MASTER

Performance of resource access protocols
time measurement of multicore real time resource access protocols

Verwielen, M.P.W.

Award date:
2016

Link to publication
Performance of resource access protocols

Time measurement of multicore real time resource access protocols

Maikel Verwielen

Committee members:
Dr. ir. Reinder J. Bril
Lic. MSc. Sara Afshar
Dr. MSc. Nabi Najafabadi
Dr. MSc. Moris Behnam

Version 1.0 (Final)
Thursday 30th June, 2016 18:15

Eindhoven, June, 2016
Abstract

A clear trend [1] in computers and electronics is the rapid increase in the number of cores per chip. These extra cores share access to resources such as I/O and memory. To keep a favourable scalability of code executed vs number of cores, it is important to have efficient access to shared resources.

Cores that share resources might get blocked by the resource access of another core. Widely used resource access protocols either wait for the resource to become available or already start processing some other task in the meanwhile. Both of these resource access protocols have their own advantages.

Non-preemptive waiting on the release of the resource allows a task to directly start executing the moment the resource becomes available. Instead of waiting, allowing other tasks to process in the meanwhile means processing time is spend on tasks instead of waiting.

Using a resource access protocol that sometimes waits on the release of a resource and sometimes allows other tasks to execute can benefit from the advantages of both protocols. The project consists of the implementation of new resource locking techniques proposed by Afshar et al in [2] and [3]. Measurements of introduced time and memory overhead of flexible spin-lock protocols are provided.

Keywords: Resource access protocols, spin-lock, preemption, implementation, measurement, FPGA, ERIKA and OSEK.
Preface

I would like to thank all the people who helped me write this thesis. I would like to express my sincere gratitude to Reider J. Bril at the technical university Eindhoven who has contributed with lots of ideas and valuable discussions. I’m grateful to my supervisor Sara Afshar from Mälardalen university Sweden without them this work would have been impossible. I am thankful to Moris Behnam and Nabi Najafabadi for participating in the graduation hearing. A special thanks to Paolo Gai who helped me with practical support.

Many thanks to my family, friends and colleagues.

Maikel Verwielen Västeras, September 2015
# Contents

## Contents

List of Figures vii

Listings xi

## 1 Introduction

1.1 Background ........................................ 2
1.2 Problem statement .................................. 6
1.3 Objectives and Goals ............................... 6
1.4 Thesis Contributions ............................... 6
1.5 Reading guide ..................................... 7
1.6 Research Questions ................................. 7
1.7 Thesis Outline ..................................... 8

## 2 Related Work

2.1 Measuring resource access protocols ................ 10
2.2 LITMUS\textsuperscript{RT} ............................... 12
2.3 OSEK and AUTOSAR ................................. 13
2.4 Spin-lock ........................................... 14
2.5 Hardware and semaphores ........................... 15

## 3 System model

3.1 The $CP$ spin-lock priority ........................ 18
3.2 The $\bar{C}P$ spin-lock priority ...................... 18

## 4 Feasibility Study

## 5 The ERIKA OS

5.1 System overview .................................... 21
5.2 Initialisation ....................................... 25
5.3 Task handling ...................................... 26
5.4 Ready queue ....................................... 27
5.5 Stack and context switch .......................... 28
5.6 Resource access .................................... 29
5.7 Remote notification ................................ 42

## 6 PROJECT

6.1 Project overview .................................... 52
6.2 Additional stacks ................................... 54
# CONTENTS

7 Flexible priority tool 55
  7.1 Requirements 58
  7.2 Specification 58
  7.3 Design 59
  7.4 Check Invariants 61

8 Spinning on a local spin_lock 62
  8.1 Requirements 64
  8.2 Specification 65
  8.3 Design 66
  8.4 Check Invariants 74
  8.5 Measurements 77

9 Context switch 88
  9.1 Introduction 89
  9.2 Requirements 90
  9.3 Specification 91
  9.4 Design 93
  9.5 Check Invariants 102
  9.6 Measurements 105

10 Conclusions 109

Bibliography 110
  Appendices 112

A Final measurements 112

B Memory organization 115
  B.1 Avalon_bus 117
  B.2 SDRAM 119
  B.3 FPGA architecture 121

C User manual 124
  C.1 Installation 125
  C.2 Creating the hardware design 130
  C.3 Creation of software 159

D Measurement of read and write 172

E Code 176
# List of Figures

1.1 Frequency vs energy for 1-4 cores ........................................... 2  
1.2 Higher priority task is blocked on a resource ............................... 3  
1.3 Both suspension and spin-based blocking can be preferable ..................... 5  

5.1 RT-Druid generated code, ERIKA OS and the Altera HAL-layer .................. 21  
5.2 The file include structure of the OO-kernel ................................... 23  
5.3 How functions of files are called between files and provided to the API-layer .... 24  
5.4 A symbol in the top corner of the page will keep track of the function group described. 24  
5.5 The call-graph of StartOS() .................................................... 25  
5.6 The ActivateTask() function puts a task in the ready queue and possibly lets it execute 26  
5.7 The ready queue is implemented like a linked list .............................. 27  
5.8 Typical stack layout of a preempted thread in ERIKA ............................. 28  
5.9 The resource access of a local and global resource ............................. 29  
5.10 The call-graph of GetResource() .............................................. 30  
5.11 The data-structures of GetResource() ......................................... 30  
5.12 The call-graph of ReleaseResource() ......................................... 32  
5.13 The data-structures of ReleaseResource() ..................................... 33  
5.14 Global resource FIFO queue .................................................... 36  
5.15 The call-graph of spin_in() .................................................... 37  
5.16 The data-structures used by spin_in() and spin_out() ............................ 38  
5.17 The data-structures used by spin_in() and spin_out() ............................ 39  
5.18 The data-structures used by spin_in() and spin_out() ............................ 40  
5.19 Core $P_2$ uses $R_0$, core $P_1$ and $P_1$ waits for release .................... 41  
5.20 The connections for inter core interrupts ..................................... 43  
5.21 The data-structures used for sending and receiving an RN ..................... 44  
5.22 The rn_handler() function ..................................................... 45  
5.23 The rn_handler() function ..................................................... 47  

6.1 Steps that need to be implemented in the ERIKA kernel .......................... 51  
6.2 The execution of a task set and the corresponding system ceiling versus time ........ 52  
6.3 A RAP with a priority below $H_P$ requires at least one additional stack .............. 54  

7.1 Specification of the priority of spinning task ..................................... 56  
7.2 A tool is added parallel to RT-Druid to extract the $CP$ and $CP$ priorities from the OIL file .................................................. 56  
7.3 Part of the input and output file of RT-Druid .................................. 58  
7.4 The tool reads the config.oil input file and writes to the Eecfg.c files ........... 59  
7.5 Behavioral diagram and data-structures ....................................... 60  

8.1 Spinning on a global variable vs a local spin_lock ................................ 63  
8.2 The data-structures used by spin_in() and spin_out() ............................ 70  
8.3 Task $\tau_{11}$ on $P_1$ uses $R_0$, task $\tau_{12}$ on $P_2$ waits for release of $R_0$ ............ 71
8.4 The data-structures used by spin_in() and spin_out() ........................................ 72
8.5 The data-structures used by spin_in() and spin_out() ........................................ 72
8.6 Overview of locking and unlocking a resource ....................................................... 73
8.7 HP/MSRP without overhead .................................................................................... 79
8.8 HP/MSRP spinning on a global spin_lock .............................................................. 79
8.9 HP/MSRP spinning on a local spin_lock ............................................................... 79
8.10 Overhead is introduced by queue management and updating global spin_lock .... 80
8.11 Overview of locking and unlocking a resource (colors map to overhead) .............. 81
8.12 Overhead is introduced by queue management and updating local spin_lock ......... 82
8.13 Overview of locking and unlocking a resource (colors map to overhead) .............. 84
8.14 A comparison of the approaches ........................................................................... 85
8.15 The measurement results (allows to compare the relative size) ............................. 85
8.16 The greatest contributor of overhead is the interrupt sending and receiving ......... 87

9.1 The spinning task can be preempted and needs to be resumed in correct manner ... 89
9.2 The variables that track the preemption order and additionally the executing task when a resource requesting task is resumed .................................................. 91
9.3 The spinning task can be preempted and needs to be resumed in correct manner ... 93
9.4 The moments the priority needs to change related to resource access .................. 93
9.5 The different type of context switches that can occur ............................................ 96
9.6 Functions used to activate, terminate and switch between tasks (concerns the original code) ........................................................................................................ 96
9.7 rn_execute() and ReleaseResource() perform the additional context switches .... 99
9.8 Context switch example where tasks arrive during the resource access .................. 99
9.9 Total diagram ........................................................................................................ 100
9.10 Measurement of ReleaseResource() .................................................................... 106
9.11 Measurement of rn_execute() ............................................................................ 106
9.12 Measurement of spin_in() .................................................................................... 106
9.13 The terms of the original ERIKA OS .................................................................. 108
9.14 The terms of the flexible spin-lock protocol, while the spinning task is not preempted .......................................................... 107
9.15 The terms of the flexible spin-lock protocol, while the spinning task gets preempted .......................................................... 107
9.16 The overhead under MSRP and FSLM ............................................................... 108

A.1 The tightly coupled data and instruction memory interconnects ......................... 112
A.2 The properties of the core, the cache and memory interface ............................... 112
A.3 The measurements performed on MSRP and FSLM .......................................... 114
A.4 The measurements results .................................................................................... 114

B.1 Multiprocessor system with shared resource ....................................................... 115
B.2 Addressing of the peripherals in the SOPC design builder ................................. 115
B.3 Storing the core specific components in the memory .......................................... 116
B.4 Overview of bus topology .................................................................................... 117
B.5 Each slave has a Arbiter in case multiple masters are connected ....................... 118
B.6 Pipelined data transfer on the Avalon bus .......................................................... 118
B.7 The architecture of the SD-ram memory .............................................................. 119
B.8 Each bank is like a matrix of rows and columns .................................................. 119
B.9 Architecture FPGA ............................................................................................ 121
B.10 Architecture FPGA .......................................................................................... 121
B.11 Architecture FPGA .......................................................................................... 122
B.12 Interconnects of logic elements and LABs ........................................................ 122
B.13 Architecture FPGA .......................................................................................... 123
B.14 Architecture FPGA .......................................................................................... 123

C.1 The overview of the 4 core demo hardware configuration ..................................... 124
| C.2 | Download the program ........................................... 125 |
| C.3 | Check whether the plugins are installed correctly ............ 126 |
| C.4 | Connect the board and turn it on ................................ 127 |
| C.5 | Open the device manager ......................................... 127 |
| C.6 | Update device driver ............................................ 128 |
| C.7 | Update device driver ............................................ 128 |
| C.8 | Update device driver ............................................ 128 |
| C.9 | Update device driver ............................................ 129 |
| C.10| Update device driver ............................................. 129 |
| C.11| Update device driver ............................................. 129 |
| C.12| Start the program ................................................ 130 |
| C.13| New project ....................................................... 130 |
| C.14| New Quartus II project ........................................... 130 |
| C.15| Create a new project ............................................. 131 |
| C.16| Select Next ....................................................... 132 |
| C.17| Select the EP3C16F484C6 ......................................... 132 |
| C.18| Select the EP3C16F484C6 ......................................... 133 |
| C.19| How to open the SOPC builder within Quartus .................... 133 |
| C.20| Provide a name for the SOPC design ............................. 133 |
| C.21| Add a core to the design ......................................... 134 |
| C.22| Add a core to the design ......................................... 134 |
| C.23| Add a SDRAM controller to the design ............................ 135 |
| C.24| Add a SDRAM controller to the design ............................ 135 |
| C.25| How to rename components in the design ......................... 136 |
| C.26| Add a internal RAM to the design ................................ 137 |
| C.27| Add a internal RAM to the design ................................ 137 |
| C.28| Add a PLL step 1 .................................................. 138 |
| C.29| Add a PLL step 2 .................................................. 138 |
| C.30| Add a PLL step 3 .................................................. 139 |
| C.31| Add a PLL step 4 .................................................. 139 |
| C.32| Add a PLL step 5 .................................................. 140 |
| C.33| Add a PLL step 6 .................................................. 140 |
| C.34| Add a PLL step 7 .................................................. 141 |
| C.35| Add a inter core interrupt port step 3 .......................... 141 |
| C.36| Add a inter core interrupt port step 3 .......................... 141 |
| C.37| The connections for inter core interrupts ......................... 142 |
| C.38| Add an inter core interrupt port step 1 .......................... 142 |
| C.39| Add an inter core interrupt port step 2 .......................... 143 |
| C.40| Add an inter core interrupt port step 3 .......................... 143 |
| C.41| Add an inter core interrupt output port ......................... 144 |
| C.42| Add a inter core interrupt output port ............................ 144 |
| C.43| Change the position of the program code in the memory ......... 146 |
| C.44| Change the cpuID control register ................................ 147 |
| C.45| Change the order in which the components appear ............... 147 |
| C.46| Add a JTAG UART to the design ................................ 148 |
| C.47| Add a JTAG UART to the design ................................ 148 |
| C.48| Adding the precision counter (used for measurements) ........... 149 |
| C.49| Adding the precision counter (used for measurements) ........... 149 |
| C.50| To connect the cores reduce the number of details visible .... 150 |
| C.51| Final overview ..................................................... 151 |
| C.52| The clock signals ................................................ 153 |
| C.53| Change the clock signal for a specific component ................ 153 |
| C.54| Auto assign ....................................................... 153 |
| C.55| Save the design ................................................... 153 |
LIST OF FIGURES

