers approach uses its own interrupt to trigger the timers, similar to the HRT approach, it does not interact with the Linux system timers, and its time preservation mechanism is completely independent of the jiffies values. As a matter of fact, the Ktimers patch does not make any change in the current API in standard Linux, it adds its own entirely new API, which is developed to have a higher resolution. The Ktimers API was accepted and implemented in the 2.6.16 version of the mainline Linux OS. The biggest objection to the Ktimers patch is its use of nanosecond values, which brings up the need for having 64-bit variables, which slows down the 32-bit systems performance. Nevertheless, the additional overhead this patch introduces is not a big problem to get worried about, specially when comparing it to its high resolution [2].

2.4 Monitoring

Despite the significant improvements that have been made in the timing behavior of different systems in real-time environment, proposing a correct-by-construction system is very difficult in real-time environment. This is mainly due to the complexity and unpredictability of real-time environments. Thus, run-time monitoring can be a helpful approach in order to provide the user with data regarding to the actual timing behavior of the system. The data that these monitors provide can be used for further analysis by the user, or the monitor itself. As a matter of fact, the intention of run-time monitoring is to determine whether the current execution meets the specified technical requirements [15]. Timing behavior monitors provide the user with necessary information which can be used to detect or predict violations of timing constraints. Examples of such information are deadline misses and context switches [1]. Note that in this report, in some parts, the term real-time behavior is used, which refers to the system behavior in real-time environment, meaning that when the system is dealing with real-time tasks. Since some terms are used in this report that are especially common in works related to monitoring of timing behavior, it feels necessary to the author to provide the definition of these terms.
Target system: The system in which monitoring, run-time checking, or run-time verification is performed is referred as the ‘target system’.

Latency and interference: detecting events of interest and processing them can be performed in different ways, each of them causing a different amount of interference with the target system. This is one of the main issues in monitoring of timing behavior, because it can alter the timing behavior of the target system significantly.

Probes: Probes are basically elements of a monitoring system which are attached to the target system in order to collect information about its internal operation [1].

Probe effect: As mentioned, including the monitoring code in the target software has the disadvantage of changing its behavior because of the amount of latency being added to it. This is because a part of the CPU time should be dedicated to the monitoring code. This latency is referred as probe effect.

Tracing and Sampling: Data collection can happen in two ways: tracing and sampling. In tracing, every occurrence of an event creates a record. So event tracing is characterized by the completeness of knowledge [16]. Sampling yields only a statistical measure of the software’s execution patterns. Tracing is mentioned as event-driven monitoring whereas sampling is called time-driven monitoring.

2.4.1 Linux Native Monitoring Tools

In this section an overview of monitoring tools that are available for many version releases of Linux OS is provided. Since a quite big number of such native tools is available, the most famous and common monitors are selected for this section. It is worth mentioning that the selected tools are the ones that are more related to our project.

2.4.1.1 Sysstat Project

The Sysstat project is managed by Sebastien Godard [17]. The monitor in this project derives the raw data about the processes from the Linux /proc file system and makes them ready for display and for building a data base from the processes execution history. The Sysstat stores the information for a period of time after collecting it, and calculates the mean values and makes it possible for the user to have access to the data of interest at predefined time intervals. This project, which is used in many commercial versions of Unix systems, contains different monitoring tools such as sar, sadf, iostat, nfsiostat, cifsiostat, mpstat, and pidstat commands for Linux [17]. The Sysstat package provides system information such as [17]: process names, process IDs, CPU usage and load, total, or per processor, physical and virtual memory, swap file usage, memory load, and paging network
interfaces, NFS servers and clients, number of Interrupts including global and local interrupts, context switches and run queues. While users might be able to obtain some of the mentioned information through other Linux monitoring tools, Sysstat tools facilities have fixed some of the issues the users had with those other tools. For example, the challenge of finding bugs and loops in active programs, which demanded the users to spend long time using tools such as top, is removed by Pidstat, which provides statistics reports for each single Linux tasks. Also, Sysstat tools make it easier for the administrators to use the data from proc file system without the need to have the necessary skills to extract it manually. Moreover, there are some aspects of the monitored system that cannot be easily explored by using tools that give a snapshot of the system execution status. In next subsections an overview of the most important Sysstat project tools is presented.

2.4.1.2 Mpstat

The Mpstat command provides data about the activities of each processor as well as the average global activities, starting with processor number 0 [17]. Although Mpstat command can be used for both SMP and UP systems, in UP systems it only displays data about average global activities. The time span between each report is selected through an interval parameter. The value 0 means that the displayed data includes processor activities since system startup. The count parameter determines how many reports should be submitted during each second. If this parameter is not specified the Mpstat generates reports constantly. All the data Mpstat presents comes from the Proc file system, thus, it needs to be mounted on the system for the Mpstat to work. For example, CPU utilization and interrupts summary data comes from /Proc/stat, the number of interrupts per seconds comes from /Proc/interrupts, and the software interrupts data can be obtained from /proc/softirqs. In order to have the information about the interrupts for instance, after choosing the I option, If the CPU keyword is used, the number of interrupts received by the processor, or the processors, is displayed. In the same way, if the SCPU keyword is used, the number of each software interrupt received by each processor, or all the processors is displayed. If the SUM keyword is used, the total number of interrupts received per second for all processors is displayed, And, by using the keyword ALL, a combination of all the previous information is reported [17]. The Mpstat report that the user sees consists of the following sections: Shows the number of the processor whose data is being displayed. Shows the CPU utilization percentage. Shows the percentage of CPU utilization while execution is happening at the application level with nice
Chapter 2. Background

priority. Niceness, or nice value, is the priority that the user sets for the tasks to be executed in Linux. The niceness scale goes from -20 to 19 with 19 being the lowest priority for a task. Shows the percentage of CPU utilization while execution is happening at the system level (kernel) without the time spent for the interrupts. Shows the percentage of time that the CPU or CPUs were idle due to an I/O request from the system. Shows the time percentage spent by the CPU or CPUs to perform the interrupt service. Shows the percentage of time spent by the CPU or CPUs to perform the softirq software interrupt service. Show the percentage of time that the CPU or CPUs were idle. The difference between intr/s Shows the number of interrupts received by the CPU or CPUs per second [17].

2.4.1.3 Pid Stat

Pid Stat tool is used for monitoring single tasks being executed by Linux kernel [17]. -p and u options: By using the p command, a list of process ids will be passed.

- d option: The d option provides the I/O statistics of the monitored process. The kB rd/s column in the d option corresponding window gives the amount of data read by the process in kilobytes per second, and kB wr/s column gives the written data by the process in the same rate. Also, the kB ccwr/s gives the kilobytes/sec data writing which has been canceled by the monitored process in case of data rewriting occurrence.

- s option: The -s option displays the stack memory usage. The StkSize column shows the size of stack allocated for the process. The StkRef column shows the size of stack memory used by the process. Note that the stack reserved for the process might not be used completely, so there is a difference between stksize and stkref.

