Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Orchestrate the Mixed-Criticality Melody: Reconcile Temperature and Safety

Semester Thesis

Max Millen

mmillen@student.ethz.ch

Computer Engineering and Networks Laboratory
Department of Information Technology and Electrical Engineering
ETH Zürich

Supervisors:
Dr. Rehan Ahmed
Dr. Pengcheng Huang
Prof. Dr. Lothar Thiele

February 14, 2017

Acknowledgements

First, I would like to express my sincere thanks to Prof. Dr. Lothar Thiele for giving me the opportunity to work on this project within his research group of the Computer Engineering and Network Laboratory at ETH Zürich. This project has given me the opportunity to deepen my knowledge of mixed critical systems, a topic that is of great interest to me. I am most grateful to my two supervisors Dr. Rehan Ahmed and Dr. Pengcheng Huang. Their support and advice throughout my semester thesis has been invaluable. Finally I would also like to thank my father, who helped reviewing the text of this report.

Abstract

Recent developments in mixed-criticality systems have lead to the use of multi-core platforms for cost and performance reasons. These multi-core platforms bring a number of challenges into the mixed-criticality systems. One of these challenges is thermal interference between tasks of different criticality levels. As modern CPUs are thermally constrained, they have to be slowed- or shut-down when they approach a critical temperature. If a low critical task causes the CPU to slow down then timing guarantees for high critical tasks are difficult to maintain. Dr. Rehan Ahmed and Dr. Pengcheng Huang have developped a server based scheduling scheme called "Thermal Isolation Server" [12] which is taking these thermal interferences into account in its scheduling strategy. The concept of the TI-Server has been proven theoretically. The goal of this thesis is to create an implementation of the TI-Server and to evaluate the TI-Server scheme on a 4 core platform. The evaluation framework is based on an existing scheduler that is running on top of a Linux kernel. This scheduler is called "Hierarchical Scheduling Framework" [13]. The extensions done to implement the TI-Server make use of existing HSF structures and add new functionalities that are compatible to the existing ones. In addition, several monitoring functions have been added to HSF in order to evaluate the TI-Server. The monitoring functions include thermal monitoring of the temperatures of all cores and the recording of traces for schedulers and servers. These traces record the execution state of the threads. Several measures are taken in order to give the testing platform the best possible real-time capabilities. Verification tests have been performed to ensure a correct implementation of the TI-Server. Thermal measurements were performed on the test platform to calibrate the CPU thermal model. This model is used in the TI-Server theory. Finally experiments were run to evaluate the TI-Server on the 4 core platform. These experiments clearly demonstrated the effectiveness of the TI-Server concept to provide thermal guarantees on the mixed-criticality system.

Contents

Acknowledgements **i**

Abstract **ii**

1 Introduction **1**

1.1 Motivation 1

1.2 Related Work 2

1.3 Overview 2

2 Thermal Isolation Server **3**

2.1 TI-Server 3

2.2 Implementation and evaluation requirements 4

3 Structure of the TI-Server evaluation framework **5**

3.1 Thermal Isolation Server 6

3.2 HSF Extensions 10

3.2.1 Thread priorities 10

3.2.2 Temperature monitoring 10

3.2.3 Jitter measurement 11

3.2.4 Traces and overhead calculation 11

3.3 Enhancing real time capabilities 11

3.3.1 Preempt_rt 12

3.3.2 Page locking 12

4 Implementation **13**

4.1 Temperature measurement 13

4.1.1 Temperature measuring task 13

4.1.2 Temperature measuring mechanism in Resource Allocator 13

4.1.3 Temperature records 14

4.2 Jitter measurement 14
4.2.1 Activation time and firing time 14
4.2.2 Jitter records 15
4.3 Page fault analysis 15
4.4 Trace recording and overhead calculation 15
4.5 Pre-partitioned EDF 16
4.6 TI_server 17
4.6.1 Implementation 17
4.7 Modifications to the HSF parser 21

5 Verification of the implementation 22
5.1 Jitter measurement 22
5.2 TI-Server implementation verification 23

6 TI-Server evaluation experiments 29
6.1 Thermal modelling 29
6.2 Thermal isolation server experiments 31

7 Conclusion and outlook 37
7.1 Conclusion 37
7.2 Outlook 37

A Installation guide 1

B Class descriptions 7
B.1 Code structure 7
B.2 Pre-partitioned EDF 9
B.3 TI_server 9
B.3.1 The TIS class 10
B.3.2 The Server-TTL 12
B.3.3 The TISDispatcher 12

C Creating an XML file for TI-Server experiments 14
C.1 main Element, the Simulation element 14

C.2 runnable element 14
C.2.1 Scheduler type 15
C.2.2 Server type 15
C.2.3 Worker type 16
C.3 element hierarchy 16
C.3.1 TI-Server example 17

D Original Project Assignment 19

Bibliography 25

List of Figures

2.1 TI-Server 4

3.1 Evaluation framework 6
3.2 State description of TI-Server 7
3.3 Illustration of the TI-Server mechanism 9

4.1 Illustration of the TI-Server mechanism 20

5.1 EDF verification test for TI-Server 24
5.2 TI-Server test on multi-core 27

6.1 Temperature traces for First-Fit 33
6.2 Individual temperature traces for First-Fit 33
6.3 Temperature traces for TI-Server 35
6.4 Individual temperature traces for TI-Server 35

B.1 The thread hierarchy after extensions to HSF 8

List of Tables

3.1 Thread Priorities 11

4.1 Trace action types 16

5.1 First 6 server active times for server on core 1 28
5.2 First 6 server active times for server on core 2 28
5.3 First 6 server active times for server on core 3 28

6.1 Thermal Calibration experiments 30
6.2 FMS Parameters 32

Chapter 1
Introduction

1.1 Motivation

In recent years the performance requirements of real-time applications, in domains such as automotive or aerospace, have increased significantly. This has lead to the transition from single core systems to multi-core systems. In mixed-criticality systems, multi-core platforms bring advantages, but they also introduce some challenges. The advantages of a multi-core system are performance increase and cost, weight and energy consumption reduction. On the other hand multi-core system challenges include the estimation of worst case execution time (WCET) of individual tasks and thermal interferences between the different cores. In this thesis the focus is on thermal interferences. Thermal interferences occur because modern CPUs are thermally constrained. This means that if a core exceeds a maximum temperature it has to be slowed down in order to lower its temperature. Furthermore, a task running on one core not only heats up the core it is running on, but also all other cores. In mixed critical systems, the heat-up caused by a low critical task could influence the performance of the CPU cores running high critical task. In order to cope with these interferences Dr.Ahmed and Dr.Huang have proposed a server based scheduling called Thermal Isolation Server (TI-Server)[12]. The TI-server is given a thermal budget. This budget upper-bounds the temperature increase caused by the server load, i.e. all tasks executed by the server. In the scheduling scheme high critical and low critical tasks are thermally isolated from each other. This means that the HI and LO tasks have a thermal budget and within this budget they are isolated. The budget of the high critical tasks is based on the maximum temperature. The remaining budget is then left for the low critical tasks and is distributed onto the TI-Servers.
In this thesis a implementation of TI-Servers is created. Also a evaluation platform is created in order to perform evaluation tests on the TI-Server concept. This evaluation platform consists of the implementation of the TI-Server itself and includes several monitoring mechanisms that allow performance evaluation of the TI-Server . In this thesis a 4 core test platform was used. It is a Lenovo

Thinkpad T440p consisting of a Intel i7-4700MQ CPU. The implementation of the evaluation platform is realized through extensions of the "Hierarchical Scheduling Framework" [13]. The main extensions are the implementation of the TI-Server and a temperature measuring mechanism. For the evaluation tests real-time capabilities are added to HSF and to the operating System.

1.2 Related Work

Previous to this thesis the "Hierarchical Scheduling Framework" has been created as described in[13]. This is a scheduling layer that operates on top of the Linux kernel, in the user space. HSF is divided into 3 parts:

- The *parser* reads all scheduler and task configurations. These configurations are specified in a XML file.
- The main part is the *Simulation* part. It consists of dispatchers that spawn new job events, of one or more schedulers and of workers that handle the task executions. For the schedulers a number of different scheduling algorithms are implemented. All these different elements are defined as separate threads. The scheduling is then done by assigning priorities to the different threads. In addition to the thread priorities the core on which a thread is executed can be specified, making HSF multi-core compatible. The underlying operating system will schedule the threads of HSF according to their assigned priorities
- The third part of HSF is the *statistics* part. In this part a number of traces are recorded. From these traces metrics such as missed deadlines or overheads are calculated.

1.3 Overview

The thesis is structured as follows: In chapter 2 the Thermal Isolation Server is explained in further detail. Chapter 3 introduces the different parts of the evaluation framework. Chapter 4 describes the implementation of these parts. Chapter 5 describes the tests that are performed in order to verify the implementations described in chapter 3 and 4. Chapter 6 presents the evaluation experiments of TI-Servers and their results run on the 4 core platform. Chapter 7 is a conclusion and short outlook. In appendix A a complete installation guide of the framework is given. Appendix B gives a detailed description of the classes that were added to HSF. Appendix C presents how to build a XML simulation file for the extended HSF, in order to the perform TI-Server tests.

Chapter 2

Thermal Isolation Server

This chapter describes the concept of the TI-Server. The goal of TI-Servers and their functionalities are explained in a first part. In the second part requirements for an implementation and evaluation of a TI-Server are presented.

2.1 TI-Server

The *Thermal Isolation Server* is a server based scheduling scheme that limits the thermal interferences of applications running on different cores. For this a TI-Server S_i is always executing on one fixed core j, its self core. The TI-Server S_i is then characterized by a thermal budget Λ^i. Λ^i is the upper bound system-wide temperature increase caused by the tasks that are executed on the server S_i. Furthermore the TI-Server is scheduled by a fixed periodic schedule. The server S_i is then characterized by a period P_i and a utilization U_i. This means that the server executes tasks for $t_a = P_i * U_i$ it then remains idle during $t_c = P_i(1 - U_i)$. During the active time t_a of the server task are scheduled according to a *Earliest Deadline First* policy. The ratio of P_i and U_i directly influence Λ^i (for a precise description of this relation see [12]).

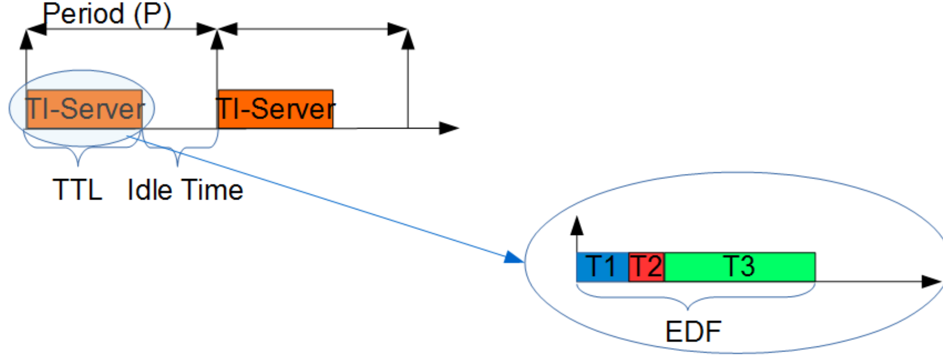

Figure 2.1: TI-Server

2.2 Implementation and evaluation requirements

To realize TI-Server, a server has to be created that can be activated and de-activated. During the active time the sever must be able to schedule its load according to a EDF scheduler. During the deactivated or idle time t_c the self core has to be idle, meaning that **no** threads are executed. For proper functioning of the TI-Server the implementation must be able to spawn activation events for the TI-Server at P_i intervals. It must also be able to spawn job events for the tasks that are under the control of the server. These jobs are then scheduled using EDF when TI-Server is active. Furthermore the implementation must include a mechanism that will deactivate the server after t_a, or if there are no threads left to schedule.

For the evaluation of the TI-Server some monitoring capabilities have to be implemented. As Λ gives a thermal bound on the temperature increase, it is important to be able to monitor the temperature of all cores of the system. With the temperature monitoring, it will be possible to verify whether temperature increase exceeds Λ or not. In addition to the temperature measuring function, it is necessary to record the execution traces of servers and tasks. This is needed to confirm the timing guarantees that are given by the TI-Server. From the traces should be calculated the server overhead as it influences the optimal P_i U_i relation. The server overhead is the time that is needed by the server to process the scheduling scheme.