C.56 Generate the design .............................................. 153
C.57 Adding files to the design ...................................... 154
C.58 Adding files to the design ...................................... 154
C.59 To compile and synthesise the design .............................. 155
C.60 How to open the assignment editor ............................... 155
C.61 The pin editor .................................................... 156
C.62 To compile the design ........................................... 157
C.63 Select OK ......................................................... 157
C.64 The Quartus programmer ......................................... 158
C.65 Add the .sof file to be programmed ............................ 158
C.66 Add the .sof file to be programmed ............................ 158
C.67 Start the Nios software as a administrator ...................... 159
C.68 Create a new RT-Druid Oil and C/C++ Project .................. 159
C.69 Create a 2 core demo design ................................... 160
C.70 Specify the project name ........................................ 160
C.71 Let the project files appear .................................... 160
C.72 Create a new system library .................................... 161
C.73 Create and build the system library ............................. 161
C.74 Change to conf.oil file to point to the system library and the hardware project file . 162
C.75 Clean the project twice, a list of OK should appear .................. 163
C.76 Building now gives some errors .................................. 163
C.77 Check the properties of the system library ..................... 167
C.78 Check the properties of the system library ..................... 167
C.79 Check the properties of the system library ..................... 168
C.80 The memory is to small to contain a design with more than one task per core .......... 168
C.81 Create the Run configuration ................................... 169
C.82 Create the Run configuration ................................... 170
C.83 Change the preferences ......................................... 170
C.84 Change the preferences to allow for multiple active runs ................. 171
C.85 Create a multi-core Run configuration .......................... 171
C.86 Press the button to test the program in case of design with tasks and resource access . 171
D.1 The influence of reading while writing ............................. 173
D.2 The influence of reading while reading ........................... 173
D.3 The influence of writing while writing ............................ 173
D.4 The influence of writing while reading ........................... 174
D.5 The influence of writing a matrix while writing ................ 174
D.6 The influence of writing a matrix while reading ................ 174
D.7 The time reading takes while 0-3 other core’s read to that shared data .................. 175
D.8 The time reading takes while 0-3 other core’s write to that shared data .................. 175
D.9 The time writing takes while 0-3 other core’s reading to that shared data .................. 175
D.10 The time writing takes while 0-3 other core’s write to that shared data .................. 175
Listings

5.1 Part of an OIL file with a textual definition of the instance cpu0 of the object CPU_DATA
5.2 A simple main program
5.3 pseudo code rq_insert(\tau_{new})
5.4 c-code rq_insert()
5.5 The pseudo code of function GetResource()
5.6 The c-code of function GetResource()
5.7 The pseudo code of the function ReleaseResource()
5.8 The c-code of function ReleaseResource()
5.9 The c-code of function spin_in() and spin_out()
5.10 Inter processor interrupt (HAL)
5.11 The interrupt signal is output by a general I/O (HAL)
5.12 The pseudo code of the function rn_send()
5.13 The c-code of the function rn_send()
5.14 Handling of a remote notification
5.15 The pseudo code of the function rn_handler()
5.16 The c-code of the function rn_handler()
5.17 The pseudo code of the function rn_execute(). Handling of a remote notification
5.18 The c-code of the function rn_execute()
7.1 Part of an OIL-file
7.2 The generated C code by RT-Druid for core P1
7.3 The output code
8.1 Pseudo code of spin_in(), spinning on a local spin_lock
8.2 Pseudo code of spin_out(), spinning on a local spin_lock
8.3 Pseudo of rn_execute(), spinning on a local spin_lock
8.4 C-code of spin_in() and spin_out(), spinning on a local spin_lock
8.5 Pseudo code original spin_in() and spin_out()
8.6 Pseudo code spin_in() (Measurement of HP with local spin_lock)
8.7 Pseudo code spin_out() (Measurement of HP with local spin_lock)
8.8 Pseudo code of rn_send(), measurements
8.9 Pseudo code of rn_handler(), measurements
8.10 Pseudo code of rn_execute(), measurements
9.1 Pseudo code of GetResource(), change system ceiling
9.2 Pseudo code of spin_in(), change system ceiling
9.3 Pseudo code rn_execute(), change system ceiling
9.4 Pseudo code ReleaseResource(), change system ceiling
9.5 Parts of the OIL file that are needed to configure a multi-stack OS
9.6 Pseudo code of rn_execute(), context switch
9.7 Pseudo code ReleaseResource(), context switch
9.8 Pseudo code spin_out(), check the executing task
9.9 C-code spin_out(), check the executing task
9.10 Initialising EE_th_next part of eecfg.c
C.1 The code to be used in software (NOT in Quartus hardware)
C.2 The code of cpu0_main.c
LISTINGS

C.3 The code of cpu1_main.c ................................................. 165
C.4 The code of task0.c .......................................................... 165
C.5 The code of task2.c .......................................................... 165
C.6 The code of cpu0_main.c .................................................... 166
C.7 The code of cpu1_main.c .................................................... 166
D.1 The code of core 0, to measure the write and read times ............ 172
D.2 The code of core $P_1..P_3$, to measure the write and read times .... 173
D.3 The code of core 1..3, to measure the write and read times ........... 173
D.4 The code of core 1..3, to measure the write and read times ........... 174
E.1 The function names are defined with extended names depending on the kernel-type . 176
E.2 The c-code of spin_in(), flexible spin-lock kernel code .................. 177
E.3 The c-code of spin_out(), flexible spin-lock kernel code ................. 177
E.4 The c-code of ReleaseResource(), flexible spin-lock kernel code ........ 177
E.5 The c-code of rn_execute(), the MSRP/HP implementation on a local spin-lock ... 178
E.6 The c-code of rn_execute(), flexible spin-lock kernel code .............. 178
E.7 The c-code of GetResource() functionally unchained in the original and flexible spin-lock protocol ........................................... 178
Abbreviations and Definitions

API Application Programming Interface
ASM ASseMbly language
AUTOSAR AUtomotive Open System ARchitecture
CISC Complex Instruction Set Computing
CP Ceiling Priority (maximum of the global resource ceilings on a core)
$CP$ maximum of the (local and global) resource ceilings on a core
CPU Central Processing Unit
ERIKA Embedded Real-tIme Kernel Architecture
FBGA Fine pitch Ball Grid Array
FIFO First In First Out
FP Fixed Priority
FPGA Field Programmable Gate Array
HAL Hardware Abstraction Layer
HP Highest Priority
IPIC InterProcessor Interrupt Controller
LAB Logic Array Block
LE Logic Element
LP Lowest Priority
LUT Look Up Table
MCC Matrix Computing Coprocessor
MSPR Multiprocessor Stack Resource Policy
MPCP Multiprocessor Priority Ceiling Protocol
OO OSEK/VDX OS kernel API
OP Own Priority
OSEK Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen
PCP Priority Ceiling Protocol
PIO Parallel Input Output
PIP Priority Inheritance Protocol
PLL Phase Locked Loop
RAP Resource Access Protocol
RISC Reduced Instruction Set Computer
RN Remote Notification
RTOS Real Time Operating System
SDRAM Synchronous Dynamic Random-Access Memory
SRP Stack Resource Policy
SOPC System on a Programmable Chip Builder
VDX Vehicle Distributed eXecutive
Chapter 1

Introduction

Computer scientists want to improve the performance of chips, to get more instructions executed in less time and area. Moore’s law states that the number of transistors doubles every two years. To increase the performance of a single core chip it is not possible to increase the clock speed much further. The increase of clock speed is limited by the properties of the PN-junction and the time it takes to start conducting. Instead of having more transistors per core, it is possible to get more cores per chip.

For instance 4 times the number of cores should result in almost 4 times as fast execution. In reality when the work is divided between 4 cores, it might be that tasks share some data with each other. We use the term ”resources” to denote the data that is being shared between tasks. A task has to wait for a resource which is in use by another task to become available. A core on which a task is waiting for the release of a resource will lose valuable cycles, if the waiting task does not progress but still is running on the core. One solution is to let the core on which a task is waiting for a resource to be released, execute another task in the meantime. The cycles of the core are not wasted waiting in this way.

Spin-based blocking. Typically a task that becomes blocked on a resource raises its priority to a non-preemptive priority. The task registers itself in the resource queue and waits for the resource to become available. For the complete duration that the resource is unavailable the blocked task remains in execution (running state) on the core. When the requesting task becomes the task at the head of the resource queue and the resource is released, the requesting task acquires access and task locks the resource.

Suspension-based blocking. Typically a task that becomes blocked on a resource releases the core. The task registers itself in the resource queue and waits for the resource to become available. For the duration that the resource is unavailable, other tasks can execute on the core. When the requesting task becomes the task at the head of the resource queue and the resource is released, the requesting task acquires access to the resource. The task raises its priority to a non-preemptive priority and locks the resource.

However, switching between tasks takes time. In order to let the core not waste CPU cycles, the state of the executing task needs to be stored and the data of the new task that is selected to execute on the core needs to be fetched from the memory. Thus the switching causes overhead. Switching the tasks instead of wasting the CPU cycles while spinning can decrease the worst-case performance. Let us assume that a task $\tau_1$ gets blocked on a resource $R_1$ and suspends (leaves the core while waiting). Another task $\tau_2$ with a lower priority than $\tau_1$ will start to execute after $\tau_1$ is suspended. If the new task $\tau_2$ enters a non-preemptive critical section, then it becomes impossible for task $\tau_1$ to continue its execution the moment resource $R_1$ becomes available before $\tau_1$ finishes its critical section. This implies that $\tau_1$ will incur an extra delay under such a scenario and its finishing time will be extended.

Switching a task (while it is blocked on a resource and is waiting) might give rise to larger worst-case finishing times. This can cause a high priority task to miss its deadline and leads to an unwanted situation.
CHAPTER 1. INTRODUCTION

One recent work [2] has proposed a solution to increase the performance of resource sharing protocols. The proposition is to use an intermediate protocol instead of suspending a task or non-preemptively busy-waiting. When a task spins non-preemptively it raises its priority higher than that of any priority on the core. Under this protocol suspending of a task is also seen as busy-waiting but with a priority lower than that of any task on the core. By such a view, the new protocol has suggested that whenever a task has to wait for release of a resource it spins but can use any arbitrary priority between these two extremes. They have called this model as flexible spin-lock model. In this work we aim at implementing a specific set of spin-lock protocols from the introduced range of spin-lock priorities for a core on a multi-core platform to measure the overheads introduced under each protocol.

1.1 Background

Every few years the number of transistors per area of silicon doubled. All of these additional transistors should result in more computation power. If the transistors become smaller, then an equal amount of transistor could fit in a smaller area. When the area becomes smaller, also the diameter gets smaller and the largest distance from one end of the chip to the other end gets reduced. The time the electrical signal takes to propagate over a smaller distance is shorter. Also charging the capacity of a smaller transistor can be done in less time. For decades there was a development of smaller transistors and meant that chips could run with a higher frequency.

The energy consumed in a transistor can be computed by the following formula: $P = C \times V^2 \times f$. To be able to increase the frequency we need more (threshold) voltage. An increase in frequency would mean a more than linear increase in power. At some point we cannot increase the frequency any further since the consumed energy generates heat and can harm the chip.

Figure 1.1 illustrates the relation between the frequency and consumed energy for a different number of cores. A single core chip running at 2GHz has an energy usage of around 14Watt. When we double the computation power we can either double the frequency or double the amount of cores. A single core chip of 4GHz already uses 38Watt, while a dual-core chip running 2GHz only uses 22Watt. It can be imagined that running a single core on 8GHz would take a lot more energy than a four core chip on 2GHz.

![Figure 1.1: Frequency vs energy for 1-4 cores](image)

There are a few options to use the exponential growth of transistors to increase the computation power. A common way of scaling up is to make more complex instructions. More complex instructions makes it possible that in each cycle an instruction is performed that contains more work (CISC, complex instruction set computing). Instead of a 32 bit instruction a 128 bit instruction can be used. The additional transistors can instead be used to make hardware accelerators. Examples of hardware accelerators are a floating point processor, or matrix computing coprocessor (MCC) [4]. These are useful developments but cannot keep up with the exponential growth of additional transistors. What can be observed is the trend to use multi-core chips.
Real time systems are computing systems of which the correct behaviour is not only relied on the correctness of the output but also the moment in time in which it is produced. An example of such systems is the control system in a car to automate the release of the airbag in case of a collision. It is required that the airbag is released before the person crashes into the steering wheel.

In real time systems we have the concepts of hard and soft real time systems. A hard real time system has deadlines for the execution of tasks where missing the deadline would render the output useless. In a soft real time system the output is most beneficial before a certain time, after that time the usefulness gets decreased over time.