- w option: The -w option reports task switching. The cswch/s column shows the number of voluntary context switches per second, i.e. the context switches that happen due to unavailable resources needed for a thread to be completed. The nvcswch/s column reports the number of non-voluntary context switches per second. Non-voluntary context switches occur when the process is stopped by the kernel due to completion of the CPU time dedicated to it.

- r command: The r command reports page fault and memory utilization. There are two types of page faults: the minor page fault and the major page fault. In minor page fault a page of interest is loaded concerned page
is loaded in the memory but it is not confirmed as loaded. The number of this type of page fault per second, is displayed under the mint/s column. In major page fault however, a memory page needs to be loaded from the hard disk into the page cache. A significant number of major page faults can be harmful to the system, thus the system memory should be increased. The VSZ column displays the size of virtual memory for the monitored process, and RSS displays the amount of physical memory used by it.

2.4.1.4 Top

Top is another monitoring tool that provides real-time report of the process activities with an interface for the user to filter and manipulate the monitored data by using a command or by specifying the feature in the system configuration file [18]. Four data output areas are presented to the user: the summary area, the message/prompt Line, the columns header, and the task area. In the following parts the four areas are explained [18]:

- The summary area: the summary area, which is displayed in the first five lines, gives a summary of the system activities. The first line includes the command name, current system time, system uptime, number of users and the average load of the CPU for the last one minute, last five minutes and last fifteen minutes [18]. The second line gives the total number of processes in the system as well as their number based on their status. The next line gives CPU utilization between the current time and the time of the last refresh. The next two lines provide information about the memory usage. This information includes the total physical memory, the used and free memory chunks and the part of memory used for blocking I/O buffers, all in kilobytes. Also, the next line gives data about cached, which together with buffers serve the dynamic memory requests from the processes. Finally, in the last line the swap memory (the memory swapped to a disk in order to extend the system physical memory) is shown.

- The message area: the single line after the summary area is the message line. The user can use this area to use prompt commands to perform the sub-commands of top. In this area the error messages are also displayed.

- The headline area: as it is clear, the column header area includes the headings for different columns.

- The task area: in this area, the details of all the processes in the system are displayed. The information given in this section includes: the IDs of the
processes (PID), the owner user name (USER), priority of the process (PR),
nice value (NI), total used amount of virtual memory (VIRT), resident size (the
used non-swapped physical memory) (RES), shared memory (SHR), and
the process status (S), which can be running (R), sleeping (S), uninterruptible
sleep (D), traced or stopped (T) and zombie, which means that the process
has finished the execution and has released all its resources but still has an
entry in the process table (Z). The other information include CPU time usage
percentage since the last refresh.

Although top command provides the user with very well-structured information
display and administration as well as a large number of options, it imposes a
significant overhead on the system due to a few thousands system calls that it
makes to update its output and the fact that it is highly configurable [18].

![Figure 2.4: An example of Top command output](image)


2.4.1.5 Vmstat

The main use of the Vmstat command is to report the virtual memory statistics. In addition, by using Vmstat the user is able to have access to data about system processes, physical memory, swap, I/O blocks, interrupts, context switches, and the CPU activity [19]. The most common Vmstat command is vmstat[delay[count]], where the Vmstat sends out data to its output as many times as determined by the count parameter, and the delay parameter determines the time span between two Vmstat reports in seconds. If the count parameter is not specified, the Vmstat will display the output forever. If the delay parameter is specified, Vmstat gives the average value of the data since the last reboot and exits. The files that Vmstat uses to extract data are /proc/meminfo, /proc/stat, and /proc/*/stat. Here is a description of how the Vmstat output looks like with the explanation of the parameters [19]:

- **Procs r:** the number of processes waiting for run time. **b:** The number of processes in uninterruptible sleep state.
- **Memory swpd:** used virtual memory.
- **free:** idle memory.
- **buffer:** the amount of memory used as buffers
- **cache:** the amount of memory used as cache
- **inact:** the amount of inactive memory
- **active:** the amount of active memory
- **Swap si:** Amount of memory swapped in from disk
- **so:** amount of memory swapped to disk IO
- **bi:** blocks received from a blocking tool
- **bo:** blocks sent to a blocking tool System
- **in:** the number of interrupts occurrence per second
- **cs:** the number of context switches per second CPU These are percentages of total CPU time.
- **us:** time spent running non-kernel code.
- **sy:** time spent running kernel code.
• id: time spent idle. (Linux 2.5.41 and later versions)
• wa: time spent waiting for IO. (Linux 2.5.41 and later versions)
• st: time stolen from a virtual machine. (Linux 2.6.11 and later versions)

Some of the most common commands that the user can use to filter the output data are:

• a: this option gives the active (has been used recently) and inactive (has not been used lately) memory columns.

• f: this option gives the number of forks (new processes created) since the system boot including the fork, vfork, and clone system calls, A system call in the Linux kernel that creates a child process, (a process that is created by the parent process) that may share parts of its execution content with the parent process.

• S: this option is used to change the units of the memory data. By default, the memory data is provided in the units of Kilobytes (1024 bytes). By using the S option it is possible to change this to 1000, 1024, 1000000 or 1048576 bytes using k, K, m or M parameter respectively.

2.4.1.6 Gnome Desktop Environment Tool

The GNU network object model environment (GNOME) tool is relatively the equivalent of the task manager existing in Windows systems. This international Red Hat project, which is built of open source software, consists of a desktop environment and a graphical interface for the user that runs on an operating system. By using GNOME the users has access to information about general system status, the number of processes, memory and swap usage, and process-level statistics as well as a graphical representation of the mentioned information. Also, system processes can be manipulated through GNOME; for example, the user can kill a process.

2.4.1.7 KDE System Guard

The KDE System Guard also provides statistical data about the system status, load, and the processes as well as the ability to manipulate the processes. The KDE System Guard window has two tabs: the process table, and the system load. The process table lists all the processes with details about the memory, CPU, and
network. The user can sort and filter the list. Here are the items that the process table covers [20]:

- **Name**: the name of the executable file that has started the process.
- **Username**: the user that the process belongs to
- **CPU**: the total amount of CPU that the process uses currently, divided by the number of processors existing in the machine (cores)
- **Memory**: the amount of physical memory being used by the process minus the swapped memory and the space occupied by shared libraries.
- **Shared Mem**: the amount of physical memory that the process shared libraries are occupying. This memory is shared among all processes using the shared libraries [20].

Additional columns in the process table:

- **PID**: the ID of the processes
- TTY: the controlling terminal on which the process is running
- Niceness: the priority of the process running in Linux (-19 to 19).
- CPU Time: the total system time that the process has been executing for, displayed as minutes:seconds
- IO Read: the number of bytes read.
- IO Write: The number of bytes written.
- Virtual Size: the amount of virtual memory being used by the process, including shared libraries, graphics memory, files on disk, etc. Using the context menu to select the Display Units can make the display more useful.
- Command: the command that was used to launch the process.
- System load: this tab gives a graphical representation of the history of the memory, CPU, and network usage.