Chapter 3

Structure of the TI-Server evaluation framework

In this chapter the prototype of the TI-Server and the evaluation framework are described. Figure 3.1 is an illustration of the different parts of the evaluation framework. It consists of 4 components that are running on a test platform. The main part, the “Scheduler” part, is a Scheduler framework that is running on top of a Linux kernel. It is based on the “Hierarchical Scheduling Framework” (HSF [13]). HSF is already multi-core capable, but for the multi-core use of the TI-Servers a new configuration has to be created, see 4.5. The “LO Scheduler” consists of the scheduling of the low critical tasks. These tasks are scheduled by TI-Servers. The proposed design of the TI-Server is described in 3.1. The “HI Scheduler” consists of all tasks that are classified as high priority tasks. These tasks are not executed inside a TI-Server, they are executed on a separate core. On this core an EDF scheduler will be running. This is already implemented in HSF. The monitoring part is explained in 3.2.2. Temperatures of the cores and state traces of the different threads are recorded for evaluation purposes. TI-Servers are used in a real-time environment, therefore it is important to ensure that the test platform is real-time capable. The measures that are taken to ensure these capabilities are described in section 3.3.

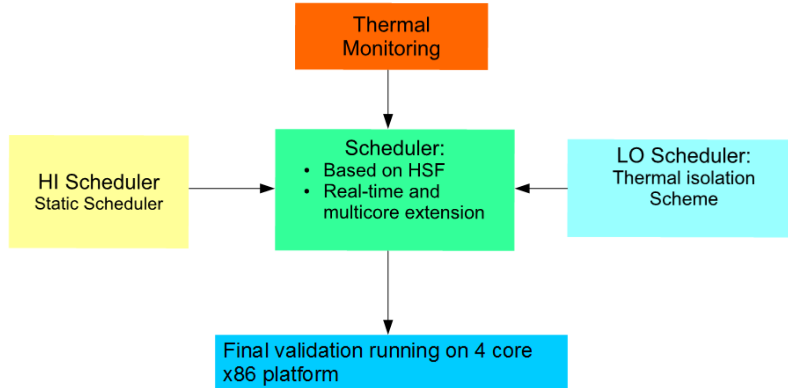

Figure 3.1: Evaluation framework

3.1 Thermal Isolation Server

As seen in chapter 2 the TI-Server S is characterized by a period P and a utilization U. This means that in time P the server is active for a maximum of t_a = PxU per period P and is inactive for t_c = Px(1-U) in which the core cools down. During t_a the server will schedule its load according to an EDF schedule. During the cool-down time the core has to remain idle such that the temperature can drop. The TI-Server is represented by a state machine that has two states (see figure 3.3):

- *active state*: In this state the server schedules its load
- *idle state*: In this state the server remains inactive. No threads are executed on the CPU core.

During t_a the server has to be in *active state* and during t_c it is in *idle state*. As illustrated in 3.3 the server must be able to switch states in both directions.

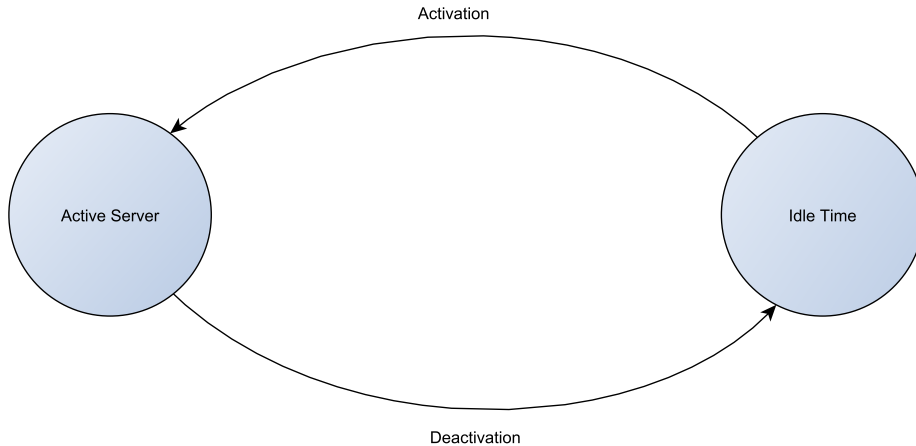

Figure 3.2: State description of TI-Server

In this thesis a TI-Server design consisting of 4 different thread types is proposed. These threads handle the server and its state transitions. The threads are described below and the procedure is illustrated in figure 3.3:

1. The *TI-Server* thread itself. This thread schedules the server load when active. The thread is in *active state* when it is not blocked by any semaphores and when it has active priority, see 3.2.1. For *idle state* the TI-server has to be deactivated. This means that the thread itself is blocked by a semaphore and all its load is deactivated.

2. *EDF scheduler*: The TI-Server can be interpreted as a hybrid between a worker and scheduler. The TI-Server, similar to the worker, is activated by new job events. During its active time the TI-Server acts like a scheduler thread. A EDF scheduler is thus used for the activation of the TI-server when a "new period" event arrives. The EDF scheduler also deactivates the TI-Server, this is done at the moment a "finished period" event is registered.

3. The *Server-TTL* is responsible for the transition form *active sate* to *idle state*. At the beginning of the active time a count-down time, time-to-live (TTL), is set for the Server-TTL by the TI-Server. This TTL corresponds to t_a. Once the TTL is, set the thread sleeps for TTL. When the thread wakes it registers a "finished period" event at the EDF scheduler, which will then deactivate the TI-server. After that the Server-TTL will block itself with a semaphore. This semaphore is released at the same time the TTL is set.

4. The *TIS-Dispatcher*: This thread is responsible for the transition from *idle state* to *active state*. At the end of an active time a count-down time,

cool-down time is set for the TIS-Dispatcher by the EDF scheduler. The TIS-Dispatcher will then sleep for that time and when it wakes up a "new period" event is registered at the EDF scheduler. As a consequence of this the EDF will activate the TI-Server and the *active state is reached.* After registering the "new Period" event the TIS-Dispatcher is blocked by a semaphore. This semaphore will be released by the EDF scheduler at the same time the TI-Server is deactivated.

In the description of the TI-Server thread it is mentioned that all load has to be deactivated for idle time. In the original implementation of HSF a thread was deactivated by setting its priority to inactive priority. If nothing else was running an idle thread would prevent the inactive thread from executing, because the idle priority is higher than the inactive priority. In the case of the TI-Server however, the core has to remain idle during cool-down time. But if no idle thread is used nothing is blocking the inactive threads from executing. All threads therefore have to be blocked either by a sleep function or by a semaphore. In **??** the implementation of this is described in detail.

For the TI-Server mechanism two special cases are possible. In the first case the TI-Server thread has served all load and there is nothing left to schedule. In this case the TI-Server will send a signal to the Server-TTL. This signal will interrupt the sleep function and a "finished period" event will be registered early. The second special case is if the idle time is up and the TI-Server is activated even though no load is available. In this case the TI-Server waits on a task to schedule. The TTL is started at the moment the first job arrives for the TI-Server. By handling these two cases in this way we maximize the time the server is able to schedule its load while at the same time still guaranteeing and idle time of t_c.

To complete the implementation necessary for evaluation, it must be possible to run one TI-Server per core. This will be realized by using the multi-core ability of HSF. One TI-Server is created per core during initialization see 4.5. This means that in the 4 testing core platform a maximum 4 TI-Servers, one for each core are running. In the evaluation experiments in 6 only 2 TI-Servers are needed, one core is reserved for the HI-Scheduler and one core is used for temperature measurement.

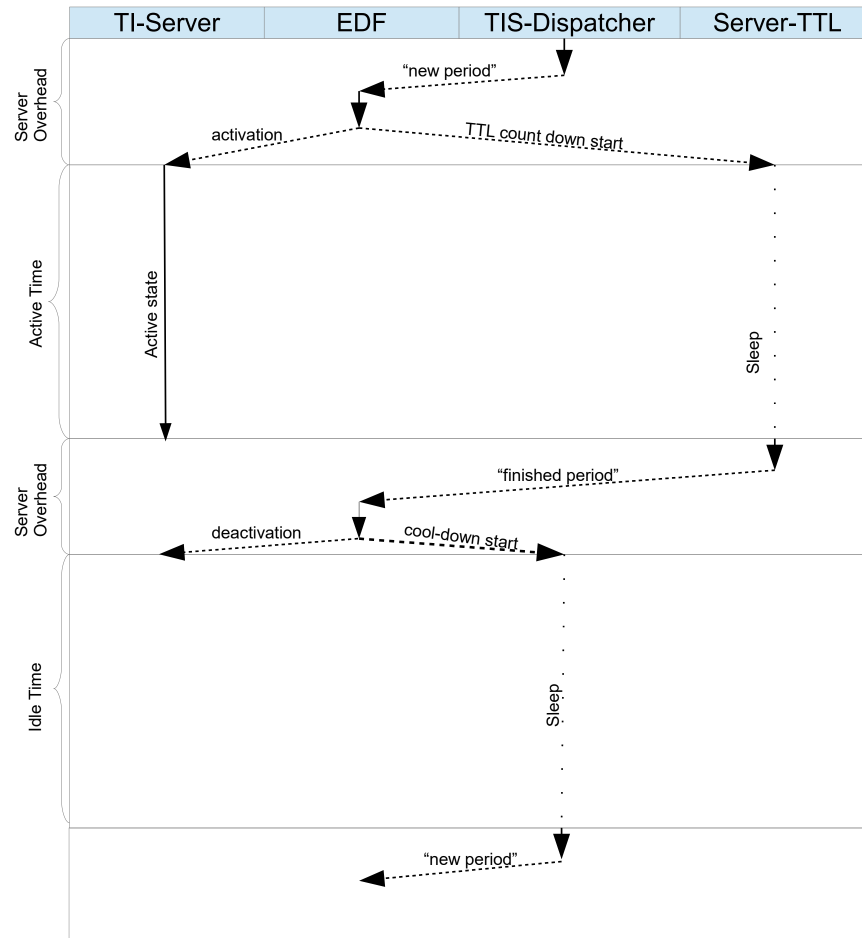

Figure 3.3: Illustration of the TI-Server mechanism

3.2 HSF Extensions

In this sections the extensions to the “Hierarchical Scheduling Framework” are described. First the thread priorities of all used threads are presented. Then the new monitoring functionalities, temperature measurement, jitter measurement as well as trace recording are described.

3.2.1 Thread priorities

HSF consists of a number of different threads. These threads receive different priorities such that they are scheduled correctly by the operating system. To prevent the operating system from preempting HSF these priorities have to be as high as possible. The highest priority thread is set to priority 98, see table 4.4. The highest priority thread is the main thread, it initializes and terminates the test-run. The next highest threads are the schedulers. HSF is built on a hierarchy of schedulers and thus each hierarchy level has its own priority. Below the scheduler levels are the dispatchers. The dispatchers are threads that create new events that are then scheduled by schedulers or servers. Because of that, dispatchers have to have a higher priority than workers. Active servers have the priority level directly below that of the dispatchers. Servers are a combination of workers and schedulers in the sense that they are activated through a dispatcher and during their active time they act like a scheduler. Because they are activated by dispatchers their priority has to be lower than that of the dispatchers. As they schedule workers, their priority level has to be higher than that of the workers. Next is the priority level of active workers. Below that remain the priority of the idle thread and finally the priority of all deactivated threads, the inactive priority. As a note, the idle thread cannot be used in the TI-Server implementation as we need timeslots in which the CPU core is completely idle. Thus the idle priority is not needed for the evaluation framework. In the original HSF idle threads blocked the inactive threads from executing, in the TI-server implementation this has been replaced by the use of semaphores and signals, see 4.6.

3.2.2 Temperature monitoring

Thermal Isolation Server try to give thermal guarantees on the system. To evaluate its effectiveness it is essential to be able to monitor the temperature of each individual core of the test platform. The temperatures are read out from model specific registers (msr). The resolution of these values is of 1 degree Celsius. The temperature monitoring data is also needed as calibration data for thermal modelling of the test platform (see 6.1)

Priority Level	**Thread Type**
98 Highest Priority	main thread
98-1-Level =	Scheduler
98-1-maxLevel-1 = disp_prio	Dispatcher
disp_prio -1	Server
disp_prio -2	active worker
disp_prio -3	idle thread
disp_prio -4	inactive worker server or scheduler

Table 3.1: Thread Priorities

3.2.3 Jitter measurement

In 3.3 the importance of small latencies in real-time applications is explained. Furthermore the steps taken in order to decrease these latencies on the test platform are described. To check the success of these measures a jitter measurement system has been implemented. This system will measure the time between the activation of a worker and the actual execution start of that worker.In our implementation the jitter can be measured under certain settings (see 4.2).