A computing system executes machine instructions that are programmed in a higher level language. The program that is being executed is often called a processes or a thread. In real time systems the process or thread is referred to as a task. When multiple processes or threads run on the same processor we need to make sure that these tasks get executed in sequence. A scheduler prescribes when each task should be executed. The scheduler makes use of a protocol to define the order and time of the execution of tasks. When the real time system consists of multiple cores it may be the case that the scheduler also divides the tasks over the cores.

The processes that run on these cores might share some peripherals. For instance two cores share one screen to output information. The access to the screen needs to be regulated to prevent both cores using the output at the same instance. A resource can be a data-structure, a set of variables, a main memory area, a file, a piece of program, or a set of registers of a peripheral device. A resource dedicated to a particular process is said to be private, whereas a resource that can be used by more tasks is called a shared resource. The access to a shared resource protected against concurrent accesses is called an exclusive access. To ensure consistency of the data-structures in exclusive resources, any concurrent operating system should use appropriate resource access protocols to guarantee a mutual exclusion among competing tasks. A resource that is shared among cores is called a "global resource". A resource that is not shared among cores is called a "local resource".

Consider two tasks $\tau_1$ and $\tau_2$ that share exclusive access to a resource $R$. If preemption is allowed and $\tau_1$ has a higher priority than $\tau_2$, then $\tau_1$ can get blocked in the situation depicted in Figure 1.2. Here, task $\tau_2$ is activated first, and, after a while, it enters the critical section and locks the semaphore. While $\tau_2$ is executing the critical section, task $\tau_1$ arrives, and, since it has a higher priority, it preempts $\tau_2$ and starts executing. However, at time $t_2$, when attempting to enter its critical section, it is blocked on the semaphore and $\tau_2$ is resumed. $\tau_1$ is blocked until time $t_3$.

![Figure 1.2: Higher priority task is blocked on a resource](image)

When an additional task $\tau_{\text{new}}$ with a priority between $\tau_1$ and $\tau_2$ would arrive between time $t_2$ and $t_3$, the task $\tau_{\text{new}}$ would preempt the execution of lower priority $\tau_2$. The result is that task $\tau_{\text{new}}$ executes while task $\tau_1$ with a higher priority needs to wait. When a lower priority task is executing while a higher priority task such as $\tau_{\text{new}}$ in this example is available a priority inversion occurs. The priority inversion can be prolonged by execution of any intermediate priority task thus causing $\tau_1$ to miss its deadline. This scenario is referred to as unbounded priority inversion for task $\tau_1$ and should be avoided under a resource sharing protocol.
A solution for the unbounded priority inversion problem was first solved by Lehoczky et al [5] by use of the Priority Inheritance Protocol (PIP). The trick that they use is changing the priority of the task that causes the priority inversion such that the intermediate priority tasks are prevented from execution while a higher priority task is blocked. The moment a task blocks another task, the blocking task gets its priority raised to the highest priority of all the tasks it blocks. That way no intermediate priority task can interfere. The worst-case blocking time gets reduced and unbounded priority inversion due to intermediate tasks is prevented. In a context with resource sharing, priority inversion cannot be entirely removed.

Another protocol invented by the same people Sha, Rajkumar and Lehoczky [5] is the Priority Ceiling Protocol (PCP). The principle is the same as the prior PIP protocol. To make sure that multiple tasks cannot get blocked by a single task: they extended the PIP protocol by only allowing entering a critical section if no semaphore is locked which can block the task. The effect is that a task can never be blocked by lower priority jobs once it enters a critical section.

A further extension of the protocol has been proposed by Baker [6] and is called the Stack Resource Policy SRP. The main differences compared to PCP are the following three points:
1. It supports the use of multi-unit resources (a grouping of resources).
2. It allows dynamic priority scheduling.
3. It supports sharing runtime stack-based resources.

For each multi-unit resource $R$ there is a fixed number of units in the system. "request($R_j$, $m$)" means the job is requesting $m$ units of resource $R_q$.

The difference between PCP and SRP is the moment that a task gets blocked on a resource. With the PCP protocol the task gets blocked when it requests the resource. With the SRP protocol the task gets blocked the moment it wants to preempt another task. In SRP a core cannot start the execution of a task if a resource it uses is locked, even if there is some work to be done before it needs the resource. This reduces context switching.

All the before mentioned resource access protocols did only use a single processor. For multi processor platforms these protocols where extended. The Multiprocessor Stack Resource Policy (MSRP) was introduced in [7]. MPCP is a variant of the Priority Ceiling Protocol (PCP) [8] for multi-core platforms. A comparison between MSRP and MPCP is made in [7].

In [2] it is shown that the suspension-based protocol can also be viewed as spinning with the lowest priority level on the core. Spinning with the lowest priority is denoted as LP and is in the literature referred to as MPCP. Spinning with the highest priority is denoted as HP and is in the literature referred to as MSRP. By choosing an arbitrary spin-lock level between LP (Lowest Priority) and HP (Highest Priority) we can achieve a flexible spin-lock model. Paper [2] also introduces two specific spin-lock protocols: CP (highest local Ceiling of global resources Priority level) and OP (Own Priority level) spin-lock protocols. The final resource locking technique of which we measure its performance is $\hat{CP}$. $\hat{CP}$ has been presented in [3]. In the paper it is also shown that $\hat{CP}$ dominates the classic HP spin-lock protocol and all spin-lock priorities in between. The focus in [3] is on a particular class of spin-lock protocols from the flexible spin-lock model [2]. The class assigns a spin-lock priority higher than or equal to the CP spin-lock protocol.
1.1.1 Both suspension and spin-based blocking can be preferable

The purpose of this subsection is to illustrate example case scenarios where a suspension-based approach and a spin-based approach can outperform each other.

Figure 1.3a illustrates the scheduling trace of a task set under both a suspension and spin-based protocol. As soon as \( \tau_2 \) on core \( P_1 \) needs the resource that is already in use by core \( P_0 \), it starts spinning non-preemptively under the spin-based protocol. The waiting is wasting cycles. These cycles could be used to process task \( \tau_1 \) as shown under the suspension-based protocol, illustrated in the same figure. The cycles the spin-based protocol wastes waiting are used for processing under the suspension-based protocol, thus the utilization of the core is higher under the suspension-based case.

Figure 1.3b illustrates the situation in which a lower priority task \( \tau_2 \) starts executing when task \( \tau_1 \) becomes blocked. Task \( \tau_2 \) enters a critical section and becomes non-preemptable. Task \( \tau_1 \) cannot continue its execution the moment the shared resource becomes available. Thus, the finalisation time of \( \tau_1 \) is delayed and it might miss its deadline. The task set might not be schedulable under the suspension-based protocol.

These two scenario’s show the benefits of the spin- and suspension-based protocols. A flexible spin lock model suspends in some cases and spins in others, allowing a trade-off between utilization of cores versus delaying the worst-case finalisation time.

Let \( f_{sup}^i \) denote the finalisation time of task \( \tau_i \) under suspension-based protocol. Let \( f_{spin}^i \) denote the finalisation time of task \( \tau_i \) under spin-based protocol. Note that in [7] a comparison is made between MPCP and MSRP. The conclusion is that the schedulability of MPCP vs MSRP depends on several properties. We can compare the finalisation times of Figures 1.3a and 1.3b. Figure 1.3a illustrates that suspension-based protocol performs better in the sense of using the processor cycles. Figure 1.3a illustrates \( f_{sup}^1 << f_{spin}^1 \) and \( f_{sup}^2 << f_{spin}^2 \), thus the task set might be schedulable under the suspension-based protocol and not under the spin-based protocol.

Figure 1.3b illustrates that spin-based protocol performs better in the sense of incurred extra delays to tasks. Figure 1.3b illustrates \( f_{spin}^1 >> f_{sup}^1 \) and \( f_{spin}^2 << f_{sup}^2 \) thus task \( \tau_1 \) might miss its deadline under the suspension-based protocol.
CHAPTER 1. INTRODUCTION

1.2 Problem statement

In [2], it has been shown that under the worst-case schedulability analysis, response times of tasks may change under different variants of flexible spin-lock protocols. This worst-case analysis implies that the schedulability of the task set may change under each protocol. It is shown in [3] that response times of tasks can be improved under variants of the flexible spin-lock protocols compared to HP that only allows spinning with the highest priority.

When it comes to a real implementation, each protocol may introduce system overhead that can affect the schedulability. Such system implementation overheads have not been considered in system analysis in [2] and [3]. However, in order to bridge the gap from theory to practice for system analysis, it is important to identify and include such overheads in the analysis. The first step towards this purpose is to implement these locking protocols on a multi-core platform. Preferably an OSEK/AUTOSAR compliant RTOS on a platform with at least four cores. Next step is measuring the induced overheads under each of the resource access protocols.

We have identified two types of overheads: the time and memory that is introduced under each of the aforementioned protocols. When a shared resource is locked on a processor, other cores cannot use the resource until the resource is released. Additional requests to the same resource need to be placed in a queue to be served later. All of this process should be done in such a way that no read/write conflicts can happen. We should strive to implement the resource access protocol rather efficient.

In this project we want to compare the introduced system overhead under different spin-lock protocols. Therefore we would like to know when/where the overhead is incurred such as e.g. upon locking and unlocking, updating the queue’s or during communication between cores. We can measure the system overhead in terms of time and memory by for instance varying the following parameters: the number of processes, the number of shared sources, spin-lock priorities.

1.3 Objectives and Goals

The objective of this thesis is to implement flexible spin-lock protocols on a platform and determine its time and memory consumption compared to a protocol that only allows spinning with the highest priority.

Prior to the master thesis a feasibility investigation was performed. The aim of this project preparation was to orientate on the literature and decide on prerequisites for the research. The outcome was the selection of the board and OS.

The main purpose of the master thesis is determining the improvement of flexible spin-lock protocols over a protocol that only allows spinning with the highest priority. The quality of the implementation has influence on the performance. It is therefore desirable to have a functional correct and a time and memory efficient solution.

To provide the measurement data the following deliverables are required:
- Implementation of flexible spin-lock protocols.
- Operational system, containing resource access by flexible spin-lock protocols.
- Measurement of the time and memory used under each protocol.
- A draft version of a paper for a workshop or conference.
- The master thesis reporting the measured results.

1.4 Thesis Contributions

The thesis provides the following main contributions:
- A specification and implementation design of flexible spin-lock protocols.
- Identify the places where the overhead may occur.
- Time and memory usage measurements of flexible spin-lock protocols.
- Manual for programming hardware and software on FPGA board.
1.5 Reading guide

The document contains a lot of content. Here we provide a suggestion on how to proceed reading.

**Limited time:**
- Chapter 1 Introduction p. 1 - 8.
- Chapter 5 ERIKA, Section System overview p. 20 - 24.
- Section 5.6.3 Resource access: Introduction to MSRP p. 35
- Chapter 8 Spinning on a local spin_lock, Sections Introduction and Measurement p. 62 - 63 and 77 - 87.
- Chapter 9 Context switch, Sections Introduction and Measurement p. 88 - 89 and 105 - 107.
- Chapter 10 Conclusions p. 109.
**An total of 32 pages.**

**Regular time:**
- *Suggestion to read:*
  - Chapter 1: Introduction p. 1 - 8
  - Chapter 2: Related Work p. 9 - 17
  - Chapter 3: System Model p. 17 - 18
  - Chapter 4: Feasibility Study p. 19
  - Chapter 5: The ERIKA OS p. 20 - 37, 42 - 50
  - Chapter 6: The project p. 51 - 54
  - Chapter 8: Spinning on a local spin_lock p. 62 - 73, 77 - 87
  - Chapter 9: Context switch p. 88 - 101, 105 - 107
  - Chapter 10: Conclusion and Future Work p. 109

- *Suggestion to skip:*
  - Chapter 5: The examples of spin_in() and spin_out() in ERIKA p. 38 - 41
  - Chapter 7: Flexible priority tool 55 - 61
  - Chapter 8: Spinning on a local spin_lock, Section Check Invariants p. 74 - 76
  - Chapter 9: Context switch, Section Check Invariants p. 102 - 104

**An total of 92 pages.**

1.6 Research Questions

**Question 1** How much time and memory do flexible spin-lock protocols consume compared to a protocol that only allows spinning with the highest priority.

*Hypotheses*

The analysis of the protocols indicates that the flexible spin-lock protocols dominates some other protocols. The analysis abstracts from reality and makes use of models. The models do not take the overhead of implementation into account. This overhead has influence on the time and memory usage of the protocols. Since it is possible that sometimes flexible spin-lock protocols reduce the time spend spinning compared to a protocol that only allows spinning with the highest priority, scenario’s that have a lot of global resource sharing and long access times might benefit most. We need resource queues for every implementation/protocol. Therefore we expect only a slight difference in the memory usage.

**Question 2** Where is the overhead induced in the code?

*Hypotheses*

We need to keep track of the tasks waiting on a resource in the resource queue. The updating of these queue’s need to happen atomically (as if it was a single instruction without being interrupted). Also we expect that notification between cores takes a lot of time.

**Question 3** Can we make a model that takes the overhead into account?

*Hypotheses*

Perhaps it is possible to add constants to the model representing the time required for: updating the queue, switching between tasks and inter-core communication (assuming those are the contributors to time overhead).
CHAPTER 1. INTRODUCTION

1.7 Thesis Outline

Chapter 1: Introduction
First we introduced the research context, gave a motivation for the work and research questions. To get some basic understanding of resource access protocols we provide some background. Additionally an overview of the thesis contributions is presented.