![Figure 2.6: An example of KDE System Guard output](image)
2.4.1.8 Proc File System

The proc file system is a well-structured interface which exists in the official kernel releases in many versions of Linux, and is a widely used system monitoring tool. This virtual file system allows the user to both monitor and manipulate the kernel data structure without the need of adding any memory [21]. One of the important features that Proc file system benefits from is its simple API. Also, unlike many of the common Linux native monitoring tools the Proc file system provides direct access to the kernel. Compatibility with POSIX and high security are the other important features of this virtual file system. Furthermore, imposing low overhead on the system along with run time system monitoring are the features that make Proc file system fascinating for embedded systems. For instance, while Top command takes up to more than a thousand system calls to generate the output for a simple page, the proc file system performs this process by only about 40 system calls, thus creating much less overhead and less processor demand. However, there are two main disadvantages of using Proc file system: the size of the code that supports Proc in Linux is large, and it is not portable, even in some versions of Linux OS. The Proc file system can be used in viewing system, memory and hardware statistical information and modifying different run time parameters, and thus kernel behavior. The /proc/sys allows the user to use the echo command to write values in the system files and thus modify the kernel. Also, many monitoring tools are based on Proc file system, such as top and Sysstat project tools. There is no need to parse complicated input data formats to create the output of the Proc file system user-friendly. This is because of the freedom this file system has in representing the output due to the use of non-volatile memory. However, for using text mode interface the input should be parsed. The Proc file system uses two interfaces: binary interface, and text mode interface. Although the binary interface is simpler because it does not need parsing and scanning the input file formats, its output is not meaningful for the direct use of human. Some useful subdirectories in proc/sys [21]:

- proc/sys/fs: this directory includes specific file system, dentry state (status of the directory cache for dynamic allocation of directory entries), and quota information.
- proc/sys/fs/binfmt: misc supports registering additional binary formats
- proc/sys/kernel: this subdirectory includes general kernel parameters and contains data about general kernel behavior.
Chapter 2. Background

- proc/sys/vm: the files in this directory can be used to modify the performance of the virtual memory (VM)

- proc/sys/dev: this location contains device exclusive parameters

- proc/sys/sunrpc: there are four files in this directory that are responsible for debugging operations of Remote procedure calls. The four files are NFS, NFS-daemon, RPC and NLM.

- proc/sys/net: the interface to the networking operations is located in this address.

- proc/devices: tt contains devices available for the kernel.

- proc/modules: kernel modules are loaded in this location.

- proc/cmdline: kernel command line is written here.

- proc/meminfo: details about memory usage are included in this directory.

- proc/config.gz: configurations of the kernel in a compressed file format can be found here.

Important commands: By using proc info command the user is able to have access to important system information. Using parameter a, the user is able to have all the information in the same list. By using parameter nN the information is updated every N seconds. Parameter d generates differential values, and parameter -dn5 presents the values that have changed in the last five seconds. Finally, pressing q quits the program.

2.4.1.9 Summary

A number of monitoring tools, mostly simple, are available in different version releases of Linux OS. These tools cover a wide range of data such as memory usage, number and IDs of the running tasks, network status, and CPU usage. However, none of those native tools provide the user with the monitoring data that are crucial for real-time analysis, specially in real-time environment. While the covered data areas by Linux native monitoring tools are valuable information for ordinary use, the timing attributes such as the execution time of tasks, their deadlines, or their periods cannot be obtained by using such tools. Moreover, these tools can not be of much help for timing behavior analysis in case of manipulated Linux scheduler. In next section, monitoring tools that are developed for controlling the real-time aspects of different systems are reviewed. However, limited work has been done
on developing a timing behavior monitoring tool for Linux OS. Consequently, the main purpose of giving a review of a selected number of such monitors is to give the reader an idea of different approaches toward developing monitoring support for different software systems. Hence, in the next section, the logic and architecture of a selected number of timing monitors is presented as well as a brief summary of their advantages and disadvantages.

2.4.2 An Overview of Related Work on Monitoring of Real-Time Performance

With ever increasing use of real-time systems, the significance of reliability of the devices using such systems seems obvious. In order to make sure that the real-time quality of the system does not fail, many techniques for run time monitoring and debugging of these systems have been developed. In general, monitoring supports the debugging, testing, and performance evaluation of the programs, and covers different aspects of systems behavior, such as memory usage, CPU usage, network and connections status, and tracing of the execution of the processes in the system. Monitoring of real-time systems focuses on the timing behavior of the processes in the system. The main goal in the monitoring in this area is to make sure that the real-time quality of the system is guaranteed, which basically means that all the tasks meet their assigned deadlines. For some of the previously

![Figure 2.7: An example of Proc File System output](image-url)
suggested monitoring tools a separate hardware support has been proposed. This special purpose hardware works in parallel with the main processor, also known as the target processor, in order to collect run-time data, and to post-process (e.g. Worst Case Execution Analysis) it. The main advantage of this method is that it allows a non-intrusive run-time monitoring of the system because the monitored data is saved and analyses in a device separated from the main processor. Though, they generally give only limited information about the target system and are limited to a few fixed observation points. Moreover, these monitors often use complicated features of the hardware in order to get the necessary information. In addition, monitors using an extra hardware cannot support the dynamic creation and elimination of processes, the migration of program parts in memory, and the use of memory management units, because the information needed is derived by using fixed addresses. Also, the installation of such monitoring tools requires deep knowledge of the system. These disadvantages makes hardware monitors inappropriate for monitoring application programs of for example modem computer systems. In another major type of monitoring tools, the monitor part is placed inside the system. This is usually done by inserting event-data-collecting code into application programs, OS kernels or monitoring tools. Such monitors usually use the same execution environment as the system to be monitored. Consequently, software monitors cause some interference with the monitored system. This will in turn increase the overhead on the system. Hybrid tools use the features of the two methods mentioned previously. Though such tools suffer from some of the disadvantages of software and hard-ware monitors, especially the hardware monitors, they benefits from the advantages of both software and hardware methods. In this approach a hardware device is used to monitor and collect data, but the difference with the hardware monitor is that the data is collected on behalf of a software tool that is run inside the system being monitored. In next sections an overview of some important approaches suggested in each areas of hardware, software, and hybrid monitors is provided, three of each.

2.4.3 Hardware Monitors

2.4.3.1 Noninterference Method

The monitor architecture consists of two main parts: the interface module, and the development module [22]. The interface module’s major duty is to latch the internal states of the target system based on predefined conditions set by the user. The responsibility of the development module, which contains a general purpose microprocessor, is to start the monitoring process, to record the target
node execution history, and to perform analysis on the recorded data. After being connected to a node of the target system and initialized, the interface module keeps collecting events of interest until it finds a stop condition, pre-specified by the user. Then an interrupt is sent to the monitoring processor to separate it from the target processor for the data recording process to take place. The recorded information is transferred to a secondary storage for further processing. The events of interest include process-level events. During the monitoring, the time at which each event happens is recorded. Using this timing information, the execution history can be examined against timing constraint requirements. If violations are found, the replay mechanism can be used to test the program behavior again in order to isolate the errors [1].