3.2.4 Traces and overhead calculation

For the evaluation of the TI-Server it is important to verify the timing guarantees that are given by the server. Therefore a number of traces have to be recorded. Activity traces for schedulers, servers and workers have to be recorded. For all three thread type, it is necessary to record the activation and deactivation time. The execution-start and -end of the threads are also recorded. From these traces scheduling overheads are calculated and timing constraints can be verified.

3.3 Enhancing real time capabilities

To ensure that the testing platform is real-time capable we have to ensure that the jitter between the scheduled and actual start time is as small as possible. To achieve this two main measures are taken:
The first measure is to install the Preempt_rt patch for the Linux kernel (see 3.3.1).
The second measure is to implement memory page locking in HSF. (see 3.3.2)
In addition to these two measures, hyper-threading and power management are disabled in the BIOS of the system. These two options may increase latencies and are therefore unsuited for real time applications.

3.3.1 Preempt_rt

In real-time applications the latency between triggering an event and the actual start time should be minimized. To enable this, a real-time operating system (RTOS) is used. Such an operating system enables the calculation of latencies. In general the Linux kernel is non-real-time. This implies that high priority tasks can be delayed by an unknown amount of time due to preemptions. Such behaviour is unsuitable for real-time systems.

In the Preempt_rt patch several changes to the kernel are made in order to minimize the latencies. Here are the main changes:

- The in-kernel locking primitives have been modified
- Critical sections are now preemptible
- A priority inversion for in-kernel semaphores and spinlocks is implemented
- All interrupt handlers are preemptible

For further information on the Preempt_rt patch see [9]. The above mentioned changes convert the kernel into a fully preemptible kernel. This enables us to define high priority tasks which will not be preempted by the kernel and will therefore have low latencies.

3.3.2 Page locking

Page faults come in two different forms: minor and major page faults. The minor page faults are pages that are already in RAM, but they are not marked by the Memory Management Unit (MMU). The page fault handler of the operating system only needs to make an entry for that page in the MMU, but it does not need to load the page from external memory. This type of fault causes relatively small latency increase. The second type of page faults, the major faults, are more costly in terms of latency. In this case the page is not yet present in RAM and needs to be loaded by the operating system first. This adds disk latency and makes this type of fault is far more costly than the previous one.

In real-time applications page faults have to be avoided to minimize latencies. Especially major faults have to be avoided. Locking of pages in RAM can be done by using mlockall(MCL_CURRENT — MCL_FUTURE).
After issuing this command all future and current pages will be locked in RAM. In our implementation this command is issued right at the start of the main function. From this point on, all thread stacks are locked in RAM. The default stack size for a thread is 8MB.

Chapter 4

Implementation

In this chapter the implementation of the different extensions to HSF are explained. At first the added monitoring functionalities are described in detail.These are temperature and jitter measurement as well as page fault analysis. In a second part the implementation of the TI-Server is presented. For guides on installation and use of the HSF software please consult the appendix A.

4.1 Temperature measurement

Temperature measurement can be done with two methods, either through a measurement task or through a inclass function . In both cases the temperature is read from model specific registers (msr) using the rdmsr() system call. Each temperature measurement records the temperatures of all cores by reading one register per core. To be able to read these registers, the msr kernel module has to be loaded before the test-run with the "sudo modprobe msr" command (see [6]).

4.1.1 Temperature measuring task

The temperature measuring task measures the temperature of every core and saves the results in the corresponding records (see 4.1.3). The temperature measuring task is attached to a worker and can then be scheduled by servers or schedulers. As the measurement is called by a worker, its period can be defined, allowing us to set the measuring frequency.

4.1.2 Temperature measuring mechanism in Resource Allocator

The second temperature monitoring possibility is built into the ResourceAllocator class. It consists of the *getCoreTemperature()* function. This function is equivalent in functionality to the temperature measuring task.

This function allows us to measure the temperature at every call of a scheduler or Server. If lower sampling and inconsistent rates are sufficient, then this method can be used. If desired both methods can be used in combination. In the current implementation the temperature is not monitored by resource allocators, as the temperature measurement task was sufficient.

4.1.3 Temperature records

For every temperature measurement a new entry to the corresponding core statistics is created. This entry consists of the temperature measurement, time at which the measurement occurred, and core the temperature is from. After the scheduling run is complete, all temperature entries are saved into a csv file called "simultaion_name_temperatures.csv". In this file the entries are ordered by core and time.

4.2 Jitter measurement

As described in 3.3 the jitter or latency between an event and the response to this event is of great concern in real-time systems. For this reason a number of steps have been taken as described in 3.3. As the jitter is not a direct part of the TI-Server it has not been explained in that implementation. The only goal of the jitter measurements is to show the impact of the different steps taken in 3.3. Due to time constraints the jitter measurement has only been implemented between Event Based Schedulers and Workers. The result of the measurements can be found in chapter 6. The following text gives a detailed description of the measurement mechanism and storage of those measurements.

4.2.1 Activation time and firing time

To measure the latency between the activation (an event occurs) and the actual execution of the task (response to the event) two time stamps are measured. The first time stamp is the activation-time, it is taken at the instant at which a worker is activated. The time stamp is then saved into a buffer. This is handled by the *save_activation_time()* function from ResourceAllocator. The time stamp is taken with *getTime()* function from TimeUtil. This function takes the time from the operating system clock. On the other end the worker saves a time stamp at the moment it starts executing, the firing-time. This time stamp is saved locally until the worker is deactivated by the scheduler. The jitter is calculated as firing-time − activation-time. This final step is handled by the *calculate_jitter()* function from ResourceAllocator.
As every scheduler and server type has a different scheduling function, the placement of the two jitter measurement specific functions *save_activation_time()* and

calculate_jitter() is different for each one. In this project it has only been included into all Event Based Schedulers, see B.1.

4.2.2 Jitter records

For every jitter measurement an entry to a record file is saved. Every entry consists of the worker, the activation and firing time and finally the jitter. After the test-run all these entries are written into a csv file called "simulation_name_jitter.csv". From all the recorded jitter measurements, mean, maximum and minimum values are calculated and put into a csv file called "simulation_name_jitter_stats.csv".

4.3 Page fault analysis

As described in 3.3.2 page locking is used in order to reduce the number of page faults. In order to evaluate the success of page locking a page fault analysis mechanism has been implemented.
The *show_new_pagefault_count()* was introduced in *Thread* class, it counts the number of page faults and prints the result out. The *getrusage()* function [3] is used to count the minor and major page faults of the current thread. If the macro _PAGE_FAULT_CHECK is 1 the page fault check will be called in *wrapper()* once at the creation of the thread and once after reading the whole stack. If the page locking was successful, then page faults should only occur at the creation of the thread. When reading the stack after creation, no page fault should occur.

4.4 Trace recording and overhead calculation

For the evaluation of the TI-Server, traces of the schedulers, servers and workers are recorded. A trace is a suite of records that consists of a thread ID,action type and the time-stamp of that action. For the different thread types the following actions are recorded:

- *For the worker* the recording of traces was already implemented into HSF. In this implementation the activation and deactivation times are recorded as well as the start and end of the task. This means that a time-stamp is recorded when the worker starts executing and when it has finished executing its task. For workers in the TI-Server, this is not sufficient as we need to know when the worker stops executing its task and when it retakes the execution. For this, two new action types are introduced, the execution start and the execution stop.

- *The TI-Server* is activated and deactivated by a scheduler. Therefore, a trace for each of these events has to be recorded. Furthermore a trace for execution start and stop of the server are recorded. Finally a trace has to be recorded when a new server period is spawned by the TISDispatcher.

- *For the EDF scheduler* the same traces as for the TI-Server are recorded. As there is no dispatcher for schedulers no such trace is recorded.

After the test-run all recorded traces are saved into csv files. For each recorded thread type a corresponding file is created. One file containing all traces is also created. For each Server period the start and endpoint are calculated as well as the overall period duration. In a period, the server execution time and the worker execution time are calculated. All these values are saved in a csv file named *"simulation_name_serverOverhead.csv"*
A list of all the different recorded events and their assigned action value are listed in the following table.

action type	**value**
job arrival	0
activation	1
deactivation	2
task start (for worker) scheduling start (for scheduler and server)	3
task end (for worker) scheduling end (for scheduler and server)	4
deadline met (worker)	5
deadline missed (worker)	6
worker execution start	10
worker execution stop	11

Table 4.1: Trace action types

4.5 Pre-partitioned EDF

In the evaluation framework, multiple TI-Servers have to run in parallel on different cores. Pre-partitioned EDF is a new scheduling class that has the goal to create an EDF scheduler on every core. Those EDF schedulers will then schedule the TI-Servers. The Pre-partitioned EDF gets a set of workers and servers from the parser. To each worker and server a core has been assigned in the definition of the test-run (in the XML file). The workers and servers are then assigned to the corresponding scheduler on the respective core. The pre-partitioned EDF has to be used as level 0 scheduler and is only used to activate and to deactivate the underlying EDFs.

4.6 TI_server

In the following the implementation of the design presented in 3.1 is explained. For this the TI-Server implementation is first described step by step in 4.6.1 with the help of figure 4.1. In this project signals have been introduced into the implementation of HSF. These are needed to ensure an idle core during t_c. A detailed description of the use of these signals is given in **??**

4.6.1 Implementation

The implementation of the TI-Server is described with the help of figure 4.1 that can be found below. In this figure the various steps of the TI-Server mechanism are numbered. The implementation of each step is then explained in the enumeration. For this reference is made to 3 classes and their functions that have been added to HSF. These classes are the *TIS*-class for the implementation of the TI-Server, the *TISDispatcher*-class for the implementation of TIS-Dispatcher and the *Server-TTL*-class for the Server-TTL implementation. A detailed description of all 3 classes can be found in appendix B.

At the start-point the TI-Server is in deactivated state. This means that it is blocked from execution by a “activation” semaphore from class TIS. The Server-TTL is also blocked by a semaphore, the “activation” semaphore from the Server-TTL class. The EDF scheduler is blocked by its “event” semaphore. This means that the EDF will wake up as soon as an event is registered. The TIS-Dispatcher is executing on the core at the start point:

1. The TIS-Dispatcher registers a “new job” at the EDF scheduler regarding the TI-Server. This is done by calling the the newPeriod() function of the TI-Server class. This call also releases the “event” semaphore [EDF-class] of the EDF scheduler. The TIS-Dispatcher has now run through its execution loop and it then loops back. There it falls onto a blocked “activation” semaphore [TIS-Dispatcher-class] .

2. As an event has been registered and the “event” semaphore [EDF-class] is released the EDF scheduler starts executing. As the TI-Server is the only registered thread it is immediately activated. For this the activate() function of the TIS-class is called. Here the priority of the TI-Server will be changed to the Server-Priority and the TI-Servers “activation” semaphore is released.

3. This step is the initiation of the Server-TTL count-down. For this the activation() function of the Server-TTL class is called. This function sets the TTL and releases the “activation” semaphore [Server-TTL-class] of the Server-TTL. If the TI-Server has a task that is to be scheduled, the activation of the Server-TTL is done at the same time as the activation of

the TI-Server. However if the Server has no load at the beginning of its active time , the Server-TTL will only be activated once the first job for the TI-Server arrives. The Server-TTL has a dispatcher priority. It therefore has the highest priority of all non blocked threads, and starts executing. It then immediately enters a sleep function for TTL, during this time it is suspended from execution.