Chapter 2: Related Work
We give an overview of the related work in this chapter. When considering existing implementations, the main work will be the implementations of of spinning-based and suspension-based protocols. Some information from papers is presented contains measurements other researcher performed on resource access protocols. Part of the related work chapter provides information on Synchronization Algorithms for Shared-Memory. We also provide some papers related to standards since we use an OSEK-compliant and AUTOSAR like OS.

Chapter 3: Feasibility Study
In this chapter we give a brief overview of the work performed prior to the actual thesis project. During the feasibility study we decided on the RTOS and development board. We came to the design decision to use the ERIKA OS and program it on a FPGA board.

Chapter 4: The ERIKA OS
An overview of the kernel-code is presented in this chapter. Some figures contain call graphs that illustrate the global working. The main kernel has 6 clusters of functions: initialization, task handling, ready queue, stack and context switch, resource access and remote notification. For implementation of the different protocols we need to make changes to Remote notification and Resource access functions.

Chapter 5: The project
In this chapter we give an overview of the changes we need to incorporate to complete the project. The work is split up in three steps (the following chapters) i.e., Spinning on a local spin_lock, flexible priority tool and the context switch. In each of those development steps is the v model used. If possible the measurement results of the implemented step are given.

Chapter 6: Flexible priority tool
To be able to make the new RAP we need to change the priority of the spinning task. The flexible spin-lock protocol assigns a priority to tasks which are not yet present in the ERIKA OS. We need to provide those priorities and decided to develop a tool to derive those. Whenever a task gets blocked on a resource the task starts spinning on the spin priority instead of the non-preemptive priority.

Chapter 7: Spinning on a local spin_lock
To make the new RAP we need to notify the release of a resource to a waiting core by means of an interrupt. A spinning task might get preempted by a higher priority task in the flexible spin-lock protocol. The preempted task is not actively spinning on the spin-lock and should get notified in a different manner. To achieve this the release of the resource should get notified to the requesting task via an interrupt. In this chapter we take the first step towards the flexible spin-lock protocol by implementing the remote notification of a released resource.

Chapter 8: Context switch
A blocked task spins on a spin-priority. When a higher priority task arrives the spinning task should get suspended. To achieve the context switch we need to adjust the system ceiling to correct priorities of tasks throughout the operation of the system. The tasks should use more than one stack to allow the additional preemption and resuming of tasks.

Chapter 9: Conclusion and Future Work
The analysis suggested that the flexible spin-lock protocol would dominate MSRP/HP. The measured overhead of queue accesses and context switches are substantial. MSRP/HP works in practice quite reasonable due to the lack of overhead. Still the flexible spin-lock protocol can probably schedule a set of tasks that could otherwise not be scheduled.
Chapter 2

Related Work

The purpose of our project is to implement and measure the time overhead of resource access protocols. Here we present related work about: resource access protocols, how to measure them, implementations in Linux and AUTOSAR, hardware (to notify the release of a resource) and spin-locks.

Resource access protocols for multi-core systems where first described by Rajkumar et al. in [9]. The protocol they presented was a multi-core variant of the priority ceiling protocol (PCP). The work presented by Chen and Tripathi in [10] contained resource access for systems using P-EDF (partitioned earliest deadline first) for periodic tasks. In [11] by Lopez et al. and [7] by Gai et al. are protocols described that also allow sporadic tasks, but do not allow nested resource access. Multi-core resource access protocol for G-EDF (global earliest deadline first) was described by Devi et al. in [12]. The flexible multiprocessor locking protocol (FMLP) is described in [13] does allow nested resource access.

In PCP whenever a job $J_{i,j}$ requests access to a resource $R_q$, its priority $\rho_{i,j}$ is compared to the priority the system is operating on i.e., the system ceiling. If the priority $\rho_{i,j}$ of job $J_{i,j}$ exceeds the system ceiling is the request handled, otherwise the job $J_{i,j}$ suspends. The SRP protocol handles the access requests immediately. Blocking only occurs when a job releases a resource. The releasing job $J_{i,j}$ may not execute after its release until its priority exceeds the system ceiling.

The Priority Ceiling Protocol (MPCP) proposed by Rajkumar et al. in [9] is a multi-core resource access protocol. The protocol extend PCP i.e., a task executes critical sections on the highest priority among tasks that share the resource. The extension of MPCP is that a task that is blocked on a global resource gets suspended. A higher priority task preempting the blocked task is allowed to execute local and global critical sections. When the resource becomes available the preempted task is notified and increases its priority the the priority of the resource ceiling which is higher than any priority of tasks that use the same resource.

The Distributed Priority Ceiling Protocol (DPCP) proposed by Rajkumar et al. in [14] is a multi-core resource access protocol. A resource is assigned to a certain core (the core is the agent of the resource). When any task on a remote core requests access to the resource an RN remote notification is send to the agent. The agent keeps track of which core is accessing and requesting the resource. The suspension and priority properties are the same as MPCP. DPCP is useful in distributed systems where processors do not share memory.

The Flexible Multiprocessor Locking Protocol (FMLP) proposed by Block et al. in [15] is a multi-core resource access protocol. FMLP is a resource access protocol that is rather similar to the protocol that we want to implement. FMLP also usages spin- and suspension-based resource access. All shared resources are divided into two groups with either a short or long resource access duration. The benefit is that in case the access takes a long time it makes more sense to suspend and perform another task in the meanwhile. While in case the resource is accessed for only a short duration the additional cost of context switching might take more than the resource access time. They claim that they provide three benefits:

1. Non-nested resource access is more efficient
2. The resource access protocol can be applied in global and portioned scheduling EDF.
3. It supports nested resource access
2.1 Measuring resource access protocols

We provide some related work that already compared multi-core resource access protocols. The set-up they use can be of interest to our project, such as the duration of critical sections and speed of the processors.

A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform is performed by Gai et al. in [7]. MPCP and MSRP are resource access protocols comparable with LP and HP in [2]. The comparison is done on a dual core processor. The simulated task sets contain a random task set of which both implementations performed better in parts of the spectrum. The second test was an application specific test about a power train controller where MSRP had clearly better results. The resources are protected by priority ceiling (MSRP or MPCP) semaphores. Assumed critical section times: 5 and 50 µs (relative short compared to the critical section duration of the comparison in [16] and in favour of MSRP).

In [7] the implementation of the MPCP protocol is divided into two parts:

1. Implementing the PCP protocol locally and implementing global synchronization between cores. A task queue can be used to implement the PCP protocol.
2. The inheritance of priorities is implemented by keeping track of the semaphores that are locked (ordered by ceiling).

The synchronized access between cores is handled with a global mutex that protects shared data. The access to the resource is ordered in a resource queue. The period enforcer technique is used to bound the blocking time. In [7] it is stated that the period enforcer has not been described in published work (nor is a complete description available online). In [7] the implementation of the MSRP protocol does not need semaphores nor queues for blocked tasks. The SRP allows a single stack implementation. The implementation in a 2 core set-up means that they do not need a queue for resource access since there can be at most one other waiting task/core. There is a fast synchronise instruction swpb.

The communication speed of the serial bus (UART) is considered to be 500kb/s resulting in a access time of 50 µs. The number of local resources is always 6 for each processor, plus 6 global resources. The utilisation ranges from 0.025 to 0.925. Tasks have harmonic periods. The number of critical sections accessed per task is [0,6]. MPCP performs better if most resource access is to global resources while MSRP performs better in case of local resources. In the power train measurement the MSRP outperforms the MPCP. The critical sections are short compared to the execution times of the tasks. 20% of the execution time of high rate tasks is spent in critical sections. Thus the power train measurement has a lot of accesses of a short duration. They suggest that a lock free algorithms should be used as much as possible in situations with high rate and short duration critical sections.

The memory utilization of real-time task sets in single and multi-processor systems-on-a-chip is minimized by Gai et al. in [17]. The paper presents a fast and simple algorithm for sharing resources in multi-core systems; MSRP. A choice for resource access and scheduling protocol can be a trade-off between memory and time. Selecting non-preemptive scheduling can reduce the stack memory while it reduces the processor utilization compared to preemptive scheduling. In this paper the pseudo-resource is mentioned. If we want task \( \tau_i \) and \( \tau_j \) to be mutually non-preemptive we can let them share a pseudo resource \( R_q \). This principle can be used to increase a priority temporarily by using dummy shared resources. The shared resource does not contain any information and its sole purpose is to raise the priority when the section is entered.
A comparison of MPCP, DPCP and FMLP on LITMUS$^{RT}$ is performed by Brandenburg et al. in [16]. The measurements use critical section length [1µs, 25µs] (short), [25µs, 100µs] (medium), or [100µs, 500µs] (long) on four 2.7 GHz processors. Their context switch takes 9.25 µs, the FMLP acquisition and release take 2.74, 8.67 µs, the MPCP acquisition and release take 5.61, 8.26 µs. In MPCP and DPCP blocked tasks get suspended. In DPCP resources are assigned to cores, which is not the case in MPCP. In DPCP a task accesses a resource via an RPC-like invocation (remote procedure call) to the processor assigned to the resource. In MPCP a global semaphore protocol is used instead. In FMLP the requests are ordered in FIFO order. Spinning waists cycles, still it is considered to provide better results since suspension analysis tends to be pessimistic. Deadlock is avoided in DPCP and MPCP by prohibiting the nesting of global resource requests. They conducted schedulability experiments assuming 4 to 16 cores and used overhead values obtained from the 4 core test platform. Overall, the long FMLP variant exhibited the best performance among suspension-based algorithms, especially in low-sharing-degree scenarios.

An implementation of the PCP, SRP, DPCP, MPCP, and FMLP is discussed by Brandenburg and Anderson in [18]. The multi-core resource access protocols DPCP, MPCP, and FMLP use PCP and SRP for access to local resources. Both PCP and SRP protocols determine a priority ceiling i.e., the highest ceiling among jobs that use a resource.

Tasks and jobs can be implemented in a system in three ways as described in [18].

1. A process represents the task and a thread represents a job.
2. A thread represents the task and one iteration through a loop of the task represents a job.
3. A task is just a concept and the code executed as a result of an interrupt routine is a job.

A kernel that allows tasks to suspend when requesting access to a resource requires support of the kernel. The kernel must maintain the priority ceilings and apply the required inheritance of priority. The resources are modelled in the kernel as an object. The object resource has for each instance R its state ∈ {locked, unlocked}, the priority ceiling, a queue of jobs requesting access. In their implementation they used the filesystem in such a way that allows to control access to the resource. In LITMUS$^{RT}$ the object instance of a resource is an inode. The kernel uses inodes as representation of files on the filesystem. By managing these inodes the synchronization between processes is achieved. Each thread has a control block (TCB) that contains a table which stores the requests for access to a resource R. Comparable to the operation of a file descriptor table, is a lookup table for resources. Priority ceilings are determined off-line or on start-up. When a thread gets suspended it is places in a waiting queue, in case of FMLP the ordering is FIFO, for M-PCP and the D-PCP is the ordering priority based. An average request or release takes 0.5µs at 2.7 GHz.

Waiting algorithms for synchronization in large-scale multiprocessors are presented by Lim and Agarwal in [19]. An algorithm called two-phase waiting is presented in is given. A job first spins when it gets blocked on resource and then suspends after a while. The time after which the task suspends is varied. The costs of preemption and resuming suspended tasks determines at which time it becomes beneficial to suspend. Most likely, the set-up was not meant for hard real-time systems. The paper does provide insight in the behaviour of protocols which both spin and suspend.
2.2 LITMUS$^{RT}$

Resource access protocols are a part of the kernel of an OS. An RTOS where a lot of the related work is performed on is LITMUS$^{RT}$. Both the implementation of protocols and the measurement set-up can provide useful insights for our project.

Real-time synchronization on multiprocessors and whether to block or not to block, to suspend or spin is discussed by Brandenburg et al. in [20]. The authors of the paper are also the developers of LITMUS$^{RT}$ which is a real time OS. LITMUS$^{RT}$ is a Linux kernel that uses real time libraries and scheduling policies to execute and preemt its tasks. In the paper they present two contributions:

1. Their implementations of resource access protocols.
2. A comparison between those implemented resource access protocols.

The access to a resource is synchronized by either lock-based and non-blocking based protocols. If the resource is a shared data object then it is possible to use non-blocking protocols. In case the resource is not a shared data object i.e., some shared hardware, the task locks the resource preventing other tasks to access it at the same time. A task requesting access to a locked resource gets blocked. The blocked task can proceed by either spinning (busy-waiting) or suspending (letting other tasks process instead). They determine four access technics:

1. lock-free, an object access is attempted in a loop and retried until successful.
2. wait-free, an object access is done by sequential execution of code (concurrent access can not occur)
3. spin-based, continuously checking of the availability of the resource (FIFO ordered).
4. suspension-based, letting another eligible task execute while the resource is unavailable.