2.4.3.2 PASM

PASM is a programmable hardware monitor, which provides a flexible set of tools for the user to specify events for a wide variety of monitoring applications [23]. The user can include a monitoring section with the application that defines events of interest, actions to be executed upon detection of those events, and the binding of events to actions. This section is then used by the compiler to automatically implement the instrumentation. Events in this monitor are associated with changes of state of the active process. An action can be recording the time of the occurrence of an event to track the timing behavior, or printing of the contents of some internal data structures when a certain point in the execution is reached [1].

2.4.3.3 ART

This monitoring system was developed for ARTS, a distributed operating system developed in 1980 [24] [25] [26]. This approach focuses on visualizing the timing behavior of the system processes. Rate monotonic and deferrable server algorithms are supported by this monitor, and the monitoring task is performed as a part of the target system. The functional structure of the monitoring system can be divided into three major parts: a part of the target operational system code that records the information of interest about the processes, called event trap, the reporter, which sends the information to the visualizer, and the visualizer, that uses the resources sent from the target system to create historical diagrams of the scheduling decisions of the target system. An event is generated each time the state of a process is changed. The ARTS monitor records process-level events such as process-creating, waking-up, blocking, scheduling, freezing, killing with completion, killing with missed deadline, and killing with frame overrun. For
monitoring timing constraints, the monitor uses the facilities that the ARTS kernel provides, such as 'Time fence'. The 'time fence' is a mechanism in the ARTS Kernel used to detect a timing error at run-time. Before each operation invocation the time fence is checked to verify that the slack time is bigger than the worst case execution time of the invoked operation, and a timer is set. If the execution is not completed within the worst case time, the timer announces an anomaly [1].

2.4.4 Software Monitors

2.4.4.1 PMMS

Program Monitoring and Measuring System (PMMS) is a monitoring approach that automatically collects high level information about the execution characteristics of a program [27]. Data collection is done by code inserted into the source program of the target system, and conditions are used to filter out events that are not relevant. The monitor handles events of interest by installing code that reacts whenever they occur. This data collection code includes Pre-condition (relevance test), Local-variables (used to store local data), Beforecode (code to collect data available before the event), Aftercode (code to collect data after the event), Post-condition( a relevance test based on data that is available after the event), and Action( code that stores data more permanently for later use). Examples of recorded event data include the time at which the event occurs, the value of program variables at that time, etc. The user can specify all the objects and relations using the high level specification language that is provided in this method. The PMMS uses a main memory for the active database to facilitate the collection, computation, and access to the computation results [27] [1].

2.4.4.2 Halsall-Hui

This monitor is designed to gather the data from each processing node of a real-time embedded system which is based on a distributed architecture [28]. The recorded information from each node includes the IDs of the tasks and processes, the type of the tasks and the system calls, and the time that those events happen [28]. The event data recording can take place in two ways. In the first method, the user inserts a library function call, with the corresponding variable name as a parameter, at the appropriate point in the source code, so that whenever that code section is being executed the recording function is called and run. Each of these functions can record a specified event. The event data is then sent to the single monitor of that node, which is also a library function. In the second method, an
interrupt is used to periodically refer to a data table provided by the user, which includes event identities, the recording frequency, and the variables to be recorded. This event information is saved in a system file, or an application [1].

### 2.4.4.3 OSE Monitor

The main idea behind this approach is to add a second layer scheduler to the OSE (Operating System Embedded) real-time operating system to make it easier to query the execution result of real-time tasks [29]. This adjunct scheduler uses the specifications of real-time tasks, such as the period and execution time of each task, from a parameter file. According to these parameters the second layer scheduler schedules the tasks by allowing them to be sent to the core scheduler in Earliest Deadline First (EDF) or Rate Monotonic Scheduling (RMS) scheduling algorithms. Thus, it is clear that the second layer scheduler process must have the highest priority among all the OSE processes. The monitor process works with the lowest priority, i.e., as a background OSE process, in order to make sure that it does not interfere with the scheduling process. Upon completion of a task, the monitor receives a signal from the second layer scheduler. Two types of log files are created in this process: a scheduling log file, and a monitoring log file. The scheduling log file, which is created by the second layer scheduler, contains the time points at which a task in the task set is scheduled, completed, or preempted. Monitoring log file, which is created by the monitor, is updated only when an instance of a task is completed [1].

### 2.4.5 Hybrid Monitors

#### 2.4.5.1 Hybrid Monitor

This monitor is designed by combining hardware monitors and software monitors [30]. A Test and Measurement Processor (TMP) is integrated to each node of the distributed system in order to record their process and intercommunication activities. The main principle of this method is that the target system generates events of interest, and the TMP hardware processes and time stamps them. The collected event data is stored in a FIFO memory in the CPU. Every time an event is sent into the FIFO buffer the CPU of the TMP is notified by an interrupt activating the processing of the events data. The number of messages, the message length, failed messages, the system time (the time spent for the processes in kernel), and the application time (the time that application processes spend in kernel
minus the system time) can be measured using this monitor as well. Events are
time stamped locally in this method [1].

2.4.5.2 ZM4

In this approach, a hardware system called ZM4, and an event trace processing
software called SIMPLE, which works independently from the monitor, are de-
veloped [31]. The connection between the hardware and the monitored system
is local area network type. Hosting the monitoring system, storing the measured
data, and presenting an analysis interface for the users are the responsibilities of
the control and evaluation center (CEC) of the ZM4 system. Also, a number of
monitor agents are built as slaves for the CEC. Each monitor agent is connected
to a target system node. Another responsibility of these probes is time stamping
and recording of the events. Time stamping is done using a global clock with the
precision of 100 ns. SIMPLE, which works on Linux and MS-DOS, is an evaluation
environment used for analyzing the recorded event traces. It generates a global
view of the distributed system’s behavior and performs trace validation and anal-
ysis as well. Whenever the monitor recognizes an event, it stores an event record
which consists of event token and a time stamp. The sequence of events is stored
as an event trace [1].

2.4.5.3 SoC-based Monitor

The System on Chip-based monitor uses a hybrid method for run-time verification
of embedded systems [32]. The monitor consists of event recognizer, a verification
tool, and the monitor output. The event recognizer decides if the collected data
is relevant to the event definition. After passing this step, the event data is sent
to verification section where it is compared with the requirement constraints. In
case a violation is observed, it is sent to the output of the monitor. The events
detection code is inserted in the source code of the target system, but no code
is needed for transmitting the events to the event recognizer. In fact, the event
data is transmitted from the target system to the event recognizer by a dedicated
monitoring core called ’event dispatcher’ [1].

2.4.6 Summary

A number of solutions in the area of monitoring was covered in three major classes:
hardware monitors, software monitors, and hybrid monitors. While by using hard-
ware monitors the amount of interference on the monitored system is reduced, lack
of flexibility in such monitors is a disadvantage. On the other hand, software monitors provide the ability for interaction between the user and the monitor as well as independence from extra hardware use. However, the amount of interference and overhead in such systems is not negligible. As an alternative method, hybrid monitors were introduced with the claim that they benefit from the advantages of the two mentioned classes of monitors. Nevertheless, such monitors also inherit some disadvantages of the software and hardware monitor.