4. After being activated by the EDF scheduler the TI-Server starts scheduling its Server load. Inside the the server, an EDF scheduler, the "server EDF" is scheduling the workers that have registered jobs. These jobs are registered by the corresponding dispatcher of the worker. A worker is schduled by activating it. For this the activation function of the worker class is called in which the "activation" semaphore [Worker-class] is released and the priority of the worker is changed to active. Before a worker is activated, the currently active worker has to be deactivated. This is done by calling the deactivation function of the worker. This function will set the priority of the worker to "inactive" and it will block the "activation" semaphore [Worker-class]. During the active time context switches between workers will be done because of their assigned priority as in original HSF. The "activation" semaphore [Worker-class] is of no importance during this time, as the worker does not wait for on it during executing its task. This has to be changed for idle time, see point 5 and

5. The Server-TTL wakes-up after TTL. It then calls the finishedPeriod() function of the TIS-class. This function registers a "finished job" and releases the EDF's "event" semaphore [EDF-class]. By this, the Server-TTL initiates the deactivation of the TI-Server. As mentioned previously, the Server-TTL will wake up after TTL is over. If the TI-Server has no load left while the TTL has not yet passed, the TI-Server will send a POSIX signal to the Server-TTL. This causes the sleep function to be terminated early and the TI-Server is deactivated early. A second purpose that the finishedPeriod() call is serving is to send a POSIX signal([10],[7],[8]) to all workers that have been used during the active time of the server. After receiving a signal the worker, will call its signal handler. This signal handler will acquire the "activation" semaphore [Worker class]. This will block the execution of the worker until the semaphore is released in the next active time of the server. For more details on how a idle time is guaranteed see

6. Again an event is registered for the EDF scheduler. It is immediately handled. This time the event is a "finished job" event for the TI-Server. The EDF scheduler therefore calls the deactivate() function of the TIS-class in order to deactivate the TI-Server.

7. In the call of the deactivate() function from TIS-class, the TIS-Dispatcher is activated. For this the activate function of the "TISDispatcher" class

is called. This releases the “activation” semaphore of the TIS-Dispatcher. Similar to the Server-TTL, the TIS-Dispatcher will then start execution. Its first instruction is a sleep function that will suspend the thread for the idle time.

8. After the idle time the TIS-Dispatcher will wake up by returning from the sleep function. The procedure will start all over again.

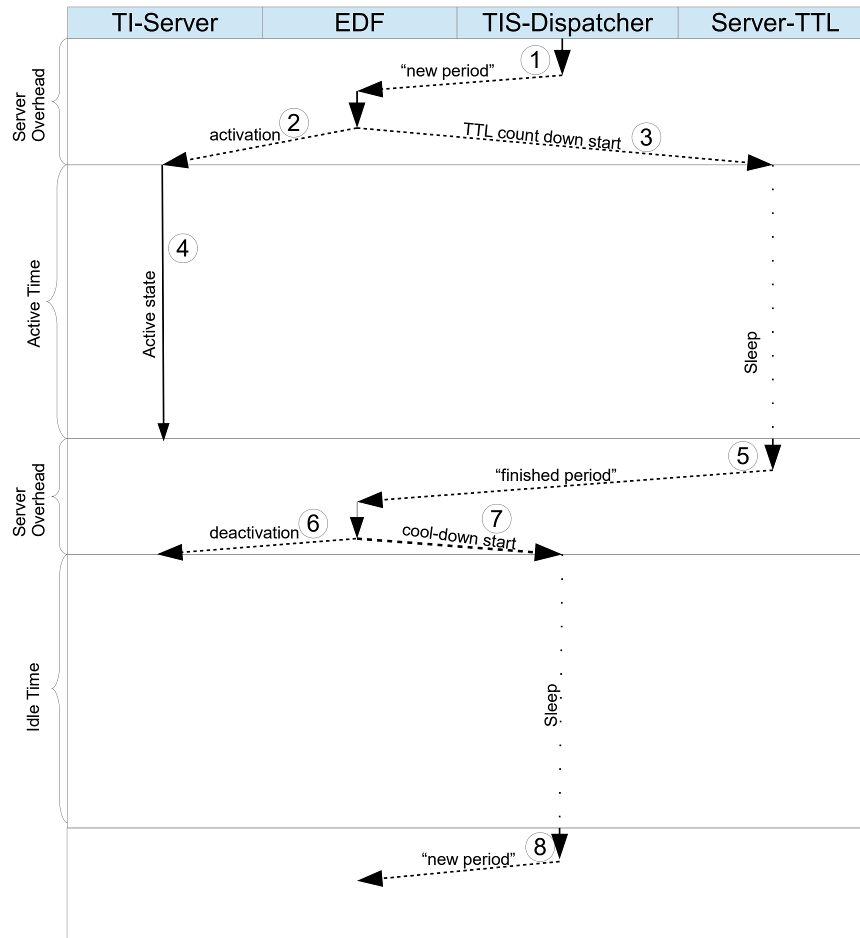

Figure 4.1: Illustration of the TI-Server mechanism

4.7 Modifications to the HSF parser

In HSF, a test-run is created by the creation of a XML file. This XML file is then parsed by HSF and all threads and their configurations are determined. In this project a number of new thread types and their configurations have been added. XML files are still used as set-up medium for test-runs. Therefore all new configurations have been added into the parser. A detailed description on how to create a TI-Server test-run XML is given in Appendix C.

Chapter 5
Verification of the implementation

This chapter describes all tests that were performed to ensure the correct functionality of all implementations. The first tests are jitter measurements that verify the effectiveness of the Preempt_rt patch, see 3.3.1. The second set of tests are tests that are performed in order to verify the implementation of the TI-Server

5.1 Jitter measurement

In this section we test the effectiveness of the measures from 3.3. On each core a busy wait task is executed periodically every 5 ms with WCET of 4ms. These tasks are scheduled by an EDF scheduler on their respective core. The test run has a duration of 60 seconds.
The first set of tests is done in the non-patched Linux kernel. The maximum measured jitter when the test run is executed at a low priority is about 40us. When the priority is set to max (e.g 98 for the main thread) the maximum measured jitter is around 20us.
In the second test set we use the rt-patched kernel. Here the maximum latency measured for low priority is still 40us however for maximum priority the max latency drops now below 10us.

We can thus see that the priority has a direct impact on the max latency in both cases. This is due to the fact that while running on low priority the threads of HSF can be preempted by everything that has higher priority. This holds true for both kernels. When running with high priority the rt-patch does affect the maximum latency.

5.2 TI-Server implementation verification

Before evaluating TI-Servers performance it is necessary to make sure that the implementation is working correctly. For this a number of different TI-Server configurations are tested. The first setup aims to verify the EDF scheduling of the server. The second series of tests verifies the parallel execution several servers. At the same time the server behaviour can be verified under certain marginal conditions: How the server behaves if the load is smaller than t_a and how it behaves if the load is not empty at the end of active time. This test also illustrates what happens when the server is activated but there is no load.

Verification of EDF in TI-Server:

Test setup:

- 1 core used : core 0
- 1 TI-Server, S_0, with t_a = length of test run and $t_c = 0$
- workers as load of TI-Server S_0:
  - *worker 1*: task "busy_wait" , Period $P = 10ms$, $WCET = 5ms$
  - *worker 2*: task "busy_wait" , Period $P = 20ms$, $WCET = 5ms$

Expected behaviour:
According to the EDF schedule *worker 1* will always execute before *worker 2*.

Results:
In figure 5.1 the results are illustrated. *Task 1* is always prioritized over *task 2*. *Task 2* is not able to finish executing in one instance because of the overhead caused by the server. This was to be expected. The results show that the TI-Server is indeed scheduling the server load according to an EDF schedule.

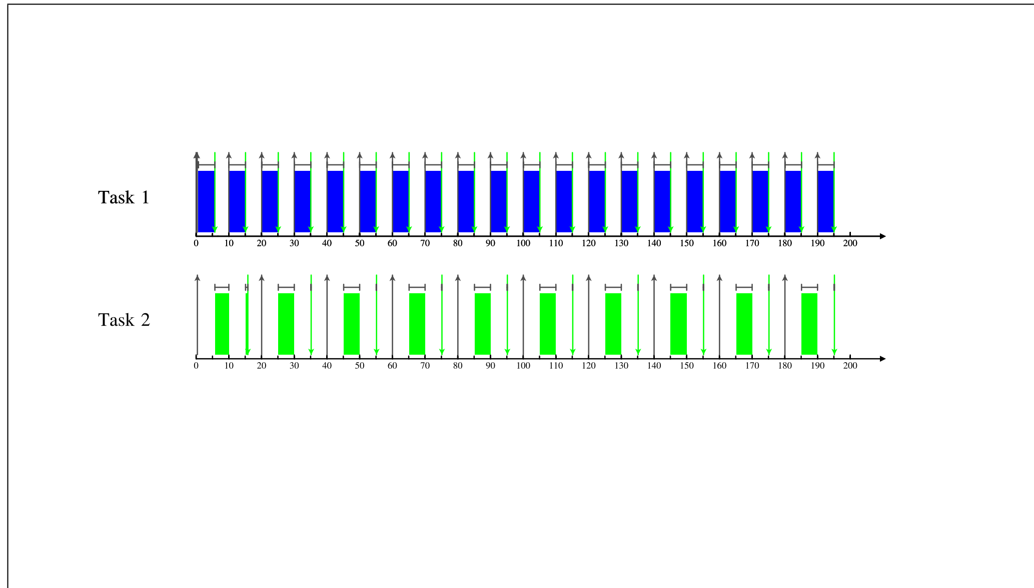

Figure 5.1: EDF verification test for TI-Server

Multiple TI-Servers test:

Test setup:

- 4 cores are used
- HI-Scheduler is an EDF scheduler that is running on core 0 ,*worker 1* and *worker 2* are scheduled by the HI-scheduler
- LO-Scheduler consists of 3 TI-Servers:
  - S_1 running on core 1, with $t_a = 5ms$ and $t_c = 5ms$
  - S_2 running on core 2, with $t_a = 7ms$ and $t_c = 3ms$
  - S_3 running on core 3, with $t_a = 4ms$ and $t_c = 6ms$
- 5 workers:
  - *worker 1*:
    - running on core 0 under HI-Scheduler
    - “temperature measurement” task
    - $Period = 1ms$
  - *worker 2*:
    - running on core 0 under HI-Scheduler
    - “busy_wait” task

    * $Period = 10ms$ and $WCET = 5ms$

- *worker 3*:
    * running on core 1 under S_1
    * "busy_wait" task
    * $Period = 25ms$ and $WCET = 10ms$
- *worker 4*:
    * running on core 2 under S_2
    * "busy_wait" task
    * $Period = 10ms$ and $WCET = 6ms$
- *worker 5*:
    * running on core 3 under S_3
    * "busy_wait" task
    * $Period = 20ms$ and $WCET = 5ms$

Expected behaviour:

- *core 0*: *worker 1* and *worker 2* should be scheduled according to an EDF schedule.
- *core 1*: As the period of *worker 3* is double t_a of S_1 it will take at least $2 * t_a$ to finish the job of *worker 3*
- *core 2*: The period of *worker 4* is smaller than t_a of S_2. One t_a should be sufficient to finish the execution of *worker 4*, the execution should not be interrupted.
- *core 3*: In this case the period of *worker 5* is larger than t_a of S_3 but it is smaller than $2 * t_a$. The execution of task 5 will thus take two active times of the TI-Server. The second active time should be smaller than t_a because the server will run out of load before t_a has expired.

Results:

In figure 5.2 the results are displayed. In table 5.2, 5.2 and 5.2 statistics of the first 6 periods of the 3 TI-Servers are listed. From this figure and tables it can be seen that:

- *On core 0:* the HI-Scheduler is executing *task 1 and 2* according to an EDF schedule. Task *1* has a smaller period an is thus preempting *task 2*. The HI-Scheduler is working correctly.

- *On core 1:* The *worker 3* takes 3 active times of S_1. The first 2 active times have a duration of t_a and the third active time is very short. This can be observed in 5.2. As WCET of the "busy_wait" task of *worker 3* is exactly double t_a of S_1 and because of server overhead, the execution of *worker 3* is not finished after $P + t_a$. This behaviour is unavoidable as there is always some overhead.

- *On core 2:* *worker 4* has a smaller WCET as the t_a of S_2. It therefore finishes executing in one active time of S_2. In Figure 5.2 it can be observed that the core 2 is some times idle for more than t_c, for example after the second iteration of *worker 4*. In Table 5.2 the time between the end of period 2 and start of period 3 is approx t_c however. In this case the active time of the server begins, but there is no load for the server to schedule. In this case the server waits in active mode until an event happens without consuming ttl. This "wait time" is shown in the last column of table 5.2.

- *On core 3:* As *worker 5* has $t_a < WCET < 2 * t_a$, every second active time of the server is smaller than t_a. Same as for S_2, S_3 is some times activated without load. Again the server waits in active mode.