The paper compares those 4 techniques, and determines in which situation a certain technique should be used. The conclusions of the paper are:

1. Lock-free protocols are preferable for small (non-complex) objects.
2. Wait-free and spin-based protocols are preferable for complex objects.
3. Wait-free is preferable to lock-free (probably not always possible).
4. Deep nested resource access or frequently occurring long resource access is poor under any protocol.
5. Suspension based blocking should not be used in combination with P-EDF for global resources.
6. They claim that spin-based is generally preferably over suspension-based protocols, unless tasks access at least 20% of their execution a critical section. (they disregarded very long resource access such as to external memory).

A status report on LITMUS$^{RT}$ is provided by Brandenburg et al. in [21]. In the paper a more detailed description is provided on how LITMUS$^{RT}$ operates. The development of LITMUS$^{RT}$ focusses primarily on: the scheduling algorithms as a plugin to Linux, synchronization mechanisms and libraries that allow to schedule a real-time task set. Of our particular interest is the resource access implementation. The resource access needs to be synchronized. Short resources are accessed using queue locks and long resources are accessed via a semaphore protocol.

In spin-based blocking the synchronization often takes place via locking mechanisms. A queue lock is a FIFO ordered spin-lock. The spinning on the spin-lock is often done locally i.e., the local data is updated with coherent caches or distributed shared memory. The local spin-locks are implemented without much interaction of the kernel-layer. The FMLP uses spin queues to implement the spin-based blocking, the Linux kernel often uses unordered spin-locks. The non-ordered access adds unpredictability and pessimism to the analysis. They deemed it unfeasible to make all spin-locks ordered.

In suspension-based blocking complex kernel-layer synchronization protocols are used via OS-provided system calls. An important property in selecting the synchronization primitive is the analyzable blocking behavior. They did not consider mechanisms of which the analysis was not available. There exists a real time library that provides shared objects i.e., libso. The library provides services to the kernel layer that allows for synchronization. The function mmap(2) is a system call that abstracts the shared objects and provides process naming and in-object memory management. The libso library uses the MCS algorithm described in [22] to implement the queues and a flag-based mechanism to denote non-preemptive sections.
2.3 OSEK and AUTOSAR

Automotive is a field where real time scheduling and resource access is common practice. Therefore is chosen to implement the protocols we want to measure in an OSEK compliant RTOS. OSEK is a standard defining properties about the RTOS used in automotive. Here we present work that is related to resource access in an OSEK/AUTOSAR RTOS.

Porting an AUTOSAR-Compliant operating system to a high performance embedded platform is presented by Zhang et al. in [23]. Their choice of RTOS is ArcCore\(^1\) and the board is a Raspberry Pi\(^2\). AUTOSAR is a layered software architecture that decouples the application software (ASW) from the lower level basic software (BSW) by means of a standardized middle-ware called runtime environment (RTE). They make use of a single core chip. The porting process and kernel development process is divided into 4 steps; initialization (exception vectors), memory modelling, exception handling and context switching. The next part of paper focusses on the development process of a SPI\(^3\) driver according to AUTOSAR requirements, as an example on how to develop a driver. These steps of porting are relevant for considering the choice of platform.

An overview of AUTOSAR multicore operating system implementation is presented by Devika and Syama in [24]. OSEK and AUTOSAR are standards in the automotive industry. The OSEK consists of various parts; OS (services of the real time kernel), COM (communication services), NM (network management services) and OIL (OSEK implementation language). The OS services are: 1) task management, 2) conformance classes, 3) scheduling policy, 4) task synchronization, 5) resource management, 6) alarms and counters, 7) communication, 8) interrupts, 9) interprocess communication, 10) task management. The document is relevant to understand the AUTOSAR standard.

The paper [24] also provides some information on AUTOSAR and spin-locks specifically. In single-core processors there is a function called GetResource(). When a task acquires a resource its priority is temporarily raised to the highest among all tasks that use the resource. GetResource() uses the immediate priority ceiling protocol, which ensures that the priority inversion is no more than the duration of a single resource-holding duration. This principle does not work on multi-core systems. The extension of AUTOSAR \(^4\) [25] contains a new mechanism for mutual exclusion. This is a busy wait mechanism that polls a (lock) variable “S” until it becomes available using an atomic test and set functionality. Once a lock is obtained by a task, other tasks cannot access resource “R” until the lock “S” is released. The paper presents a possible deadlock and starvation. The deadlock can happen when a task spins on a low priority. A higher priority task preempts the spinning task and at some point requires the same resource. A more detailed description can be found in the paper [26].

Mechanisms for guaranteeing data consistency and flow preservation in AUTOSAR software on multi-core platforms are presented by zeng et al. in [27]. The paper describes communication mechanisms and tradeoffs on multi-core platforms. They provide a way of synchronising the resource access that is of interest to our implementation. The release of a resource must be notified across cores. In the paper they require flow preservation. In the example they used a core writes a variable and another core reads the variable. The order of reading and writing should be preserved. They propose two solutions: 1. enforcing synchronization 2. wait free flow preservation. In case of enforcing synchronization the writing core should notify the reading core that the data is available. They state that an activation signal can be sent between cores (inter processor interrupt signal). Often the RTOS provides functions or services that allow for communication and synchronization between tasks also called interprocess communication (IPC) such as semaphores and message queues. The event management of the RTOS keeps track of the state of tasks i.e., which task awaits an IPC to indicate the resource availability. They also provide a comparison between MPCP, MSRP and the wait-free method.

---

\(^1\)ArcCore is a RTOS developed for an embedded platform to be used in the automotive industry.
\(^2\)Raspberry Pi is an affordable computer that you can use to program embedded applications.
\(^3\)SPI: Serial Peripheral Interface, is a bus used for synchronous serial communication.
Spin locks in AUTOSAR: blocking analysis of FIFO, unordered, and priority-ordered spin locks is presented by Wieder et al. in [28]. In the paper are different spin-lock protocols described that are used in AUTOSAR. The resource access technique that is mandated by the AUTOSAR standard is spin-based [25]. The authors study four queue orders and two preemption models. The four queue orders used in the spin-based resource access are (i) FIFO-ordered, (ii) unordered, (iii) priority-ordered with unordered tie-breaking, and (iv) priority-ordered with FIFO-ordered tie-breaking. Each of the queue orders has been analyzed assuming both preemptive and non-preemptive spinning.

2.4 Spin-lock

The major research trends since 1986 in shared-memory mutual exclusion are presented by Anderson et al. in [29]. In resource access protocols that use local spin-locks the spinning is implemented by means of polling in a loop. The loop repetitively checks the spin variable. Such spin variables must be locally accessible, i.e., they can be accessed without causing message traffic on the processors-to-memory interconnection network. Two architectural models are presented by the literature that provide functionality of locally accessible shared data: distributed shared-memory (DSM) machines and cache-coherent (CC) machines.

The performance of spin lock alternatives for shared-memory multiprocessors are presented by Anderson in [30]. Resource access protocols either spin or suspend. When overhead is taken into account it is often more efficient to spin on a resource instead of suspending. For small critical sections compared to the overhead time it is beneficial to spin. Spinning on a resource by polling the availability can cause contention on the bus or communication bandwidth (depending on implementation). Questions the paper tries to answer are there atomic hardware primitives that allow algorithms to efficiently spin-wait on a resource, or is more complex hardware required. They compare the spin-lock problem with acquiring access to a network and apply the already existing solutions to spin-locks.

They propose a method themselves that assigns a sequence number to an access requesting core. Parameters that have a large impact on the performance is the connection to the memory and whether a core has hardware controlled coherent cashes and the policy of the cash. In case of a coherent cash, the core spins on the cash and not on the bus. Only when the value is updated in cash is the test and set instruction transmitted on the bus. They conclude that for cores without extensive hardware control software queuing and back off spin locks perform best (after trying to acquire the lock the core waits some time). Coherent cashes can make the spin-locks more efficient. They take the updating of the cashes in consideration i.e., some bus traffic and delay between release and arrival at requesting core. That a resource is unavailable for one core might be notices by other cores by snooping the bus. Hardware queuing makes the spin-locks more efficient. They question whether the amounts of spin-waiting by the cores is significant enough to justify the additional hardware.
2.5 Hardware and semaphores

The flexible spin-lock protocol makes use of both spin- and suspension based resource access. To implement the FSLM we need to notify the release of a resource e.g., communication in concurrent systems. Here we present some hardware and semaphore related work. A semaphore can denote the whole service an RTOS provides to synchronise access including the resource queue and mutex. Often a semaphore refers to a hardware mechanism that allows mutual exclusive access.

An overview of interprocess communication is presented by Lamport in [31]. The first part of the paper is about formalising the communication between processes. The second part is about the communication between processes in concurrent systems.

A typical communication between cores involves the storage of the message in memory by the sending core. The core sending raises the voltage on the input of the receiving core. A driver of the receiving core notices the change in electrical charge and sets a bit in the register of the receiving core. The receiving core checks periodically i.e., every clock tick the interrupt registers and starts executing the interrupt routine. The interrupt handler checks the memory location in shared memory where the sending core did put its message. Often the process receiving the message can be considered as two tasks:

- A task that registers the change on the input of the core (in our case the interrupt handler)
- A task receiving the message (in our case the task requesting access)

RTOS semaphores and event semaphores are explained by Moore in [32]. A binary semaphore has the values 1/0 denoting locked or unlocked. An access requesting thread can check the value and lock it to prevent additional threads to access the resource while using it. A multiple resource semaphore is used to regulate the access to multiple identical resources. Software defined semaphores and mutexes are services provided by the OS. The c programming language provides a set of interfaces or functions for managing semaphores. Event semaphores provide a method to synchronize tasks with internal or external events and with other tasks. An event semaphore is the occurrence of external event \( A_{\text{event}} \) that causes an interrupt, resulting in codeA() to execute.

Hard real time systems are explained by Buttazzo et al. in [33]. The textbook provides an example and assumes two tasks \( \tau_1 \) and \( \tau_2 \) require access to \( R_q \). A typical access request could be wait(s) and a release signal(s). An executing task requests access to the resource by use of the semaphore calling wait(s). A queue is maintained of tasks that request access to the semaphore. If the resource is unavailable the task enters a waiting state until another task executes the signal(s) function.

An interrupt-generating hardware semaphore is presented by Wenniger in [34]. The overhead of a resource access protocol depends amongst others on the time it takes to notify and detect the release of a resource. Often operations performed by a chip can be performed in either software and hardware\(^4\). The principle presented is not relevant, but interesting for our work. The hardware will generate an interrupt on a resource requesting core when the resource becomes available, eliminating the need to poll to detect the change in availability of a resource.

A method to use hardware semaphores for resource release notification is presented by Emerson et al. in [35]. The patent is a semaphore mechanism in hardware that besides the conventional mutual exclusive access to a resource also contains a resource queue and a notification signal when the resource becomes available. The principle can not be used in existing hardware (as in our project). An integration of mutex, queue and notification in hardware illustrates that any remote notification time measured in our project might take less time in other hardware.

\(^4\)In FPGA in the hardware description, in CPU by designing a new chip
Programming models for hybrid FPGA-CPU computational components are presented by Andrews et al. in [36]. The paper is about hybrid chips containing both a CPU and FPGA\(^5\). In our project we use a FPGA and run a soft-core on it as if it is a CPU, thus the hardware synchronisation of the paper is of our interest. To extend the thread model to the hybrid FPGA-CPU they come up with a thread model which can be considered a thread object. They suggest a spin lock, block lock, spin semaphore and block semaphore. A traditional CPU semaphore used in resource access protocols reads atomic and (mutual exclusive) writes to memory. Instead of providing the same mechanism for the hybrid chip they suggest an alternative way. They come up with a spin lock controller, an IP (Intellectual Property) block that can be used in the hardware design. The IP receives requests by reading its input register. The IP contains a control structure and denies or accepts the requests. The suggested blocking semaphore allows queuing and suspension of threads that do not receive access to the semaphore. When the resource is unavailable a call to the thread API initiates the scheduler and the thread continues either spinning or is suspended. It is clear that integrating the semaphore with a spin lock controller and thread queue manager all in hardware means that the overhead is significantly less. Additionally the contention on the bus is decreased.

The timing Effects of DDR Memory Systems in Hard Real-Time Multicore Architectures are presented by Cazorla et al. in [37]. In a multi-core environment several cores can requests data from the SDRAM memory in parallel. The timings of the SDRAM that become important in such a multi-core scenario are: the Request Execution Time (RET) and the Inter-Task Delay (ITD).

- RET is the part of the response time that is invariant to interference.
- ITD is the part of the response time that is subject to interference.

In [37] they propose an upper bound to the interference of cores on the response time. When the worst case interference is defined by a bound it allows composability of threads independently from co-running threads. The ITD consists of:

- The access to the memory is done by arbitration. The number of time slots a thread has to wait adds to the delay. The implementation of the waiting queue and the arbitration policy determine the delay.
- The duration of the scheduling slot (Issue Delay), is the time between two consecutive requests to be issued. The Longest Issue Delay (tLID) influences the throughput. The tLID consists of:
  - The policy of row buffer management.
  - The timing constrains of the DRAM technology used (i.e., T6 or T8, 6 or 8 transistors used).
  - The address mapping scheme, (two simultaneous requests to the same bank gives bank conflict).
  - The order of issued DRAM requests (i.e., An read command takes significantly less time if the preceding request was also an read command).