Despite the latency disadvantage of software monitoring method, its flexibility has provided the framework for the designers to decrease the overhead to some extent in a number of ways. However, the trade-offs between different factors usually imposes challenging constraints. In this thesis work, one solution for this issue is implemented for monitoring systems in which data analysis is supported. In this approach, two methods of monitoring data collection are used to separate data collection and data analysis in order to decrease the overhead caused by monitoring code instrumentation. The next chapters are dedicated to explain different aspects of the solution design, implementation, and evaluation.
Chapter 3

Solution

The proposed solution in this thesis is a software monitoring system which is embedded in Linux kernel and works closely with a middleware real-time module. The main reason for choosing software method is the flexibility that this method provides as well as the bigger volume of data that can be achieved through this approach compared to the hardware approach. Other reasons include hardware limitation and adaptation constraints. However, as mentioned in previous sections, the level of interference in this approach is higher. In monitoring of general properties of the system this can be negligible compared to the benefits that this method has, but in the real-time world this is a major issue. In order to tackle this problem, the data collection and data analysis sections are separated in the proposed monitoring system. In this approach, the data collection section is placed in the module code, and a periodic process is responsible for analysis of the data gathered by the data collection section. However, one can argue that adding a new process to the scheduler could also cause interference with the real-time tasks that need to be executed without the occurrence of deadline misses. The proposed solution for this problem is to have a monitoring process with the lowest possible priority, so the real-time tasks will suffer with lowest amount of latency. This solution is explained with more detail in the next sections.

The first part of this chapter explains the real-time module’s design, how it is connected to Linux kernel, and how it handles different tasks. The second part is dedicated to the design, work flow, and implementation of the monitor as well as a list of design features and limitations.
Chapter 3. Solution

3.1 Module

The main idea is to integrate a real-time scheduler to Linux without modifying the kernel. In order to do so, a loadable module that plays the role of a middle-ware between real-time tasks and the Linux kernel can be a good suggestion. As a matter of fact, this module is responsible for handling real-time tasks by making decisions such as task release, run, and stop. When a real-time task is ready to run, the middle-ware module places it into the Linux run queue by using a system call and changes its state to running state. In the same way, when a real-time task needs to be stopped, the module removes it from the Linux run queue and changes the task state to sleep state. This process results in having only one real-time task in the Linux run queue at each time. After the module is loaded in the Linux kernel, the Linux applications are registered to it. At first, the user will define the task and the server as well as their attributes in the module framework by using a prepared API function. This data is then stored in a data structure in the module. There is a corresponding Linux task for each task defined in the module. When a Linux task gets connected to one of the defined tasks in the module framework, the module scheduling policy will be changed to “SCHED_FIFO” to put the task in the run queue. Whenever the task needs to run, a wake up process is used by the module to run it. Different real-time scheduling processes can be implemented in this method such as EDF, and of course fixed priority scheduling.

3.1.1 ExSched

The real-time module is mounted on Linux kernel via a structure called ExSched [33]. ExSched is a framework developed to implement different schedulers as plug-ins for operating systems without modifying the kernel of the operating system. This framework includes three main components: a core kernel module, scheduler plug-ins, and a library for the user space applications. The core kernel module of Exsched can be mounted on kernels that support loadable modules, and can be accessed by the user space applications by I/O system calls such as ioctl(). In order to implement the preferred scheduling policies, the designer needs to create callback functions which are invoked by the core module. The core module uses the original primitives of the host operating system to implement the designed scheduling policies. In Linux, the core module uses the SCHED_FIFO policy, and uses Linux kernel scheduling functions such as schedule, and sched_setscheduler.
3.1.2 Communication Between Tasks (API)

The real-time module library contains a number of API functions which are used for communication between tasks through ioctl() system call. A list of the functions is presented in table 3.1. When the servers are released periodically, providing their children with a pre-specified chunk of the CPU time in each period. If there is no active task/server inside these periodic servers, they detach their CPU budget allocations. Upon receiving a run message, the module releases all the tasks and servers. When the stop() function is called, the module gives the responsibility of the real-time tasks to Linux scheduler by first stopping the release, and then using a wake up process call to run the real-time task. The API which is called upon finishing a task instance is called the tasks_fiinished_job API. Please
note that in this work we call a task instance a job. As mentioned before, the
status of the task is changed upon receiving the stop or run system calls. When a
job is finished, the function task\_finish\_job(task id) puts the corresponding task to
sleep mode. The task wakes up when its next instance needs to run. The schedul-
ing events that happen in our system are: task and server release, server budget
depletion (periodic and constant bandwidth), finishing a job, and tasks or servers
exiting the system. The possibility of preemption of the tasks and the way they
can be preempted depends on the scheduling policy that is being applied. The
same way as explained the constant bandwidth server policy in the introduction
section, a task can only run if its server can preempt the running server or task.

Table 3.1: A list of some of the module functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>run()</td>
<td>Module releases tasks and server</td>
</tr>
<tr>
<td>stop()</td>
<td>Linux kernel takes control of scheduling</td>
</tr>
<tr>
<td>create_task()</td>
<td>New task is created</td>
</tr>
<tr>
<td>detach_task(task id)</td>
<td>Linux task is detached from the module</td>
</tr>
<tr>
<td>release_task(task id)</td>
<td>Task is released</td>
</tr>
<tr>
<td>task_finish_job(task id)</td>
<td>Task status is set to sleep mode</td>
</tr>
<tr>
<td>attach_task_to_mod(task id)</td>
<td>Linux task is attached to module</td>
</tr>
<tr>
<td>set_task_param(id, period, deadline, exec time, priority )</td>
<td>Attributes of task are set</td>
</tr>
</tbody>
</table>

3.1.3 Tasks and Queues

A table of the attributes provided for the task definition in the suggested module is
provided in section 3.3. As can be seen in that section, many descriptor elements
can be assigned by the user. The period\_timer attribute is used to run tasks
periodically, and the last line is related to the constant band width server, to which
the task gets attached to. Note that some of the attributes that the user does not
want to add can be ignored. For example, only one of the server types should be
chosen by the user. Release time, execution time, absolute deadline, and relative
deadline are other task characteristics that the user can define. A point worth
mentioning is that these characteristics will also be used by the monitoring section.
The ready queues have the tasks organized based on their priorities. As mentioned
before, two types of scheduling policies are supported in the suggested module:
EDF and fixed priority scheduling. The task deadlines in these two scheduling
policies are different. In EDF, the absolute deadline of the tasks determines their
priorities while in fixed priority scheduling, their priorities are directly assigned
by the user. The ready queue is implemented in the form of a linked list using the
list_head structure provided in the Linux kernel. The two functions developed for
inserting and deleting tasks to or from the ready queues are: insert_queue(queue,
entity), and delete_queue(entity), respectively.