In table 5.2, 5.2 and 5.2 it can be observed that the first period of each server starts witch a delay of 1ms. This delay is due to initialization and should not influence the evaluation process. In this test we have shown that the TI-Server implementation is able to work on the multi-core platform. Activation and deactivation of the server also work as expected.

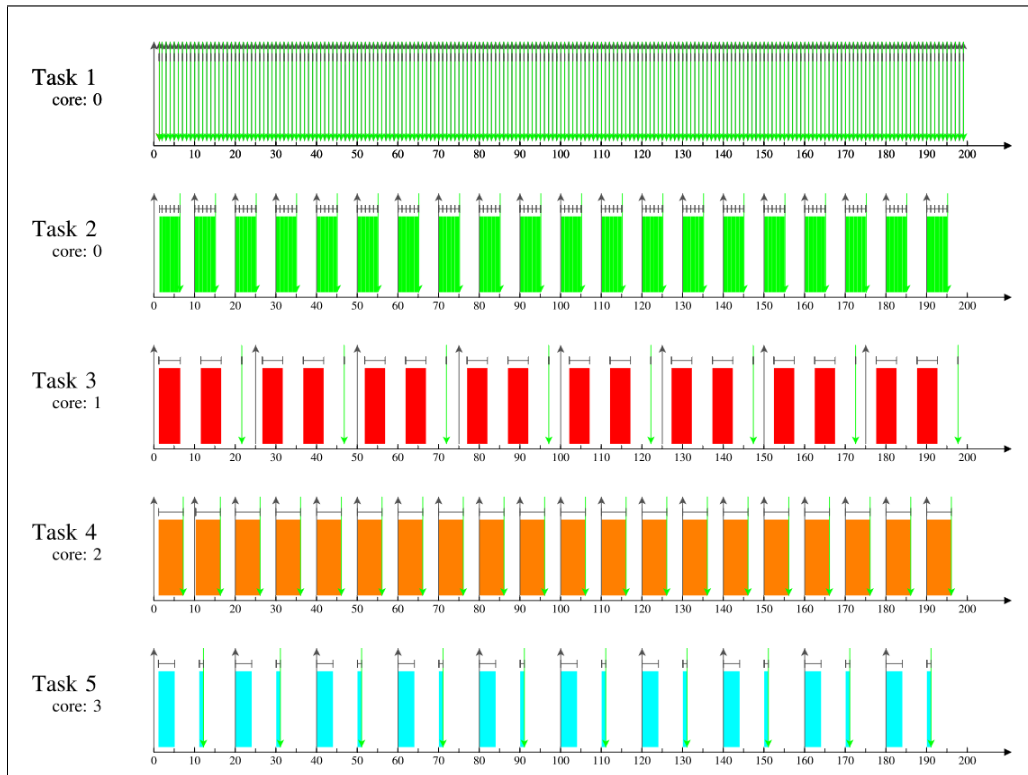

Figure 5.2: TI-Server test on multi-core

Server	active time Start (us)	active time End (us)	active time(us)	worker time (us)	active wait time (us)	Overhead (us)
S_1	1202	6502	5300	5290	0	18
S_1	11534	16560	5026	5005	0	20
S_1	21596	21639	43	2	0	40
S_1	26677	31694	5016	5003	0	13
S_1	36734	41750	5015	5004	0	11
S_1	46789	46814	25	1	0	24

Table 5.1: First 6 server active times for server on core 1

Server	active time Start (us)	active time End (us)	active time(us)	worker time (us)	active wait time (us)	Overhead (us)
S_2	1058	7252	6194	6059	0	134
S_2	10269	16377	6107	6060	0	47
S_2	19417	26143	6725	6069	601	53
S_2	29159	36093	6934	6059	843	30
S_2	39108	46111	7003	6062	911	29
S_2	49129	56117	6988	6059	892	35

Table 5.2: First 6 server active times for server on core 2

Server	active time Start (us)	active time End (us)	active time(us)	worker time (us)	active wait time (us)	Overhead (us)
S_3	1105	5134	4029	4007	0	21
S_3	11155	12214	1053	1012	0	45
S_3	18236	24065	5828	4000	1788	40
S_3	30079	31120	1040	1011	0	29
S_3	37134	44039	6905	4004	2885	14
S_3	50054	51100	1045	1011	0	34

Table 5.3: First 6 server active times for server on core 3

Chapter 6

TI-Server evaluation experiments

In this chapter the different experiments are described, that were performed to evaluate TI-Servers. The first set of experiments are thermal calibration experiments. These experiments are necessary to calibrate the thermal model of the testing platform. The second set of experiments are the evaluation experiments of the TI-Server. These experiments aim to experimentally validate the TI-Server theory.

6.1 Thermal modelling

TI-Servers are created in respect to the thermal model of the CPU they are running on. It is thus imperative to determine the thermal model of the platform before being able to test TI-Servers on that platform. The thermal modelling itself is not part of this project and it is done by the supervisor Dr. Ahmed. In the following the different tests needed for thermal modelling are described and the results are presented.
First we need to make sure that all tests are done under the same thermal environmental conditions. Two measures are taken:

- The *fan speed* is set to a fixed frequency. The fan speed can be set with the thinkfan tool [11],[4], as the test platform is a Lenovo Thinkpad. The thinkfan tool allows 7 different fan speed levels. For all experiments level 7 is used.
- The *CPU frequency* has to be the same for all experiments. The CPU frequency is fixed by setting the "performance" CPU governor for all cores. This is done with the help of the "cpuset-freq" tool [2],[5].

In total 22 different tests are executed. After each test a cool-down of at least 5s is respected. In each test a thermal monitoring task is running on core 0. All

dispatchers and schedulers are running on core 0 as well. On the other cores "busywait" tasks are executed if the core is active. The tests can be divided into 8 subsets:

Set	**nbr of active cores**	**active time**	**idle time**	**temperature measurement period**
1	1	0.5 ms	5 s	100 us
2	1	10 ms	5 s	100 us
3	1	100 ms	5 s	1 ms
4	1	5 s	5 s	1 ms
5	2	10 ms	5 s	100 us
6	1	5 min	5 min	500 ms
7	2	5 min	5 min	500 ms
8	3	5 min	5 min	500 ms

Table 6.1: Thermal Calibration experiments

For the thermal modelling it is preferable that during idle time no tasks are executed. But we cannot guarantee that the operating system will not execute tasks outside of the evaluation framework during these idle times. For this reason tests from sets 1-5 are repeated multiple times in order to get one instance where the operating system does not interfere.

6.2 Thermal isolation server experiments

For the evaluation of the TI-Server a flight management system (FMS) is simulated. In this system a number of tasks are executed, see list 6.2. These tasks are divided into 2 criticality levels. Tasks having criticality 1 are scheduled by the "HI-Scheduler" and tasks with criticality level 2 are scheduled by the "LO-scheduler". In the following experiments, all tasks that are executed are not the actual FMS tasks, but they are "busy-wait" tasks that have corresponding periods and WCETs. We set a maximum system temperature of 70 degrees. This temperature should not be surpassed by any core during the experiments. Two different experiments (each having a duration of 1h to reach steady state temperatures) are executed in order to evaluate the performance of the TI-Server. The first experiment partitions the LO critical task according to first-fit on core 2 and 3. The tasks are then scheduled by a plain EDF scheduler. In the second experiment the LO critical tasks are partitioned and scheduled according to the TI-Server theory:

Purpose	Task	CL	Period (ms)	Exec.Time (ms)
Sensor data acquisition	τ_1	2	200	10
	τ_2	2	200	10
	τ_3	2	200	10
	τ_4	2	200	10
	τ_5	2	200	10
Localization	τ_6	2	200	10
	τ_7	2	1000	50
	τ_8	2	5000	50
	τ_9	2	1000	50
	τ_{10}	2	200	10
	τ_{11}	2	1000	50
	τ_{12}	2	200	10
Flightplan management	τ_{13}	2	1000	50
	τ_{14}	1	1000	50
	τ_{15}	2	1000	50
	τ_{16}	1	1000	50
	τ_{17}	2	1000	50
	τ_{18}	2	1000	50
	τ_{19}	1	1000	50
	τ_{20}	1	1000	50
Flightplan computation	τ_{21}	2	1000	50
	τ_{21}	2	1000	50
	τ_{23}	2	5000	750
	τ_{23a}	2	5000	180
	τ_{23b}	2	5000	150
	τ_{23c}	2	5000	90
	τ_{23d}	2	5000	75
Guidance	τ_{24}	2	200	10
	τ_{25}	2	200	10
Nearest Airport	τ_{26}	1	1000	50

Table 6.2: FMS Parameters

First-Fit

Test setup:

- core 0 : Temperature measurement
- core 1 : EDF with HI tasks
- core 2 and 3: First Fit partitioning of low critical tasks. The low critical tasks are scheduled with EDF on their respective core

Results:

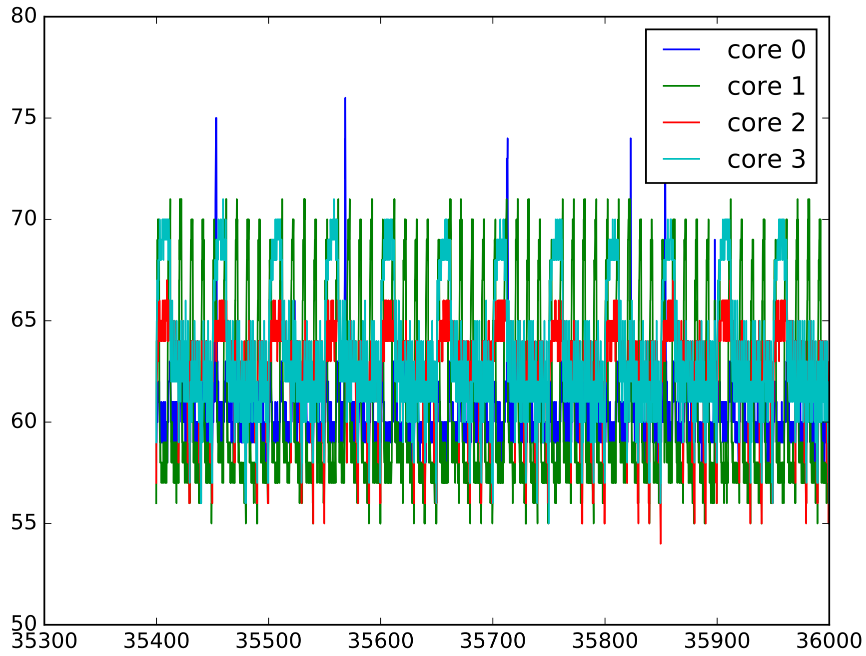

Figure 6.1: Temperature trace for the last 60 s of the first fit experiment

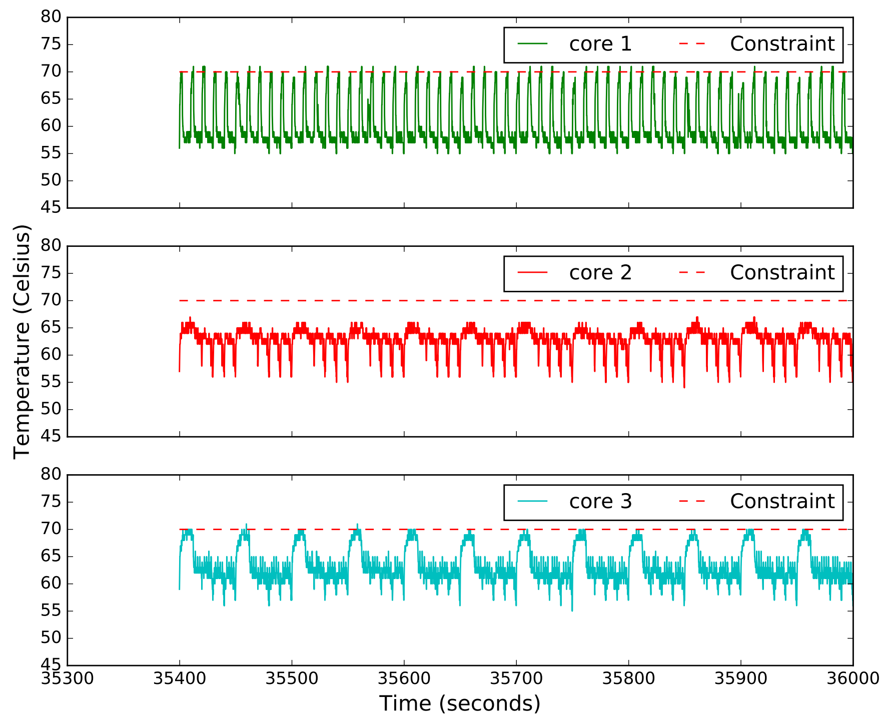

Figure 6.2: Temperature trace for the last 60 s of the first fit experiment of core 1,2 and 3 separately. The “constraint” line denotes the maximum temperature that should not be surpassed.