In DDR memory (double data rate) both the rising and the falling flank are used to transport data. The memory is divided into banks, those banks can be accessed in parallel. Each bank is divided into rows and columns. Each bank contains a row buffer that cashes the most recent accessed row in the bank. To open a row an ACT command is used. The read and write times are given by CAS and CWD commands. After the operations on a row are completed a pre-charge PRE command stores the data (closing the row). Policies either close the row after each use (by precharging for faster access to different row), or leave the row open (faster access to the same row).

\(^5\)which is nowadays an active topic, Intel(cpu) bought Altera(FPGA) last year.
# Chapter 3

## System model

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_k$</td>
<td>Processor (core) with identifier/ index $k$</td>
</tr>
<tr>
<td>$\mathcal{P}$</td>
<td>Set of processors</td>
</tr>
<tr>
<td>$\tau_i$</td>
<td>Task with identifier/ index $i$</td>
</tr>
<tr>
<td>$T$</td>
<td>Set of tasks</td>
</tr>
<tr>
<td>$T_{P_k}$</td>
<td>Subset of tasks assigned to processor $P_k$</td>
</tr>
<tr>
<td>$T_{P_k,q}$</td>
<td>Subset of tasks assigned to processor $P_k$ that use resource $R_q$</td>
</tr>
<tr>
<td>$R_q$</td>
<td>Resource with identifier/ index $q$</td>
</tr>
<tr>
<td>$\mathcal{R}$</td>
<td>The set of shared resources.</td>
</tr>
<tr>
<td>$\mathcal{RS}_i$</td>
<td>The set of shared resources accessed by tasks $\tau_i$.</td>
</tr>
<tr>
<td>$\mathcal{RS}_i^G$</td>
<td>The set of global resources accessed by tasks $\tau_i$.</td>
</tr>
<tr>
<td>$J_{i,j}$</td>
<td>A job i.e., the $j^{th}$ instance of task $\tau_i$</td>
</tr>
<tr>
<td>$r_{i,j}$</td>
<td>The release time of job $J_{i,j}$</td>
</tr>
<tr>
<td>$c_{i,j}$</td>
<td>The execution time of job $J_{i,j}$</td>
</tr>
<tr>
<td>$d_{i,j}$</td>
<td>The deadline of job $J_{i,j}$</td>
</tr>
<tr>
<td>$f_{i,j}$</td>
<td>The finalisation time of job $J_{i,j}$</td>
</tr>
<tr>
<td>$\rho_i$</td>
<td>The priority of task $\tau_i$</td>
</tr>
<tr>
<td>$C_i$</td>
<td>Worst case execution time of task $\tau_i$</td>
</tr>
<tr>
<td>$\theta_i$</td>
<td>Period of a task (we assume relative deadlines equal to periods).</td>
</tr>
<tr>
<td>$d_{i,j}$</td>
<td>$r_{i,j} + \theta_i$.</td>
</tr>
<tr>
<td>$e_{ik}$</td>
<td>Is the $k$-th critical section of task $\tau_i$ on resource $j$</td>
</tr>
<tr>
<td>$\omega_{ik}$</td>
<td>It the maximal duration of the $k$-th critical section.</td>
</tr>
<tr>
<td>$B_i$</td>
<td>Blocking time of task $\tau_i$</td>
</tr>
<tr>
<td>$\lambda_i$</td>
<td>Static preemption level task $\tau_i$, $\tau_i$ is only allowed to preempt $\tau_j$ if $\lambda_i &gt; \lambda_j$</td>
</tr>
<tr>
<td>$\Delta_\alpha$</td>
<td>Measured overhead term with identifier/ index $\alpha$</td>
</tr>
<tr>
<td>$\Theta_\alpha$</td>
<td>Worst case overhead term with identifier/ index $\alpha$</td>
</tr>
<tr>
<td>$n$</td>
<td>Number of tasks</td>
</tr>
<tr>
<td>$m$</td>
<td>Number of cores</td>
</tr>
<tr>
<td>$spin_{P_k,q}$</td>
<td>The maximum time any task on processor $P_k$ needs to spin before the resource $R_q$ becomes available</td>
</tr>
<tr>
<td>$spin_i$</td>
<td>The total amount of time a task $\tau_i$ has to wait for all its global resource requests</td>
</tr>
</tbody>
</table>
3.1 The CP spin-lock priority

To be able to make a tool that derives the spin-lock priority we need to know the definition as is given by [3]. Afshar et al. provide the following definitions:

The priority of the task \( \tau_i \) is denoted by \( \rho_i \).

The set of tasks on a processor \( P_k \) is denoted as \( T_{P_k} \).

The task \( \tau_i \) that spins on a resource \( R_q \) gets a spin priority conform the RAP protocol that is used. The spin-lock priority for processor \( P_k \) is denoted as \( \rho_{P_k}^{\text{spin}} \).

The set of local and global resources that are accessed by jobs of a task \( \tau_i \) is denoted as \( \mathcal{R}S_i^L \) and \( \mathcal{R}S_i^G \).

**DEFINITION 3.** The highest local ceiling of any global resource on a processor \( P_k \) is denoted as \( rc_{P_k}^G \),

\[
rc_{P_k}^G = \max \{\rho_i | \tau_i \in T_{P_k} \land \mathcal{R}S_i^G \neq \emptyset\}. \tag{3.1}
\]

Under the CP spin-lock protocol \( \rho_{P_k}^{\text{spin}} = rc_{P_k}^G \).

The CP spin-lock priority for a core is the highest priority among tasks that use a global resource on the core. To be able to derive the CP spin-lock priority of a core \( P_k \) we can sort tasks that use a global resource in order of priority. The highest priority is the CP spin-lock priority.

3.2 The \( \hat{C}P \) spin-lock priority

**DEFINITION 4.** The highest local ceiling of any resource on a processor (either local or global) on a processor \( P_k \) is denoted as \( rc_{P_k}^{LG} \),

\[
rc_{P_k}^{LG} = \max \{\rho_i | \tau_i \in T_{P_k} \land \mathcal{R}S_i^G \cup \mathcal{R}S_i^L \neq \emptyset\}. \tag{3.2}
\]

Under the \( \hat{C}P \) spin-lock protocol \( \rho_{P_k}^{\text{spin}} = rc_{P_k}^{LG} \).

The \( \hat{C}P \) spin-lock priority for a core is the highest priority among tasks that use a resource on the core. To be able to derive the \( \hat{C}P \) spin-lock priority of a core \( P_k \) we can sort tasks that use a global or local resource in order of priority. The highest priority is the \( \hat{C}P \) spin-lock priority.
Chapter 4

Feasibility Study

Here we present a summary of the feasibility study. As a preparation of the measurement of resource access protocols we performed a feasibility study. It is beneficial for increasing the background knowledge and make a selection of environment.

The contributions of feasibility study where:
- Report about the feasibility study.
- Investigation on prerequisites needed to perform the assignment.
- Basic understanding of the different spin-lock protocols.
- Definition on the requirements.
- Understanding of the measurement of timing and memory overhead.
- Selection of an appropriate RTOS and an appropriate platform for implementation.
- Start on the exploration of the selected RTOS.
- Ported version of the chosen RTOS on the platform.
- Planing containing deliverables.

We want to measure the overhead in a practical setting like the automotive industry. This aspect is an important motivation, when considering possible RTOS to perform the project. The two properties the RTOS must comply are that it is opensource and OSEK compliant.

<table>
<thead>
<tr>
<th>Priority</th>
<th>Opensource</th>
<th>AUTOSAR</th>
<th>OSEK</th>
<th>MSRP</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LITMUS*</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>Much</td>
</tr>
<tr>
<td>ERIKA</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>Average</td>
</tr>
<tr>
<td>ArcCore</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>Average</td>
</tr>
</tbody>
</table>

Table 4.1: Properties of RTOS used for comparison

We have chosen to port the ERIKA operating system to the Nios soft-core on an Altera FPGA. The main reason for choosing ERIKA is: facility for at least 4 cores, the availability of MSRP and ERIKA is OSEK compliant. We perceived that the 4-core port would be easier to achieve. When queue’s are already implemented it saves a lot of time. Comparing the implementations against MSRP gives a possible benchmark. We earlier tried to port to a Raspberry Pi. There was no port available for the multi-core Raspberry Pi, and no proper document for porting it ourselves. Therefore we changed to the FPGA board.

We were able to run a hello world program on the ERIKA OS on 4 cores. The status is that we can program the board to have a hardware description of the 4 core board. The hardware description includes: the use of external memory, receiving interrupts from the buttons, writing a value to LEDs.

Performance of resource access protocols 19 Thursday 30th June, 2016 18:15
Chapter 5

The ERIKA OS

Here we present the kernel code of the multi-core ERIKA OS. We want to measure the RAP that is part of the kernel-layer. It is beneficial to understand how the current kernel-layer works. Before we can measure a new protocol we need to apply changes to implement the new RAP. In this chapter we give an overview of the structure of the OS and the kernel functions.

Chapter overview

System overview p.21 The system overview contains an introduction into the ERIKA OS. The three layers are explained of which the kernel layer is of our interest. The file structure is presented and the system is divided into 6 primary function groups that are further explained in the rest of this chapter. The function groups are: Initialization, Task handling, Ready queue, Stack and context switch, Resource access and Remote notification.

Initialization p.25 The initialization is the startup phase of the system. The programming of the cores happens sequentially, the initialization makes sure cores are synchronized. The interrupt routines are registered and data-structures are initialized.

Task handling p.26 The jobs of a task needs to get activated. First the task will be put into a ready queue. When the priority of the task at the head of the ready queue is larger than the priority of the executing task the task will become the executing task.

Ready queue p.27 The operations on the ready queue are used in task handling. The operations on the ready queue are: putting a task into the ready queue, removing a task from the head of the ready queue and checking the task at the head of the ready queue. The queue itself is ordered first by priority of the tasks, tasks of the same priority are FIFO ordered.

Stack and context switch p.28 The actual execution of a job is initiated by stack and context switch functions. The 3 main functions are a context switch: to execute a task that contains no stack (starting a job), to execute a task that already has a stack (resuming a job), to terminate a job.

Resource access p.29 To get access to a resource the function GetResource() is provided, the release of the resource is done by ReleaseResource(). Both Get- and ReleaseResource() make sure the priority of the system ceiling is adjusted. The system ceiling is a dynamic variable i.e., it represents both the priority of the executing task as well as the resources to which the task has access. If the resource is a global resource the functions spin_in() and spin_out() are used to get a spin lock on the resource and to signal its release. A task polls (spins) non-preemptively on the resource until it becomes available.

Remote notification p.42 Communication between the cores can by done by shared memory and by interrupts. To send a remote notification (RN) the sending core builds an RN message in shared memory (which is protected by a mutex). The message contains the type of interrupt for instance remotely activate a task. The sending core then sends a signal to the I/O of the receiving core. The receiving core starts executing an interrupt routine and checks the RN message.
5.1 System overview

The multi-core ERIKA OS is a kernel-layer built on top of a Hardware Abstraction-Layer (HAL-layer) and provides functionality to a user or Application Programming Interface-layer (API-layer). Figure 5.1 illustrates what software is used in the system. The hardware description is omitted in the picture. In the HAL-layer all the drivers are defined and libraries are available. An example is a library function IOWR_ALTERA_AVALONPIO_DATA() that can put a 0 or 1 on an output pin. But also the function printf() is provided in the HAL. As a programmer it is possible to write a program that directly runs on the CPU without an OS. This could be done by writing a C program while using functions defined in the HAL-layer. It is even possible to define our own drivers and add them to the HAL-layer. However, having an OS can provide a lot of benefits. The operating system is a software that manages computer hardware and software resources and provides common services for computer programs.

![Figure 5.1: RT-Druid generated code, ERIKA OS and the Altera HAL-layer](image)

The ERIKA OS shown in Figure 5.1 provides some primitives (functions) to the API-layer. For instance the scheduling of tasks and the related resource access. To be able to schedule the tasks that become available, the OS needs some information about the tasks, such as the mapping of tasks to cores, the priorities of the tasks, which resources each task uses and whether the resources are local or global. The OSEK standard states that all of this information should be provided in a CONFIG.OIL file. The CONFIG.OIL file looks like a textual definition of the objects: cpu, task and resource. A part of the CONFIG.OIL can be seen in Listing 5.1.

```c
1 CPU_DATA = NIOSII {
2 ID = "cpu0";
3 MULTI_STACK = TRUE;
4 APP_SRC = "cpu0_main.c";
5 APP_SRC = "shareddata.c";
6 STACK_TOP = "__alt_stack_pointer";
7 SYS_SIZE = 0x1000;
8 SYSTEM_LIBRARY_NAME = "Project_syslib";
9 SYSTEM_LIBRARY_PATH = "C:/Users/Desktop/ERIKA/software/Project_syslib";
10 IPIC_LOCAL_NAME = "IPIC_KMPUT_0";
11};
```

Listing 5.1: Part of an OIL file with a textual definition of the instance cpu0 of the object CPU_DATA
5.1.1 Initialization via RT-Druid

Here we present the RT-Druid tool of ERIKA. The tool generates some of the data-structures used in the kernel-layer.