3.2 Monitor

The suggested monitoring system uses a software approach, meaning that it does
not use any external hardware. As explained in previous sections, the main disad-
vantage with software approach is the latency that it adds to the target system.
This latency is mainly the instrumentation latency in tracing method, which can
distort the performance of the target system regarding timing, hence, result in
inaccurate monitoring data. On the other hand, the sampling method has the dis-
advantage of missing some events as it is very dependent on its frequency-based
identity. Thus, the proposed approach separates the two main parts of the mon-
itoring system, which are data collection, and data analysis. In order to avoid
event misses, the data collection is done through tracing methods, i.e. instru-
menting data collection code inside the module. On the other hand, the data
analysis task is implemented using the sampling method, meaning that analysis of
the collected data is done through a thread that is added to the module scheduler.
In this thesis work, this thread is called the monitoring task. Using this approach,
on one hand, a part of the latency imposed by the instrumentation of analysis
code is removed, and on the other hand, the danger of missing events of interest
by sampling data collection is avoided. Figure 3.2 shows how the monitoring is
divided in to two separate methods in order to help tackle the low frequency and
latency problems.

Despite the fact that using tracing method for data collection can help us avoid
missing events of interest, the use of a sampling thread for analysis can put us in
risk of having unanalyzed data. this can happen due to the fact that the sampling
thread might miss a deadline, especially because it is supposed to have the lowest
possible priority among all tasks that enter the scheduler (priority assignment is
explained in more detail later). Thee solution to this issue that is used in this
thesis work is to keep the monitored data in a a separate file, or in Linux kernel
buffer when an event of interest happens. Thus, whenever the monitoring task
is executed, the data is fished from the file, or the kernel buffer, which results
in missing less amount of data because of low sampling frequency. As a matter
of fact, there is a trade-off between higher sampling frequency, which in our case
is related to the monitoring task priority, i.e. missing less data, and additional latency and distortion to the normal execution flow of the target system.

![Diagram](image)

**Figure 3.2:** A schematic view of the data collection and analysis process in the monitor

The Linux kernel buffer is of circular type, also known as ring buffer. The advantage of a ring buffer is that it does not need to have its elements shifted around when one element is used. This means that unlike a non-circular buffer, it is not necessary to shift all elements in the buffer when one is used. In Linux, the
command `dmesg` is commonly used to access the Linux kernel buffer through the function `syslog()`, which is responsible for reading and/or clearing Linux kernel message ring buffer. The kernel ring buffer is located in proc file (explained in the second chapter), and under the directory `/proc/kmsg`, so the monitoring thread is able to have access to the data in the kernel buffer through this address. A Diagram of the interaction between the application and the buffer is shown in figure 3.3.

![Diagram of the interaction between the application and the buffer](image)

**Figure 3.3: Accessing Linux Kernel Buffer through applications**

The most important resource for monitoring of a system’s timing behavior is the exact time at which events of interest happen. This data is represented by the time stamps of those events of interest. In order to handle the scheduling tasks, we use the ordinary Linux timer (low-resolution timers) provided in kernel/timer.c. Using this timer, the start time of the running tasks is saved in their corresponding data.
structure. Also, task attributes are converted to system clock tick counts inside the system for execution of the task to take place. By using the time stamps of starting and stopping times of the jobs we can measure useful monitoring data such as execution time of the task. The definitions provided in background section are used to measure these timing properties.

As mentioned before, whenever a job execution needs to be finished it should call an API function that puts it in to sleep mode until its next execution instance needs to run. Hence, the module is informed when a job is finished. The monitor takes advantage of this possibility to record the time stamp of a job completion, i.e. it saves the time stamp of the instance that sleep mode call happens. This time instance is used by the monitoring process as the job finish time. Note that the basic premise of this monitoring project is that tasks run periodically.

![Diagram of the monitor in combination with the real-time module.](image)
Table 3.2: Priority rates of different task types

<table>
<thead>
<tr>
<th>Process Type</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard Real-time</td>
<td>1</td>
</tr>
<tr>
<td>Soft Real-time</td>
<td>2</td>
</tr>
<tr>
<td>Non-real-time</td>
<td>3</td>
</tr>
<tr>
<td>Monitoring</td>
<td>4</td>
</tr>
</tbody>
</table>

The monitoring system consists of a data collection section inside the real-time module, and a monitoring process that runs with the lowest possible priority, so it does not interfere with the execution flow of normal Linux tasks (real-time or non-real-time). Table 3.2 shows a rating of the priorities of different task types in this concept. As shown, hard real-time tasks have the highest priority and the monitoring task has the lowest priority. However, in most applications, preemptions caused by the interrupt and interrupt times are in most cases inevitable. Please note that the real-time module scheduler benefits from a higher priority than the Linux kernel for task scheduling. The output of the monitoring process is written in a log file, which is what the user can see. Upon the completion of a job, the data collection section inside the module sends the task data as well as the time stamp of the job completion to the message buffer of the kernel. The monitoring task then reads this data from the buffer, upon execution, and analyzes it. The output of the task data contains: start time of the job, completion time of the job, execution time of the job, and the necessary pre-specified data of the task including its deadline and execution time. Note that the users can change the data they want to be sent to the process, based on their preference and the amount of data they need or latency they want to avoid. Then, the monitoring process uses this data and the time stamps to analyze the system’s timing behavior using the definition of each timing entity, and writes the results in a log file which will be what the user will see. This analysis, can include the actual execution time of the tasks, the actual periodicity of tasks, and most importantly, deadline misses and successful task completions with their time stamps. When a job ends its execution after its deadline time point, a deadline miss happens. A diagram of the monitoring work flow is shown in the picture below.

An important functionality that is added to the monitoring task, beside from calculating the deadline misses and real execution times, is predicting the deadline misses. As a matter of fact, when the real execution time of a task gets closer and closer to its deadline, the monitor can alarm the user or take a proper action. This is implemented by a loop that checks if the execution time of a task is converging...
to a specified portion (say, 80 percent of the deadline) of the deadline of that task. However, this functionality cannot be used in all cases due to the unpredictable behavior of real-time tasks in real-world, but in the case of static task creation and periodic task execution, this feature can work as a good possibility to avoid the deadline misses.

In Figure 3.6 a sequence diagram of the monitoring process is provided. This diagram shows the sequence of the process since the time that the scheduler starts invoking and executing the tasks. Two tasks are being executed in this example, T1, and T2. As can be seen in Figure 3.6, first the scheduler starts the task that has higher priority (here it is T1), and upon noticing the completion of the first instance of T1, the monitoring data collection section collects the data corresponding to it as well as time stamp of important events. Then the monitoring task, which is run in the background, fishes this data from the data collection section, and performs the analysis processes. Note that in this example, T2 does not need to be run right after T2, otherwise, the monitor would need to wait until T2 completes its job, because of the higher priority of T2 compared with the monitoring process.
3.2.1 Design Features

- User friendliness: the monitor provides analysis based on the raw data that it receives from the module. Using this data it is very easy to have a thorough analysis of the timing behavior of the system during the run time at a specific time or on the process execution trace.