TI-Server

Test setup:

- core 0 : Temperature measurement
- core 1 : EDF with HI tasks
- core 2 : TI-Server with TTL = 7300us and cool-down = 2700us
- core 3 : TI-Server with TTL = 5 ms and cool-down = 5 ms
- the low critical tasks are partitioned on core 2 and 3 according to the thermal model of the testing platform. They are then schduled by the respective TI-Server

Results:

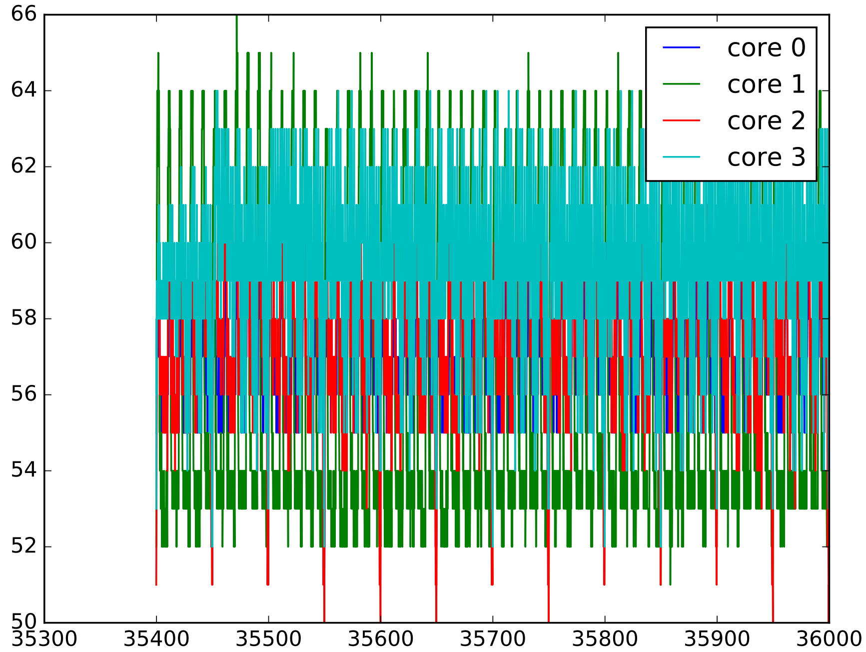

Figure 6.3: Temperature trace for the last 60 s of the experiment using TI-Servers

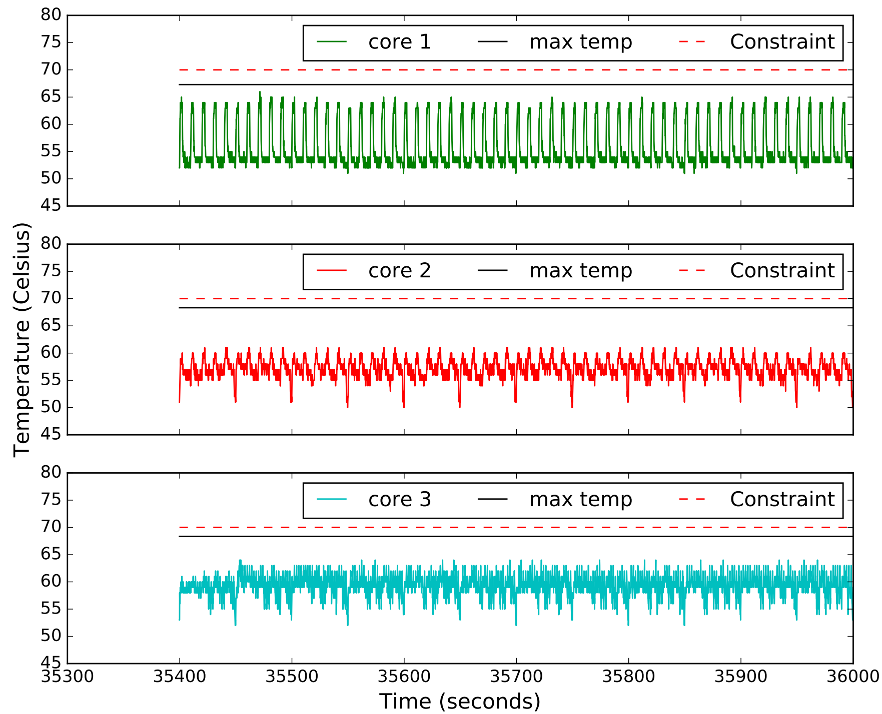

Figure 6.4: Temperature trace for the last 60 s of the experiment using TI-Servers of core 1,2 and 3 separately. The "max temp" line denotes the upper bound given by the theoretical model. The "constraint" line denotes the maximum temperature that should not be surpassed.

In both tests the same tasks are executed. In the experiment that is using TI-Servers the maximum temperature of the CPU cores is about 6 degrees cooler than the temperatures in the experiment without TI-Servers. An immediate effect on the temperature trace can also be observed in the trace of core 3. In the first experiment the temperature of core 3 has large periodic fluctuations. These are due to the EDF scheduler which is executing tasks as soon as they are available. In the second experiment the temperature fluctuations are significantly reduced for core 3. This is because the TI-Server distributes the execution times more evenly. We could also show that the maximum temperature that was given by the theoretical model (the “max temp” line in figure 6.4) was not surpassed by using TI-Servers. We also show that in the first experiment the core 1 and 3 surpass the temperature constraint of 70 degrees. In the second experiment, however, the temperature stays well below this threshold. We thus show that the temperature guarantees of the TI-Server model are met. As no deadlines are missed and all tasks are scheduled correctly we can also say that all timing guarantees of the TI-Server are met.

Chapter 7

Conclusion and outlook

7.1 Conclusion

In this thesis, the TI-Server scheduling scheme was implemented on a 4 core platform. The implementation was integrated into the "Hierarchical Scheduling Framework". The implementation of the TI-Server was done in 3 main parts, the TI-Server itself, the TIS-Dispatcher and the Server-TTL. The server itself, schedules its tasks according to an EDF schedule. The dispatcher ensures that after each active time a minimum idle time is respected on the core. The Sever_TTL ensures that the server is deactivated after a maximum active time. For a complete implementation of the TI-Server scheme a configuration was created, that can run one TI-Server per available core. Here the multi-core capability of HSF was used. In order to show the correctness and completeness of the TI-Server implementation, verification tests were performed. In addition to the implementation, the TI-Server was evaluated for thermal and timing guarantees. For this purpose HSF was extended by a thermal monitoring mechanisms. This mechanisms measure the temperature of each core in configurable intervals. Furthermore trace recording of the state of schedulers, servers and workers were added. The evaluation platform is made real-time capable through the preempt-RT patch for Linux and through memory page locking in HSF. Finally the TI-Server has then been evaluated. During these experiments the core temperatures stayed well below the theoretically calculated upper bound and therefore also under the critical CPU temperature. In comparison a first-fit partitioning and EDF schedule could not meet these constraints. This has shown the effectiveness of the TI-Server concept.

7.2 Outlook

The effectiveness of the TI-Server has only been shown for a set of tasks where all tasks are executing the same code, a busy wait loop. Further testing could

be done to analyse the effect of different task sets. The next step would be to test the TI-Server on a real mixed-criticality system.

Appendix A

Installation guide

Below find an installation guide for HSF

Listing A.1: InstallHSF

```
Hierarchical Scheduling Framework {#mainpage}
=============================================

Recommendations
---------------


To ensure real time compatebility:
 * install a preemt_rt patched linux kernel.
   This patch will make the kernel fully preemtable and it will
       minimize latencies.
   For more information :
   https://rt.wiki.kernel.org/index.php/Main_Page
   For quick installation guide: preemt_rt-patch-installation-
       guide.txt
 * In the BIOS disable the POWER_MENAGEMENT and disable
     Hyperthreading


Requirements
------------


HSF requires a *NIX kernel with standard libraries. To compile
    all sources,
these packages are needed:

* g++ (>= 4.7)
* make
* octave
* php

If you want to use the initial support for the hwloc library for
    setting the
processor affinities of the threads, the hwloc tools and library
    are also needed:

* hwloc
```

```
You can also directly link the sources without compiling and
    installing them. For
more details see: <http://www.open-mpi.org/projects/hwloc/>\n
To compile hsf without hwloc support, check that '
    USE_LINUX_AFFINITY' is set to
'1' in 'src/pthread/Thread.cpp' and uncomment 'HSFLIBS += -lhwloc
    ' in the Makefile.


For generating the output figures with the 'simfig' tool you need
    the following
additional libraries:

* libmgl-dev (>= 2.0)
* libX11-dev

Mathgl library should be compiled, and libmgl.so.7.0.0 should be
    placed in
'/usr/local/lib' (otherwise the MATHGL variable in the makefile
should be changed to the appropriate location). If you follow the
    compile
instructions of Mathgl this should happen automatically. For more
    information
on Mathgl, please visit: <http://mathgl.sourceforge.net/>

For building the documentation of the Hierarchical Scheduling
    Framework you will
also need a recent version of doxygen:

* doxygen (>= 1.8)

To build the documentation simply run 'make doc' in the HSF
    directory. The
documentation will then be generated in the 'doc' directory.


Installation and Configuration
------------------------------

1. If your HSF folder is not located in '~/git', then please
    change line 3 of
  'hsf_paths.sh' to the path of your HSF folder.
2. In the terminal, type:

       source hsf_paths.sh

  This will set a new '$HSF' variable, and add it to your '$PATH'
      variable. You can
  also add it to your '~/.bashrc' file, to have it load
      automatically
3. Privileges for executing with real-time priorities
  To execute hsf the user needs to have the privileges to switch
      the applications
```

```
  scheduling policy to real-time priority based scheduling and
      execute it a with
  real-time priority. You can do this in two different ways:

  1. (RECOMMENDED) Allow the user to execute applications with
      real-time priority
     using the limits.conf configuration file. To allow the user
         with name 'user'
     to execute its applications with real-time priority, simply
         add the file
     '/etc/security/limits.d/50-rtprio.conf' with the following
         two lines (replace
     'user' with your username):

         user      -     rtprio     unlimited
         user      -     nice       -20

    To apply these changes a reboot of your machine is required.
    Please note that this user can execute any application with
       real-time priority.
    More details about this can be found in the manpage of limits
       .conf
  2. (_NOT_ RECOMMENDED) To execute hsf with real-time priority
      the command could
     simply executed with root privileges for example be using '
         sudo'. However we
     do not recommend executing the framework as root.
     If you still want to use the sudo approach please note the
         following: On some
     older systems, you might have to add the following line to
         your bash profile
     in order to inherit you PATH variable when using 'sudo':

         alias sudo='sudo env PATH=$PATH $@'
4. Then type:

      ./install.sh
5.  run "sudo modprobe msr" before running hsf (is needed for
   temperature measurements)

6. Run HSF!
  You can now type in you terminal the following commands:

      hsf [filename(.xml)]
      simulate [filename(.xml)]
      calculate [metric] [filename]
      show [metric] [filename]
      simfig [filename]
      publish [filename]

  [metric] can be one (or more) of the following:

  * exe|exec      ->  Execution Times
  * resp          ->  Response Times
```

```
  * throughput   ->  Throughput
  * util         ->  Utilization
  * alloc        ->  Resource allocation costs
  * sys          ->  System allocation costs
  * worker       ->  Worker costs
  * missed       ->  Missed deadlines


Simulations on Xeon Phi (and Other Systems With Network File
    Systems)
--------------------------------------------------------------------------------


When simulating on Xeon Phi, please notice that the underlaying
    network file system
is likely to be blocked by the HSF framework, because the
    framework runs at higher
priority than the file system daemon. This will cause undefined
    behavior and blocking
of the framework, when it reads and/or writes to files.

To avoid these blocking, you need to copy all files needed for
    simulations to a
local subdirectory in '/tmp/' and start simulation in that
    directory.

This setup is required if one of the following criteria is met (
    know cases):

  - Simalations using the partitioned EDF-VD scheduler (
      PartitionedEDF_VD)
  - Simulations with flight management system tasks (FlyanceTask)
```