The RT-Druid [38] Code Generator is a plugin for the development framework Eclipse that is used to automatically generate configuration code at compile time. The RT-Druid Code Generator is an open and extensible environment, which is based on XML and open standards. RT-Druid allows the generation of the configuration code for the Erika Enterprise real-time kernel. The code configuration may start from an OSEK OIL or an AUTOSAR XML definition to create the configuration code for applications running on a variety of environments. RT-Druid has been entirely written in Java. It is based on the development framework Eclipse.

Starting from an OIL configuration file, the tool creates a directory that will contain all the generated files. The first file that is created is the make-file. Then, typically the files Eecfg.h and Eecfg.c are created, containing the data-structures needed to run an Erika Enterprise application. After the files are created, RT-Druid automatically runs the "make" command in the Debug directory.

5.1.2 The file structure

Here we present the file structure of the ERIKA-kernel layer. All of the files contain roughly one function. It is beneficial to specify the file structure to get more insight in the kernel layer.

The software overview has been shown in Figure 5.1. The RT-Druid generates 3 files from the CONFIG.OIL file. Those 3 files do not contain any functions, they only contain data-structures. This way the ERIKA OS files in the kernel-layer have access to information about the tasks and their priorities. The ERIKA OS includes the HAL-libraries and the RT-Druid generated data-structures. The ERIKA OS itself consists of +/- 20 files (The version we use is the OO-kernel with a lot of unused files removed such as alarm files). Figure 5.2 illustrates what those +/- 20 files are. The file ee_internal.h is included in all .c files. The file ee_internal.h on its turn includes (indirectly) all other h files. All the functions are pre-defined in ee_internal.h. Figure 5.1 illustrates how the user can use the functions from the kernel-layer in its programs. The user programs are represented by the API-layer.

Listing 5.2 shows a simple example program (part of API-layer).

```c
int main(){
    StartOS(OSDEFAULTAPPMODE);
    ActivateTask(task0);
    while(1){;;}
    return 0;
}
```

Listing 5.2: A simple main program

The user makes a main file in the API-layer as shown in Figure 5.1. In the API-layer it is possible to use functions provided by the HAL and Kernel layer. For instance the services to schedule a task and access resources are provided. Listing 5.2 shows the activation of a task. The use of the task needs to be declared in the CONFIG.OIL file first. The RT-Druid generates the data-structures that the Kernel-layer requires. Once the user has provided the code of the API-layer and the CONFIG.OIL file then it is possible for ECLIPSE to build the binary file. To build the project, ECLIPSE uses the HAL, KERNEL and API layer code and generates a binary .elf file. The ELF file is programmed onto the FPGA-board.
Figure 5.2: The file include structure of the OO-kernel

Figure 5.2 illustrates how the files are included in the Kernel-layer. Figure 5.3 illustrates what the mapping between files and functions is. Most c-files contain just one function, with the exception of `ee_rn.c` which contains `ee_rn_execute` and `ee_rn_handler` functions. The `ee_rn_execute` function is only called from the `ee_rn_handler` function, not from any other place. Most files file effectively provide one function.

<table>
<thead>
<tr>
<th>Category</th>
<th>Function</th>
<th>Description</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>StartOS()</td>
<td>Start the operating system</td>
<td>5.2</td>
</tr>
<tr>
<td>Task handling</td>
<td>ActivateTask()</td>
<td>Activates a task on the current or another core</td>
<td>5.3</td>
</tr>
<tr>
<td>Ready queue</td>
<td>rq_queryfirst()</td>
<td>Get the task at the head of the queue</td>
<td>5.3</td>
</tr>
<tr>
<td></td>
<td>rq2stk_exchange()</td>
<td>Remove a task from the head of the ready queue</td>
<td>5.3</td>
</tr>
<tr>
<td></td>
<td>rq_insert()</td>
<td>Insert a task in the ready queue</td>
<td>5.3</td>
</tr>
<tr>
<td>Stack &amp; context</td>
<td>ready2stacked()</td>
<td>Preempts the active task and starts a new task</td>
<td>5.5</td>
</tr>
<tr>
<td>switch</td>
<td>stkchange()</td>
<td>Preempts and resumes tasks with a stack</td>
<td>5.5</td>
</tr>
<tr>
<td></td>
<td>terminate_task()</td>
<td>Terminates the active task, resumes another task</td>
<td>5.5</td>
</tr>
<tr>
<td>Resource access</td>
<td>GetResource()</td>
<td>Lock the shared resource</td>
<td>5.6.1</td>
</tr>
<tr>
<td></td>
<td>ReleaseResource()</td>
<td>Release the shared resource</td>
<td>5.6.2</td>
</tr>
<tr>
<td></td>
<td>spin_in()</td>
<td>Busy wait until release and then lock the resource</td>
<td>5.6.4</td>
</tr>
<tr>
<td></td>
<td>spin_out()</td>
<td>Signal the release of the resource to the other cores</td>
<td>5.6.4</td>
</tr>
<tr>
<td>Remote notification</td>
<td>rn_handler()</td>
<td>Handling the interrupt routine</td>
<td>5.7</td>
</tr>
<tr>
<td></td>
<td>rn_execute()</td>
<td>Executes the content of the RN</td>
<td>5.7</td>
</tr>
<tr>
<td></td>
<td>rn_send()</td>
<td>Send an RN (interrupt) to another core</td>
<td>5.7</td>
</tr>
</tbody>
</table>

Table 5.1: The function description and category of the functions we use in the OO-kernel
5.1.3 Overview of files and functions

Figure 5.3 illustrates the c-files and their corresponding functions. The system always needs to call `ee_startos` if the user wants to use the OS. After the OS is started it is possible to use the kernel functions for instance to activate tasks. Table 5.1 briefly describes the functions.

In the legend is stated that some arrows indicate that the function call is new. Those green arrows show function calls that do not occur in the original ERIKA kernel layer. Later chapters describing the design of the new RAP will explain why we added those function calls. The functions of the kernel-layer can roughly be grouped in 6 categories: 5.2 Initialization, 5.3 Task handling, 5.4 Ready queue, 5.5 stack and context switch, 5.6 Resource access and 5.7 Remote notification.

Figure 5.3: How functions of files are called between files and provided to the API-layer

In the legend is stated that some arrows indicate that the function call is new. Those green arrows show function calls that do not occur in the original ERIKA kernel layer. Later chapters describing the design of the new RAP will explain why we added those function calls. The functions of the kernel-layer can roughly be grouped in 6 categories: 5.2 Initialization, 5.3 Task handling, 5.4 Ready queue, 5.5 stack and context switch, 5.6 Resource access and 5.7 Remote notification.

Figure 5.4: A symbol in the top corner of the page will keep track of the function group described.
5.2 Initialisation

Figure 5.5: The call-graph of StartOS()

Figure 5.5 illustrates the functions that are used to start the OS. The graph contains the following information; the function on the left is the function of which we show a call-graph. A box where the arrow originates from is the caller of a function. The function StartOS() starts the system, it registers the IPIC (InterProcessor Interrupt Controller) [39] and synchronizes the CPUs. One of the 4 cores is the initiator core and starts the other 3. Note that the functions EE\text{foo}() and EE_{hal}\text{foo}() are also defined as foo().

The StartOS() registers the inter-core interrupt handler. When the core starts executing the interrupt handler uses a lookup table to figure out how to handle the interrupt. The function alt_{irq}\text{register}() as shown in Figure 5.5 registers a function into the interrupt table. The function that is put into the interrupt table is nios2_{IRQ}\text{handler}(). Thus StartOS() does not execute nios2_{IRQ}\text{handler}(), but registers it as the code that should be executed upon the arrival of its interrupt. The interrupt table consists of several interrupts. During the creation of the hardware we add I/O pins that generate an interrupt upon the falling edge. As soon as the interrupt triggers, the interrupt handler checks the lookup table and executes nios2_{IRQ}\text{handler}().

Note that the ERIKA OS\footnote{We choose to use private stacks, all the functions explained in the chapter the ERIKA OS can operate in both shared and private stack operation with the exception of stkchange() p.28} provides a lot of functionality that we do not use. To reduce the complexity we explain only parts of the kernel we actually use. It is for instance possible to activate tasks or put them into the ready queue before the system is started. We do not use this functionality and it is therefore omitted. Furthermore we only use the OO-kernel.
5.3 Task handling

Figure 5.6 illustrates the steps the ActivateTask() function goes through to activate a task on the current or another core. In real time terms a task is a recurring execution of work, and a job is an instance of such an execution. Thus the ActivateTask() function should actually have been called ActivateJob() since it activates an instance of the task. At the start of the ActivateTask() function a check is performed to determine whether the task is a local or remote task. If the task occurs on another core than the activator core a remote activation needs to take place. An RN (remote notification) is sent to the core where the task should be activated. On the receiving core the ActivateTask() function starts execution. During the execution of ActivateTask() function the interrupts are disabled to prevent multiple activations disrupting each other. The right top corner of Figure 5.6 shows the implementation of the private stack.

5.3.1 RQ_insert()

The ready queue is a linked list of tasks and their priorities. The RQ_insert() function has a taskID as parameter. Let \( \tau_{\text{new}} \) denote the arriving task. The ready queue is priority-ordered. The task at the head of the queue has the highest priority. RQ_insert() adds \( \tau_{\text{new}} \) to the ready queue according to its priority compared to other tasks in the queue. In case of equal priorities task \( \tau_{\text{new}} \) is placed later into the queue i.e., FIFO-ordering is used. Whenever a new task (\( \tau_{\text{new}} \)) is added to the ready queue, a preemption check is performed. If the task at the head of the queue has a priority higher than that of the active running task on the core, then the running task is preempted and switched with the task at the head of the ready queue. The function ActiveTask() is terminated if \( \tau_{\text{new}} \) is not at the head of the ready queue.

5.3.2 stk_queryfirst()

The function returns the task at the head of the stack, which is also the executing task. We use separate stacks each task has a private stack, but those stacks are still ordered. The order of the stacks is LIFO and the head of the stack is the task that is last activated but not yet terminated.

5.3.3 rq2stk_exchange()

The function determines the task at the head of the ready queue. Removes the task from the head of the ready queue. The ID of the task that was at the head is returned and used as argument for the context switch that takes place in the next function (Ready2Stacked).

5.3.4 ready2stacked()

The function ready2stacked() is implemented in ASM (assembly) and stores the state of the executing task such that it can be resumed on a later time. The task that was at the head of the queue is activated. The starting task has no stack yet thus no registers need to be loaded. The function does not return, instead the core starts executing the task.
5.4 Ready queue

Figure 5.7 illustrates the implementation of the ready queue. In the top right corner it is illustrated which tasks the ready queue contains for this particular example. The order of the queue in this example is 8,1,4,3,2,5. The ordering of the ready queue is first in order of priority, within each priority the order is FIFO. The following data-structures are used to implement the ready queue:

- EE_rq_bitmask of size 1 byte
- EE_rq_free of size 1 byte
- EE_rq_pairs_tid of size # of tasks (int array)
- EE_rq_queues_head of size 8 (max # of priorities) (int array)
- EE_rq_queues_tail of size 8 (max # of priorities) (int array)
- EE_rq_pairs_next of size # of tasks (int array)
- EE_rq_pairs_tid of size # of tasks (int array)
- EE_rq_queues_head of size 8 (max # of priorities) (int array)
- EE_rq_queues_tail of size 8 (max # of priorities) (int array)

The bitmask contains a binary representation of the priorities that occur in the resource queue. The $i^{th}$ bit means there is at least one item of priority $\rho = i$ occurs in the ready queue. EE_rq_queues_head[$\rho$] points to the head and EE_rq_queues_tail[$\rho$] points to the tail of the queue priority $\rho$. All the queue's of all the priorities are stored in a linked list EE_rq_pairs_tid and EE_rq_pairs_next. The queue with priority $\rho$ consists of tasks at priority $\rho$. The task next in the queue of priority $\rho$ can be found by EE_rq_pairs_next which is EE_rq_pairs[EE_rq_pairs_next[EE_rq_queues_head[$\rho$]]]. Each next task is found by EE_rq_pairs_next until it is empty (-1).