- Modifiability: the monitor works as a periodic task separate from the kernel, which gives it the flexibility that the user can change its features according to the needs and the latency constraints easily.

- Light design: the implemented data collection method is sampling, which as explained before, takes samples of the system performance during run-time and checks for requirement violations. Since significantly less time needs to be spent to achieve sampling than to instrument the monitoring code for tracing, the design of the monitor is lighter.

- Flexibility: two scheduling methods are covered by the module (EDF and FPS).
3.2.2 Design Limitations

- Starvation suffered by Monitor: as the monitor is implemented as a background process, it can suffer from starvation to an extent that it never gets the chance to execute. For example, if a task set is consisted of tasks with such periods and execution times that scheduler always remains busy with scheduling the tasks and has no idle time, then the monitor will never be scheduled. Severity of problem can be reduced by implementing the monitor as a prioritized process instead of a background process. In this way, it will be scheduled by the scheduler along with other prioritized processes. However, this will affect the scheduling of user defined tasks.

- Frequency: the period of the monitor should be assigned in a way that we can make sure the monitor will miss as few task instances as possible.

- Support for dynamic tasks: dynamic tasks are not supported by the module, and all tasks are defined before run-time.

3.3 Implementation

In this section, the implementation of the design discussed in the previous section is provided.

The list below corresponds to the task attributes assigned by the user in the real-time module. These attributes are briefly discussed in section 3.1.3.

```c
struct Task
{
    struct list_head head;
    int id;
    int priority;
    int state;
    int cnt; /* job number */
    int missing dl flag;
    unsigned long period;
};
```
unsigned long release_time;
unsigned long exec_time;
unsigned long relative_deadline;
unsigned long abs_deadline;
struct task struct *linux_task;
struct timer list period_timer;
struct Server *cbs;
}

An example of a task structure is presented below:

```c
int main(int argc, char* argv[]) {
    task id = atoi(argv[1]);
    attach_task to mod(task id);
    while i job no do
    /* periodic job */
    task end job(task_id);
    end while
    detach_task(task_id);
    return 0;
}
```

Pseudo code for the monitoring section in the module is presented below:

```c
int main(int argc, char* argv[]) {
```
task id = atoi(argv[1]);
attach_task to mod(task_id);

while i,j job no do

Fish the Time Stamps/* receive time stamps and task ids from Linux kernel buffer*/

Analyze the data;

Print the data in to another file;
task end job(task_id);
end while;
detach_task(task_id);

return 0;

}

The listing below shows an example of the monitoring log file that the user sees at the end of the process.

PID =65000 /* task with the ID 65000 is started*/

Start time in ticks=8500 /* task has started running on 8500 Linux kernel clock ticks after starting the system*/

Deadline in ticks=8560 /* the time instant at which the task deadline is*/

Period of the task in ticks = 12 /* the specified period of the task execution*/

Completion time of the task in ticks =8568 /* the time instant at which the task execution is completed*/

Actual execution time in ticks= 6 /* the execution time of the task in practice*/

The distance to task deadline= 4 /* the difference between task completion and task deadline*/
Chapter 4

Evaluation

In this section, a case study and evaluation of the monitoring mechanism is presented. For this case study, four periodic tasks are defined in the module. Three of these tasks perform a simple calculation, and one task is responsible for monitoring of the system’s timing behavior. The three normal tasks are called T1, T2, and T3, and the monitoring task is called M. In the list below you can see the period and execution time of each of the tasks. In this example, we assume that the deadline and period of the tasks are the same. This information is used by the module scheduler to schedule them based on EDF scheduling policy. It is worth mentioning that the monitoring task should impose as little interference as possible to the rest of the tasks. Hence, in EDF, the monitor should not have a small period or deadline, because the priority of a task in EDF is dynamically assigned to it based on how close the system is to its deadline. The task with the earliest deadline has the highest priority. Consequently, if we assign a short period to the monitoring task, the system gets close to its deadline frequently, and obtains the higher priority frequently, thus, it preempts the other tasks too often, which results in its high interference with the system’s normal execution. This issue is displayed in figure 4.2, in which task M has a period of 400 milliseconds instead of 700 milliseconds. As can be seen, in this case more deadline misses happens among the real-time tasks due to the interference caused by the monitor. In fact, this issue can be more critical when real-time tasks arrive to the system dynamically. As it is shown in figure 4.1, task M executes every 700 milliseconds, which is the longest period among all of the tasks. The task attributes in the case study were decided in a way that deadline misses occur during the execution. As can be seen in figure 4.1, deadline miss happens for T2 in its second and fourth period, for T1 in its fifth period, and for T3 in its second and third period. Note that in EDF, if the time span between the current system time instance and the
deadlines of two tasks is the same, the scheduler can pick any of the two tasks to be executed first.

- T1: Execution time: 100, Period: 300
- T2: Execution time: 100, Period: 300
- T3: Execution time: 300, Period: 500
- M: Execution time: 100, Period: 700

---

**Figure 4.1**: scheduling of T, T2, T3, and the monitoring task (M) with EDF

---

**Figure 4.2**: The effect of a high frequency monitoring task on target system’s behavior

---

Figure 4.4 shows a part of the corresponding monitoring output that the user can see in the monitoring log file. In this trial, 10 jobs were executed for each task. As can be seen in this picture, the time stamp at which a task starts and stops executing, the actual execution of the tasks, a warning message in case of deadline miss occurrence, the total overhead of the tasks, and tardiness of the tasks are among the information which is provided by the monitor. Tardiness of a job is
defined as the difference between its completion time and its deadline, if the job misses its deadline [34]. Tardiness is usually useful for soft real-time tasks, as it shows how late a job that has missed its deadline was completed. Consequently, a system with smaller tardinesses shows a better timing performance. Also, as explained before, the monitor can notice when a task completion time gets closer and closer to its deadline. As a matter of fact, this feature uses the same concept as the task tardiness, but for the jobs that do not miss their deadlines.

Figure 4.3: T1, with a deadline equal to 1800, has a tardiness of 120 milliseconds.

Now assume that we want to schedule our tasks with fixed priority scheduling algorithm. Figure 4.5 shows an example of this case. In this example, two tasks, named T1 and T2 are the real-time tasks, and task M is responsible for monitoring of the timing performance. Since in fixed priority scheduling the priority of the tasks determines which task should be executed first, the low amount of interference by the monitor is fulfilled by simple assigning the smallest priority to it. The priority and deadline of each task can be seen in the section below. As shown in figure 4.5, task M misses its deadline in its first period. This is not a very unpredictable case, as we know that M has the lowest priority. when M starts executing in its second period, it shows the information which was saved previously by the module in Linux kernel buffer. However, if M keeps missing its
Figure 4.4: A part of the monitoring output for EDF scheduling of T1, T2, T3, and M
deadline over and over, the monitoring data that the user reads cannot get updated. Also, some monitoring data can be missed if the kernel buffer gets filled up and the monitoring task does not get the chance to be executed for a long period of time.