Below find an installation guide for the preempt-RT kernel patch

Listing A.2: InstallRT

```
Installation guide for the preempt_rt linux patch.

This guide follows the instructions for
https://ubuntuforums.org/showthread.php?t=2273355

1) create a working directory where the patched kernel will be
    build and compiled:

  mkdir ~/kernel && cd ~/kernel

2)Download the patch and the corresponding linux kernel: (at the
    moment of my project 4.6.7 was the newest rt patch version)

  #Download the RT patch
  wget https://www.kernel.org/pub/linux/kernel/projects/rt/4.6/
      patch-4.6.7-rt14.patch.gz
```

```
  #Download kernel
  wget https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.6.7.
      tar.gz

3)Unzip the kernel:

  #x - extract
  #z - pipe through gunzip
  #v - verbose (text output)
  #f - from file
  tar -xzvf linux-3.18.11.tar.gz

4)Patching the kernel

  #Move to kernel source directory
  cd linux-3.18.11

  #c - pipe file contents to stdout
  #d - decompress
  gzip -cd ../patch-3.18.11-rt6.patch.gz | patch -p1 --verbose

5)create config file for the new kernel:

  sudo apt-get install libncurses-dev

  make menuconfig

the previous cmd will open a graphical interface: select the
    following options:
  Processor type and features ---> Preemption Model --->Fully
      Preemptible Kernel (RT) [Enter] #Select
  Kernel hacking ---> Memory Debugging  ---> Check for stack
      overflows #Already deselected - do not select
after that save and exit

6)complie the kernel (be advised that this step will take some
    time)

  make

7)make modules and install

sudo make modules_install
sudo make install

8) verify and update


Verify that initrd.img-4.6.7-rt14, vmlinuz-4.6.7-rt14, and config
    -4.6.7-rt14 exist. They should have been created in the
    previous step.


  cd /boot
```

```
  ls

Enable the obtion to select the new kernel while reboot

 sudo update-grub

9) reboot and check
  sudo reboot

after rebooting check the kernel
 uname -a

it should give : Linux nb-10460 4.6.7-rt14 #1 SMP PREEMPT RT Wed
    Oct 5 15:18:10 CEST 2016 x86_64 x86_64 x86_64 GNU/Linux
```

APPENDIX B

Class descriptions

Here the classes that have been added to HSF are described.

B.1 Code structure

In this section the code structure of HSF after the implementation of all extensions is shown. In the figure below B.1, a new class has a green box. In the class boxes the section before the dashed line are the variables. The section after the dashed line are the functions. Private or protected functions and variables are proceeded by a #. Public functions and variables are proceeded by a +. If a function or variable has been added to HSF during this project it is followed by a +.

Figure B.1: The thread hierarchy after extensions to HSF

B.2 Pre-partitioned EDF

The functions of the PrePartitionedEDF class are:

- *constructor*: the constructor of the class takes 4 parameters. The first is the ID of the thread inside the test run. The second is the scheduler level, see 3.2.1. This level has to be 0, if not the constructor will return a warning. The third parameter is the number of used cores in the test run. For each used core (starting from core 0) an EDF scheduler will be created. The forth parameter is a boolean, "thermal_calib". If it is set all schedulers will run on core 0 else they will run on their respective core. This mechanism is used during the thermal calibration tests of the platform, see 6.1.
- *activate()*: This function sets the activation state variable to "activated" and it activates all EDF schedulers. If the EDF schedulers have not been initialized it will first initialize the EDF schedulers by calling the member function initialize().
- *deactivate()*: This function sets the activation state variable to "deactivated" and all EDF schedulers are deactivated.
- *finishedJob()*: This function prints an error message as this function (which originates from the scheduler class) should never be called.
- *join()*: All EDF schedulers are joined, then the thread itself is joined.
- *newJob()*: This function prints an error message as this function (which originates from the scheduler class) should never be called.
- *partitionTasks()*: This function takes a list of runnables (servers and workers) and partitions them to the different EDF schedulers according to the core on which they should run on.
- *schedule()*: This function is the thread itself. It is an empty function and will terminate immediately after test run start. The PrePartitionedEDF scheduler is only used for initializing the test run, no scheduling is done by the thread itself.
- *updateRunnable()*: This function prints an error message as this function (which originates from the scheduler class) should never be called.
- *initialize()*: This function starts the threads of all the EDF schedulers.

B.3 TI_server

As described in 3.1 the implementation consists of 4 components. 3 of these components are new extensions to HSF. In the following the 3 new thread types

are described in detail. The TI-Server thread is implemented in a class called *TIS*, see B.3.1. The TIS-dispatcher has been implemented is the *TISDispatcher* class, see B.3.3. The third component, the Server-TTL is implemented in the class described in B.3.2.

B.3.1 The TIS class

The TIS class is the principle component of the TI-server implementation. It implements the TI-Server thread and it defines all functions that are needed for interaction with the TI-Server thread. In the following find a description of all the functions defined in this class:

- *activate()*: This function sets the priority of the server to its active priority (server priority see 3.2.1 and starts the TTL countdown (see B.3.2). In addition to that, the function activates the last worker if this one did not finish its job in the previous active period. Finally the activation function also frees the activation and event semaphore that are needed by the server in order to be able to serve the load.

- *deactivate()*: This function is the opposite of the previous function. It first deactivates the server-ttl (see B.3.2). It then deactivates the current worker if this one has not been deactivated previously. This is the case if the server is deactivated while the worker has not finished its job yet. After deactivating all load the TIS-dispatcher is activated(see B.3.3) in order to initiate the TI-Server cool-down period. Finally the priority of the server thread is lowered to the inactive priority.

- *join()*: This function first joins the parent scheduler of the server. It then joins the the server-ttl thread. After joining the two threads, that are directly linked to the server, the function frees all semaphores and then joins the server thread.

- *new_job()*: This function has exactly the same functionality as the new_job() function of the EventBased class. Its purpose is to register a new job into the job queue ('"active_queue_"). In this queue the load of the server is defined. It also frees the event semaphore in order to register an event for the server thread itself, which will then handle that event if active. The function is called by the worker assigned to the server.

- *finished_job*: This function is the same as the equivalent EventBased function and it registers if a worker has finished its task. The event semaphore is freed in order to regiseter an event for the server thread. The function is called by the worker assigned to the server.

- *overrunJob()*: This function is the same as the equivalent EventBased function and it registers if a worker has overrun. Same as before the event

semaphore is freed. The function is called by the worker assigned to the server.

- *updateRunnable()*: This function again has an equivalent in the EventBased class and its purpose is to update the active_queue if a worker changes its criteria. The function is again called by the worker that is assigned to the server and it too frees the event semaphore.
- *serve()*: This function is the function that is executed by the server thread itself. Its purpose is to handle incoming events and to activate and deactivate workers. The serve function is very similar to the *schedule()* function in the EventBased class. The server function consists of a while loop that checks if the test run is still active. Inside the loop the function first waits for the activation and the event semaphore. After that the respective events are handled same as in *schedule*. If the server has no events left and the server is still active, meaning that there is some time of the TTL left, then the server sends a signal to the server-ttl thread in order to terminate early. In consequence the server thread is deactivated.
- *setCooldown()*: This function sets the cool-down time t_c of the server and is called by the parse class.
- *setTTL()*: This function sets the active time t_a of the server and is called during the initialization of the test run by the parse class.
- *handleUpdate()*: Is equivalent to the handleUpdate function of the EventBased class. It erases the old entry of the updated runnable in the active Queue and inserts the new entry.
- *handleOverrun()*: Same as in EventBased.
- *handleFinish()*: This function is equivalent to the handleFinish() function of EventBased. It deletes the finished runnable from the active queue and updated the current runnable.
- *newPeriod()*: This function is called by the TIS dispatcher (see B.3.3) after a cool-down period. It registers a new job at the EDF scheduler. The EDF scheduler then in turn will activate the TI-server.
- *finishedPeriod()*: This function is called by the server-ttl thread (see B.3.2). All worker that have been activated during the previous period are send a Signal that will block them from execution.(for more details see **??**). After that the finishedJob function of the EDF is called in order to register a finished job. The EDF will deactivate the server as a consequence.

B.3.2 The Server-TTL

The Server-TTL class is the implementation of the Server-TTL thread. The TTL sleep time of the thread is done with clocknanosleep()[1]. When the thread wakes up it calls the finishedPeriod() function of the TIS class in order to register a "finished Period" event at the EDF scheduler.
If the TI-server is active but has no load left, it will send a signal to its server-ttl. This will cause the clocknanosleep to interrupt and the finishedPeriod() call will be made early. Below is a list of all the functions defined in the Server_TTL class:

- activate(): this function sets the priority of the thread to the dispatcher priority (see 3.2.1. It also frees the active semaphore and sets the state variable to "activated".
- deactivate(): This function sets the priority of the thread to inactive priority 3.2.1. It sets the state variable to "deactivated".
- join(): This function will set the state variable to "activated", it will free the active semaphore and it will then join the thread.
- getTTL(): This function returns the TTL.
- getRemain(): This function returns the remaining time of TTL if the clocknanosleep() is interrupted.
- setTTL(): This function sets the TTL.
- signalHandler(): This function causes the clocknanosleep() to interrupt but it does nothing else.
- wrapper(): This function executes the server-TTL activity described above in a while loop that checks if the test run is still active. It is executed by the server-TTL thread.

B.3.3 The TISDispatcher

The TISDidpatcher is class that defines the TIS-dispatcher thread. For sleeping the clocknanosleep() function is again used, same as in the implementation of the Server-TTL. After wake-up the newPeriod() function of the TIS class is called and a "new period" event is registered at the EDF scheduler.
Below find a list of all the functions defined in the TISDispatcher class:

- activate(): This function activates the dispatcher by freeing the active semaphore and by setting the priority of the dispatcher to the dispatcher priority (see 3.2.1).

- join(): This function will first join the corresponding TI- server. Then the active semaphore is freed and the dispatcher thread is joined.
- dispatch(): This function is the function that is executed by the dispatcher thread itself. It executes the above described behaviour in a while loop that is checking if the test run is still active
- getCooldown(): This function returns the cool-down time.
- setCooldown(): This function enables to set the cool-down time and is called during initialization by the parser.
- setTIS(): This function sets the links the TI-server with the TISDispatcher. This function is called during initialization by the parser.

Appendix C

Creating an XML file for TI-Server experiments

In order to execute any simulation with HSF an XML file has to be created which can then be parsed by HSF. In this section all elements that are needed for the simulation of a TI-Server are explained.

C.1 main Element, the Simulation element

The XML file is composed of one main element in which all Schedulers, servers and workers are defined. The element has one attribute defining the duration of the simulation. The Simulation element thus looks as follows:
¡simulation name=”...”¿
¡duration value=”...” unit=”...”/¿
...
...
...
¡/simulation¿

”name” is the name of the simulation and can be any string.
”value” takes an integer.
”units” takes ”h” if the duration is given in hours, ”min” when in minutes, ”sec” when in seconds, ”ms” when in milliseconds, ”us” when in microseconds and ”ns” when in nanoseconds.

C.2 runnable element

After defining the Simulation element, this one is filled with runnable elements. The runnable elements have a ”type” value defined. This devides the runnable element into 3 classes, the scheduler, the server and the worker. Each class has its

unique properties and variables. In C.2.1, C.2.2 and C.2.3 the different runnable will be explained.

Here is the basic definition of the runnable element:
¡runnable type=”...” ... ¿
...
...
¡/runnable¿

C.2.1 Scheduler type

The scheduler runnable element has a additional value, the ”algorithm”. In this project the following values are needed:

- ”ppEDF” this creates an pre partitioned EDF scheduler see **??**. As an additional attribute this element takes ”cores”= an integer value indicating the number of used cores.
- ”thermal_calib” is the same as ”ppEDF” except that all schedulers and dispatchers are running on core 0.
- ”TI_Server” is exactly the same as ”ppEDF” under a different name
- ”EDF” creates an EDF scheduler. This is only defined on core 0 for the moment. If more cores should be used use ”ppEDF”.