- T1: Priority: 100, Period: 300
- T2: Priority: 200, Period: 400
- M: Priority: 100, Period: 500

![Diagram of scheduling of T1, T2, and the monitoring task (M) with FPS](image)

One could wonder if it is possible to design a mix of the above scheduling methods in order to have better control over the deadline misses. A possible way to implement such an idea would be the case that the real-time tasks were scheduled through EDF scheduling policy and the monitoring task was scheduled by FPS. This way, the real-time tasks can be scheduled by a very efficient scheduling policy, and on the other hand, the monitoring task, which needs to be executed with the lowest priority, can be scheduled by a policy in which the user is able to have a simpler view on its priority, and thus, its execution. To answer this question we can look at the formula for schedulability in EDF scheduling when the deadlines of the tasks are the same as their periods. The point is that according to the formula, the CPU resources are not free for the monitoring task if deadline misses happen for the real-time tasks. This is because the utilization bound for EDF in our case (deadlines are equal to periods) is 100% [35].

\[
\sum_{i=0}^{n} \frac{C_i}{T_i} \leq 1
\]

- Overhead

Generally, the overhead of the monitoring code can be calculated in two ways: instrumenting codes that measure the time that a method takes to execute, and
using an external time monitor, which indicates what part of the code executed at any given time. The instrumentation method can be done manually by adding code that records the time before and after methods of interest. Using Aspect-Oriented Programming (AOP) libraries to create Aspects that record execution time of the codes is another way of instrumenting overhead measurement code. Obviously, the instrumentation approach can utilize other facilities depending on the programming language. For using an external examiner, a thread can be used to look from time to time into the other threads and give us data about their activity. As explained previously, in our case we use the manual instrumentation method, in which monitoring code in placed inside the target code.

Overhead calculation was done in three steps. First, the Linux system was executed without the module. In the next step, the module was run with the assigned real-time tasks but without the monitoring task, and in the third step, the module was run with both the real-time tasks and the monitoring task. The context switch overhead was not included in the calculation of overhead imposed by the module and the monitoring task, because this overhead corresponds to the context switching that Linux kernel is responsible for. In fact, the module and monitor overhead include the over head of the task and server release APIs, which are run at the beginning of the module execution, and the over head of the task finishing API, which is run at the end of the module execution. The overhead was measured using time stamps of the execution length of the timer handlers and the time stamp of the task_finishjob API function. The time stamp was obtained at the beginning and at the end of the mentioned APIs and then, these two time stamps were subtracted, and the total amount was divided by the total period of time that the experiment was run. The total measured overhead of the module was less than 2%, (about 130 milliseconds) and the monitoring overhead was between 0.1% and 0.5% in the tested cases. This result is in agreement with the light design feature of the monitor that was discussed before. The pseudo algorithm for the overhead calculation for task finish API is shown in the section below.

---

**Api_Finish_job starts;**

**Get_Time (TimeStamp_start) gets the time stamp at the beginning of API execution;**

**Api_Task_Finish_Job API task code is run;**

**Get_Time (TimeStamp_End) gets the time stamp at the end of API execution;**
TimeSpan = Times_Sub(TimeStamp_end - TimeStamp_Start);

Total overhead += TimeSpan_Nanosec + TimeSpan_Sec*100000000;

The overhead imposed by the monitoring thread is negligible due to its lower priority and the simplicity of the task. This overhead can get modified by the user by changing the period of the thread, as described in the evaluation section. However, the user does not have as much freedom when it comes to the overhead imposed by the data collection code execution. Nevertheless, the non-manual data collection approaches such as using AOP libraries, might provide more flexibility regarding overhead.

![Figure 4.6: The measured amount of the overhead imposed by the monitoring thread with different periods](image)

Figure 4.6 shows the relation between the period of the monitoring task and the overhead that it imposes to the system. As mentioned previously, reducing the period can increase the interference to the target system, consequently, target system performance gets distorted. Figure 4.7 on the other hand, shows the effect of this distortion on the execution flow of tasks in the system. The rate of deadline misses increases by reducing the period of the monitoring task. In fact, when the ratio of the period of the monitoring task to the other tasks (Horizontal Axis) decreases, the percentage of deadline misses raises. Also, the period level of normal tasks shows the same effect on the rate of deadline misses.
Figure 4.7: Rate of deadline misses corresponding to period of the monitoring task
Chapter 5

Summary and Future Work

5.1 Summary

The run-time monitoring system proposed in chapter 3 is capable of providing crucial information regarding the timing behavior of Linux operating system. The monitor consists of a data collection section, which is implemented inside a middle-ware module that is responsible for handling real-time tasks, and a monitoring task which is responsible for sending the data to a log file in a user friendly way for the user. The monitoring task is run by the module scheduler with the lowest priority in order to impose as low amount of interference as possible. Upon starting the task execution, the monitor collects data from the Linux real-time middle-ware module in form of time stamp of the events of interest and corresponding processed data. This data is then presented in a log file in a user-friendly format. The processed data includes time stamp of task start and completion, deadline misses, the actual execution time of the tasks, task tardiness, and module and monitor overhead. Moreover, the distance between task completion time and its deadline is used to predict deadline misses in the system based on the current behavior of the tasks in the system.

5.2 Future Work

This thesis work can be improved and extended in order to obtain a better design for the monitor. A number of future tasks for this purpose are provided in the sections below:

- One main limitation of this system is the overhead that the module imposes on Linux kernel. This is due to the time that the CPU should assign on running the module scheduler. This classic issue is important in our work...
due to our interest in system’s behavior in the real-time domain. One way to tackle this issue is to use more efficient queue structures. Another issue is the overhead added by the monitoring segment in the module.

- The monitoring analysis section adds some latency to the target system as well. However, as explained before, there is a trade-off between the volume of analysis and the latency the monitor imposes to the system. On the other hand, the monitor can miss some events of interest due to its low priority, and its frequency. In order to tackle this issue, the monitor can be run with a higher frequency, although in practice it has been observed that it causes extremely high pressure on the Linux kernel. Hopefully, in future kernel designs this will not be as big of an issue.

- The monitor is dependent on the scheduling algorithms that the system supports. Currently, the system supports EDF and FP scheduling algorithms. Other scheduling methods can be supported to increase efficiency of both the monitor and the target system.

- A feature that can be added to this system is adding support for dynamic tasks. In fact, in most of the industrial real-time systems tasks, such as event-based tasks, are created and handled dynamically. The importance of monitoring of dynamic tasks is even more evident due to the complexity and unpredictability that dynamic tasks can add to real-time systems.

- Support for monitoring of sporadic tasks is another important feature that can be considered for future work on this topic. Real-time concepts such as soft and hard real-time tasks and tardiness of tasks are commonly discussed in sporadic environments, and as a matter of fact, by adding this feature, the system can be improved to be more useful in real-world situations instead of a research prototype.

- Another important feature that can be added to this design is support for multi processor systems. Research on common scheduling methods in multicore systems will be helpful in achieving this goal.
Bibliography