In the following example a ppEDF scheduler with 4 cores is represented:

¡runnable type=”scheduler” cores=”4” algorithm=”ppEDF” ¿
...
...
¡/runnable¿

Inside the scheduler runnable workers and schedulers can be defined. With this a scheduler hierarchies can be build. The lower a scheduler is in the hierarchy the lower its priority level is. In ppEDF, thermal_calib, TI_Server, and all event based schedulers servers can be defined. For all other scheduler servers are not yet implemented.

C.2.2 Server type

The scheduler runnable element has a additional value, the ”server_type”. The only used value is:

- ”TIS” creates a thermal isolation server see ??. For the thermal isolation server 3 diffrent attrbutes have to be defined, the core attribute setting the core on which the server runs, the TTL attribute which sets maximum active time of the server and the cooldown this defines the cooldown time of the server.
  Exemple:
  ¡runnable type=”server” server_type=”TIS” ¿
  ¡core value=”0” /¿
  ¡cooldown value=”10” units=”ms” /¿
  ¡TTL value=”10” units=”ms” /¿
  ...
  ...
  ¡/runnable¿

Inside a Server element worker elements can be defined. These workers then represent the load of the respective server.

C.2.3 Worker type

The last element is the worker element. The worker gets two new values. The task value defines the task that will be executed by the worker. Currently ”busy_wait” and ”temp_meas” are used in this theses. The second value is the periodicity. It can be ”periodic”, ”aperiodic”, ”sporadic” or ”periodic_jitter”. Depending on the periodicity the worker will have different attributes.

- If ”periodic” then the worker has an period attribute
  ¡period value=”10” units=”ms” /¿
- If ”sporadic” no additional attribute is needed.
- If ”periodic_jitter” then period and jitter attributes are needed
  ¡period value=”10” units=”ms” /¿
  ¡jitter value=”1” units=”ms” /¿
- If ”aperiodic” then release_time is needed
  ¡release_time value=”10” units=”ms” /¿

All worker element has a wect attribute ¡wect value=”10” units=”ms” /¿

C.3 element hierarchy

In order to build larger and complex schedulers a hierarchy of the elements described in C.1 and C.2.Here are the most important rules to follow in order to build a successful simulation:

- The heighest element in the hierarchy always has to be an simulation element.
- Inside the simulation element a level 0 (heighest level) scheduler has to be defined first. This can be any scheduler type. But type ppEDF TI_Server and thermal_calib have to be level 0
- inside the level 0 scheduler a hierarchy of schedulers, servers and workers can be defined as described in C.2.1, C.2.2 and C.2.3.

C.3.1 TI-Server example

Here is an example of TI-Server simulation. In this simulation 4 cores are used. On core 0 an EDF scheduler is scheduling 2 workers. The first worker is measuring the temperature of all cores in 1 ms intervals. The second worker executes a busy wait task with period

Listing C.1: Test

```
1 <simulation name="tis">
2
3 <duration value="6" units="sec"/>
4
5 <runnable type="scheduler" cores="4" algorithm="TI_Server">
6
7   <runnable type="worker" periodicity="periodic" task="temp_meas">
8      <core value="0"/>
9      <period value="1"  units="ms" />
10     <wcet   value="100" units="us" />
11     <relative_deadline value="1" units="ms" />
12  </runnable>
13
14  <runnable type="worker" periodicity="periodic" task="busy_wait">
15       <core value="0"/>
16       <period value="10"  units="ms" />
17       <wcet   value="5" units="ms" />
18       <relative_deadline value="10" units="ms" />
19     </runnable>
20
21  <runnable type="server" server_type="TIS">
22     <core value="1"/>
23     <cooldown value="5"  units="ms" />
24     <ttl value="5" units="ms" />
25
26     <runnable type="worker" periodicity="periodic" task="busy_wait">
27       <core value="1"/>
28       <period value="25"  units="ms" />
29       <wcet   value="10" units="ms" />
30       <relative_deadline value="25" units="ms" />
31     </runnable>
32
```

```
33     </runnable>
34
35     <runnable type="server" server_type="TIS">
36         <core value="2"/>
37         <cooldown value="3"  units="ms" />
38         <ttl value="7" units="ms" />
39
40         <runnable type="worker" periodicity="periodic" task="busy_wait">
41           <core value="2"/>
42           <period value="10"  units="ms" />
43           <wcet   value="6" units="ms" />
44           <relative_deadline value="10" units="ms" />
45         </runnable>
46
47     </runnable>
48
49     <runnable type="server" server_type="TIS">
50         <core value="3"/>
51         <cooldown value="6"  units="ms" />
52         <ttl value="4" units="ms" />
53
54         <runnable type="worker" periodicity="periodic" task="busy_wait">
55           <core value="3"/>
56           <period value="20"  units="ms" />
57           <wcet   value="5" units="ms" />
58           <relative_deadline value="20" units="ms" />
59         </runnable>
60
61     </runnable>
62
63 </runnable>
64
65 </simulation>
```

Appendix D

Original Project Assignment

Here find the original project assignment

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Semester Thesis at the
Department of Information Technology and
Electrical Engineering

for

Max Millen

Orchestrate the Mixed-Criticality Melody
Reconcile Temperature with Safety

Advisors: Dr. Rehan Ahmed
Pengcheng Huang

Professor: Prof. Dr. Lothar Thiele

Handout Date: 10. 10. 2016
Due Date: 16. 01. 2017

1 Project Description

Mixed-Criticality is emerging as a significant trend for future computer systems, e.g. automotive, avionics, medical and cloud systems. For such systems, applications of different safety/security criticality levels share a common commercial-off-the-shelf computing platform, to meet increased performance demand and to reduce system cost. The ultimate design goal for mixed criticality systems is to provide different levels of assurance to applications of different criticality levels, while the main difficulty is that resource sharing could lead to mutual interferences among critical and non-critical applications, jeopardizing safety/security guarantees.

Meanwhile, all modern processing platforms are thermally constrained i.e. their temperature has to be below a threshold to ensure safe operation. With continuously shrinking feature sizes of circuits, power densities of processors have increased rapidly. This increase in power density corelates to increase in temperature; making temperature an increasingly stringent design constraint. In a mixed-criticality setting, temperature could greatly affect system reliability; possibly even lead to system failure due to thermally-triggered shutdown. Therefore, it is imperative to consider thermal constraints while designing mixed-criticality systems.

2 Project Goals

As part of this thesis, you will develop an evaluation framework for testing mixed criticality thermal protection mechanisms on a hardware testbed. The hardware testbed will comprise of an x86 platform running Ubuntu. Depending on your interest, you may also work on algorithm development/extension and theoretical analysis. The tasks of this thesis are detailed as follows:

- Understand the basic building blocks of the scheduling framework. These include POSIX threads, RTPreempt Linux kernel patch.
- Prepare the evaluation platform for development.
- Build a basic scheduling framework that is able to spawn realtime tasks according to a static schedule.
- Integrate a thermal model developed by the project supervisors into the scheduling framework.
- Build a "thermal isolation" server that is able to regulate execution of Lo Criticality applications to guarantee thermal isolation.
- Perform evaluation experiments on the hardware.

The evaluation experiments should demonstrate:

- The working of the thermal isolation scheme
- In case of deviations from the expected result, the reasons for deviations should be clearly explained and mitigation strategies should be proposed.

3 Project Tasks

3.1 Get to know the building blocks

the building blocks include:

1. RTPreempt Linux kernel patch[2]: With this kernel patch, the kernel gains “hard” realtime capabilities. As part of this task, RTPreempt patch will be applied to a linux installation.

2. POSIX Threads [1]: POSIX thread libraries allow a user to spawn threads with different properties (priority, core affinity e.t.c.). In this thesis our realtime processes will be POSIX threads.

Due Date: 2 Weeks

3.2 Get to know the existing code/framework

We currently have the following two modules implemented:

1. The meter app periodically reads processor temperature and dumps its value in a log file.

2. SF3P[3]: This is a scheduling framework developed here at TIK for fast prototyping of scheduling algorithms. It may be easier to add thermal protection to SF3P.

there modules can give you a good starting point on how to build/evaluate scheduling algorithms and how to record temperature traces.
Due Date: 1 Week

3.3 Building the basic hypervisor

The basic hypervisor receives a static schedule as a text file and executes that schedule by spawning POSIX threads and issuing sleep/run commands according to the schedule. Ability to assign real-time priorities to threads and to set core affinity of different threads is important. Hypervisor should also have a lightweight module which periodically monitors/logs system temperature. You will need to verify that SF3P has these capabilities; and where missing, add these capabilities.
Due Date: 2 Weeks

3.4 Explore different source applications

Typically, the heating of a given core is application dependent (Different applications have different thermal characteristics). In this task you will try to build applications have different thermal characteristics.
Due Date: 1 week

3.5 Build the LO Scheduler

Based on the scheme developed by project advisers, implement the LO scheduler. The LO scheduler should be able to support various LO criticality servers which are assigned thermal budgets. Tasks of a given LO server are only allowed to execute if sufficient budget is available. Budget consumption and replenishment mechanisms will also need to be implemented.
Due Date: 5 Weeks

3.6 Perform evaluation experiments

Perform experiments illustrating that the proposed thermal protection mechanism works. In case of deviation from this expectation, the reasons for deviation should be clearly explained and possible mitigation strategies should be proposed.
Due Date: 1 Week

3.7 Finalizing report/final presentation

The last few weeks of this project will be left for writing report and preparing the presentation. It is highly recommended that you should keep working on the report continually as you progress through the project.
Due Date: 2 Weeks

3.8 Participate in the theoretical development of the thermal protection scheme (Optional)

Depending on your interest, you may participate in the development of the actual thermal protection mechanism. You contribution may include:

- An optimal scheme for assigning budgets to Lo criticality servers.
- Given a thermal budget, try to give some timing guarantees.

3.9 Write a paper (Optional)

This work can result in a research paper, in case the results from the previous tasks are satisfying a research paper can be written together with the advisors.

4 Project Organization

4.1 Weekly Meeting

There will be a weekly meeting to discuss the project's progress based on a schedule defined at the beginning of the project. A progress report should be provided at least 24 hours before the meeting.

4.2 Report

A hard-copy of the report is to be turned in. The copy remains the property of the Computer Engineering and Networks Laboratory. A copy of the developed software needs to committed to the SVN repository at the end of the project.

4.3 Initial and Final Presentation

In the first month of the project, the topic of the project will be presented in a short presentation (less than 5 minutes) during the group meeting of the Computer Engineering Lab. The duration of the talk is limited to three minutes. At the end of the project, the outcome of the project will be presented in a 15 minutes talk, again during the group meeting of the Computer Engineering Lab.

4.3.1 Grading

The grading will be based on the departement regulation and take into consideration all the aspects of the students work during the project. In order to complete the project with a positive grading, the **minimal** requriements torwards the student are to complete the tasks **3.1 - 3.7**

References

[1] Linux tutorial: Posix threads. `http://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html`.

[2] Rt preempt linux kernel patch. `https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO`.

[3] SF3P tutorial. `http://www.tik.ee.ethz.ch/~euretile/scheduling/sf3p_manual.pdf`.

Zürich, October 2016

Bibliography

[1] clock nanosleep - high-resolution sleep with specifiable clock. http://man7.org/linux/man-pages/man2/clock_nanosleep.2.html.

[2] cpufreq-set(1) - linux man page. https://linux.die.net/man/1/cpufreq-set.

[3] getrusage - get resource usage. http://man7.org/linux/man-pages/man2/getrusage.2.html.

[4] How to control fan speed. http://www.thinkwiki.org/wiki/How_to_control_fan_speed.

[5] How to use cpufrequtils. http://www.thinkwiki.org/wiki/How_to_use_cpufrequtils.

[6] msr - x86 cpu msr access device. http://man7.org/linux/man-pages/man4/msr.4.html.

[7] pthread kill - send a signal to a thread. http://man7.org/linux/man-pages/man3/pthread_kill.3.html.

[8] pthread sigqueue - queue a signal and data to a thread. http://man7.org/linux/man-pages/man3/pthread_sigqueue.3.html.

[9] Real-time linux wiki. https://rt.wiki.kernel.org/index.php/Main_Page.

[10] signal - overview of signals. http://man7.org/linux/man-pages/man7/signal.7.html.

[11] Thinkfan. http://thinkwiki.de/Thinkfan.

[12] Rehan Ahmed Pengcheng Huang and Lothar Thiele. On the design and application of thermal isolation servers.

[13] Lukas Sigrist. Implementation and evaluation of mixed-criticality scheduling algorithms for multi-core systems, 2014.