Let us assume a new task $\tau_{new}$ with thread ID 27 and priority 6 wants to enter the ready queue. Figure 5.7 depicts the situation before adding $\tau_{new}$. EE_rq_free contains an index of the ready queue that is empty, in the example of Figure 5.7 the empty index is 3. The thread ID is placed in the ready queue, thus EE_rq_pairs_tid[3] becomes 27. The free descriptor needs to indicate the next free index, thus EE_rq_free uses EE_rq_pairs_next[EE_rq_free] and becomes 7. There is no next item yet thus EE_rq_pairs_next[3] becomes -1. Since there already exists a queue of priority 6 the item needs to be added after the thread 8. Thus EE_rq_pairs_next[8] becomes 3. Now the new item is added to the queue. After the insert function terminates a preemption check will be performed. Since the executing task has the same priority no context switch will take place.

 void rq_insert(int t) {
    EE_rq_free = EE_rq_pairs_next[EE_rq_free];
    EE_rq_pairs_tid[t] = t;
    EE_rq_pairs_next[t] = -1;
    if (EE_rq_queues_tail[p] == -1) {
        EE_rq_bitmask |= EE_rq_pairs[tid[t]];
        EE_rq_queues_head[p] = temp;
    } else {
        EE_rq_pairs_next[EE_rq_queues_tail[p]] = temp;
    }
}

Listing 5.4: c-code rq_insert()
5.5 Stack and context switch

To preempt the running task and activate another task we need to perform a context switch. A context switch involves storing and restoring the state of a task or thread so that execution can be resumed at a later moment in time. The context switch involves switching registers, stack pointer and program counter. Note that the NIOS is a 32-bit embedded-processor and has 32 registers in its core. The state of some registers needs to be saved on the stack in case a task gets preempted.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>always 0</td>
<td>no</td>
</tr>
<tr>
<td>r1</td>
<td>scratch register</td>
<td>no</td>
</tr>
<tr>
<td>r2-3</td>
<td>return values</td>
<td>must be preserved in the register not on the stack</td>
</tr>
<tr>
<td>r4-7</td>
<td>arguments</td>
<td>must be preserved in the register not on the stack</td>
</tr>
<tr>
<td>r8-15</td>
<td>caller-saved registers</td>
<td>no</td>
</tr>
<tr>
<td>r16-23</td>
<td>callee-saved register</td>
<td>saved on the stack</td>
</tr>
<tr>
<td>r24-25</td>
<td>exception and breakpoint temp</td>
<td>no</td>
</tr>
<tr>
<td>r26</td>
<td>global pointer</td>
<td>no</td>
</tr>
<tr>
<td>r27</td>
<td>stack pointer</td>
<td>saved inside HAL data structures</td>
</tr>
<tr>
<td>r28</td>
<td>frame pointer</td>
<td>saved on the stack</td>
</tr>
<tr>
<td>r29, r30</td>
<td>return addr for bpoints/exceptions</td>
<td>no</td>
</tr>
<tr>
<td>r31</td>
<td>return address</td>
<td>used to change context not stack</td>
</tr>
</tbody>
</table>

Table 5.2: The registers of the NIOS core are saved on the stack

---

5.5.1 Ready2Stacked(τ_i)

The function Ready2Stacked(τ_i) has one argument, the task it needs to activate. The function will stop the executing task, save its state on the stack and start executing the task that is given as a parameter. Let’s assume that task τ_2 is executing and task τ_1 is already preempted and on the stack. It is possible to activate task τ_1 again with Ready2Stacked(τ_1) what would happen is that the function would put task τ_2 on the stack and start another instance of task τ_1. Thus the function does not check whether the task it is activating already has a stack.

5.5.2 stkchange(τ_i)

The function stkchange(τ_i) has one argument, the task it needs to resume. The function will stop the executing task, save its state on the stack and load the state of the task that was given as a parameter. Let’s assume that task τ_2 is executing and task τ_1 is already preempted and on the stack. It is possible to resume task τ_1 with stkchange(τ_1) what would happen is that the function would put task τ_2 on the stack and load the state of task τ_1 and resume it. (The function requires each task to have a private stack).

5.5.3 terminat_task(τ_i)

The function terminat_task(τ_i) has one argument, the task it needs to terminate. The function will stop the executing task, save its state on the stack and load the state of the task that was given as a parameter. Let’s assume that task τ_2 is executing and task τ_1 is already preempted and on the stack. It is possible to resume task τ_1 with terminat_task(τ_1) what would happen is that the function would put task τ_2 on the stack and load the state of task τ_1 and resume it. (The function requires each task to have a private stack).
5.6 Resource access

GetResource() p. 30 The GetResource() function is used before accessing a resource. The priority of the system ceiling gets increased to the priority of the resource ceiling. (In case of nesting of local resources, it might be the case that the system ceiling doesn’t change.) In case the resource is local the function returns and the task can access the resource. In case the resource is global the spin_in() function is called.

ReleaseResource() p. 32 The ReleaseResource() function is used after accessing the resource. The priority of the system ceiling gets reduced to the priority of the task before accessing the resource. In case the resource is local the function returns and the task continues its operation. In case the resource is global the spin_out() function is called.

spin_in() p. 35, 36 The spin_in() function is only called from GetResource() and only in case the resource is global. The function implements a spin lock. The task waits (non preemptive in HP explained in Section 1.1 ) on the resource to become available, by putting itself into a resource queue. Once the task (and the core executing the task) is at the head of the resource queue and the resource becomes unlocked the task acquires the lock of the resource.

spin_out() p. 35, 36 The spin_out() function is only called from ReleaseResource() and only in case of a global resource. The function implements the unlocking of the spin lock. spin_out() resets the variable indicating the lock status of the resource to unlocked.

The resource access data-structures

The system ceiling EESYSستری is used to keep track of the priority the system is currently executing on. Any time a task starts executing the system ceiling is adjusted to the priority of the executing task. Acquiring a resource increases the system ceiling to the resource ceiling. Releasing a resource returns the system ceiling to the priority before accessing the resource. The benefit of keeping track of the system ceiling is that any arriving task that ends up at the head of the ready queue can be compared to the system ceiling to determine whether a context switch should take place.

The resource queue consists of EE哈尔spin_value[ ] and EE哈尔spin_status[ ]. The resource queue is used to order the access to the resource the tasks request access to. All tasks accessing a resource and all spinning tasks occur in the resource queue.

Figure 5.9: The resource access of a local and global resource

The functions Get- and ReleaseResource() are used to change the system ceiling. The functions spin_in() and spin_out() are used to put the task into the queue and (un) lock the resource. First the local resource is accessed, only the system ceiling changes to the resource ceiling (non-preemptive priority in HP). Second the global resource is accessed, both the system ceiling changes and the lock value is updated.

Figure 5.9 illustrates a task accessing a local and a global resource.

All information illustrated in the Figure 5.9 is related to core P₀
CHAPTER 5. THE ERIKA OS

5.6.1 Resource access: GetResource()

Here we explain how the ERIKA implementation of GetResource() works. The GetResource() function is used to acquire a resource and is part of the RAP. To implement a new RAP we need to change this function amongst others.

Figure 5.10 illustrates the call-graph of the function GetResource(). The purpose of GetResource() is to control the access to shared resources. GetResource() keeps track whether or not a resource is shared between cores (the ID of the resource has a bit indicating that it is a global resource). The access to a resource is also protected by mutual exclusion. In case the resource is already in use, the task starts spinning on a global spin lock. The current implementation does not yet allow preemption of a spinning task by higher priority tasks. The use of GetResource() provides the controlled resource access to the API-layer.

The execution of the function

Steps taken in GetResource():
1. All interrupts get disabled.
2. The original system ceiling is stored.
3. The current system ceiling is possibly increased to the resource ceiling.
4. In case the resource is global the task starts spinning.
5. When the resource becomes available to the task, the resource is locked and access is granted.
6. All interrupts are enabled again.

Data-structures and initialization

Figure 5.11 illustrates how the data is structured. GetResource() has a parameter ResID, which specifies the resource ID and whether the resource is global. There exists a define (EE_GLOBAL_MUTEX equal to 0x8000 0000) that is used to determine whether a resource is global. The resource identifier ResID is split in a global bit (MSB) and a local part (all other bits). The local resource ID is put back in ResID. (The local resource ID is used for indexing). EE_sys_ceiling contains the priority that the core is currently executing its task on. In the example of Figure 5.11 the EE_sys_ceiling is 0x02 before requesting the resource and 0x12 after requesting access to the resource. The system ceiling is used by the system to track the priority of the executing task. Whenever a change occurs in the system i.e., either another task arrives or a resource is released, the system ceiling is used to determine whether a context switch needs to take place. The original system ceiling (required when the task has finished accessing the resource) is stored in EE_resouce_oldceiling. The system ceiling is increased by setting the bit that indicates the resource ceiling (if the bit was already set the system ceiling is not changed).

The system_ceiling is initialized with 0x0. EE_resouce_oldceiling[R] is initialized with 0x0. EE_resouce_ceiling is initialized with the a priority based on MSRP by the RT-Druid tool. Listing 5.5 presents the pseudo code and Listing 5.6 presents the c-code of GetResource().
 CHAPTER 5. THE ERIKA OS

// and /* denote the commands that are original from the ERIKA kernel.
/* Are comments made by interpretation and are not original.

GetResource(Rq){
    Mask off the MSB, that indicates whether Rq is a global or a local resource
    Check whether the identifier of the resource does not exceed the number of resources
    Check whether the priority of the active task does not exceed the system ceiling
    Disable interrupts
    Retrieve the active task identifier
    Update an array that keeps track of which task uses which resource
    Update an array that keeps track which resource is in use
    Increase the system ceiling to the spin priority
    In case Rq is a global resource, lock the other cores (spin_in())
    Enable interrupts
}

Listing 5.5: The pseudo code of function GetResource()

StatusType GetResource(ResourceType ResID){
    register EE_FREG flag; register TaskType current; register EE_UREG isGlobal;
    isGlobal = ResID & EE_GLOBAL_MUTEX; /*determine whether the resource is global
    ResID = ResID & ~EE_GLOBAL_MUTEX; /*determine the local resource identifier
    if (ResID >= EE_MAX_RESOURCE){return E_OS_ID;} /*check whether the resource ID is out of bounds
    if (EE_resource_locked[ResID] || EE_th_ready_prio[EE_stk_queryfirst()] > EE_resource_ceiling[ResID]) {
        return E_OS_ACCESS; /*check the system ceiling
    }
    flag = EE_hal_begin_nested_primitive(); /*the interrupts get disabled
    current = EE_stk_queryfirst(); /*retrieve the active task identifier
    EE_resource_stack[ResID] = EE_th_resource_last[current]; //insert the resource into the data structure
    EE_th_resource_locked[last] = ResID; /*update the task that last used the resource
    EE_resource_locked[ResID] = 1; /*data-structure that keeps track of the resource is updated
    EE_resource_ceiling[ResID] = EE_sys_ceiling; /*save the system ceiling
    EE_th_resource_stack[ResID] = EE_th_resource_last[current]; //possibly increase the system ceiling
    EE_hal_end_nested_primitive(flag); /*the interrupts get enabled again
    if (isGlobal) spin_in(ResID); /*if this is a global resource, lock the others CPUs
    return E_OK;
}

Listing 5.6: The c-code of function GetResource()
5.6.2 Resource access: ReleaseResource()

Here we explain how the ERIKA implementation of ReleaseResource() works. The ReleaseResource() function is used to release a resource and is part of the RAP. To implement a new RAP we need to change this function amongst others.

The function ReleaseResource() in Figure 5.12 implements the release of a resource. ReleaseResource is used to release both local and global resources, the later also performs a spin_out() function to signal the release to other cores. The function spin_in() later explained in section 5.6.4 toggles the field that indicates if a task has locked a resource, such that other cores know it is available. ReleaseResource() updates the system ceilings and checks whether the executing task still has the highest priority. When the current task releases the resource it might be the case that the task at the head of the ready queue has a higher priority than the system ceiling (and current executing task). In this case the active executing task needs to be preempted. Instead of calling the scheduler this is done inside the ReleaseResource() function to make it atomically with the release of the resource.

The execution of the function

Steps taken in ReleaseResource():
1. Disable the interrupts.
2. Reset a bit that keeps track of whether the resource is in use or unlocked.
3. Restore the system ceiling to the level before spinning (stored in GetResource()).
4. Get the task-ID from the head of the ready queue (\(\tau_i\)).
5. If there exists a task in the ready queue (\(\tau_i \neq -1\)) and its priority is higher than the system ceiling do
   (a) till (f):
      A higher Priority task is activated and should be executing.
      (a) Change the status of the executing task to ready.
      (b) Change the status of the task at the head of the ready queue to running.
      (c) Increase the system ceiling (with bit denoting \(\rho_i\)).
      (d) Remove the task from the head of the ready queue
      (e) Perform a context switch, preempt the executing task and start executing the task that was previously at the head of the ready queue.
6. Enable the interrupts.

Note that a preempted is not inserted into the ready queue. The array EE_4th_next[] as was illustrated in Figure 5.6 keeps track of the preemption order and allows to resume preempted tasks. The ready queue only contains tasks that arrived and have not yet partly executed.
Data-structures and initialization

Figure 5.13 illustrates how the data is structured. ReleaseResource() has a parameter ResID (i of \(R_i\)), which contains whether the resource is global (MSB) and the local resource ID. The MSB is put in isGlobal and later used, in case the resource is global the spin_out() function will release the spin lock. EE_sys_ceiling contains the priority of the executing task, thus before release the EE_sys_ceiling contains the maximum priority (for global resources only, and only in the current implementation). After release EE_sys_ceiling contains the priority of task (its dispatch priority). That priority EE_sys_ceiling needs to be updated with the value the executing task had prior to the locking, which is stored in EE_resource_oldceiling.

After the system ceiling is updated, the kernel needs to figure out whether it needs to perform a context switch. It might be the case that a higher priority task arrived during the non-preemptive resource access. The arriving task had a priority between the executing tasks priority and the maximum (non-preemptive) priority. During the activation of the arriving task it had a priority lower than the system ceiling and thus it was put into the ready queue. Thus the releasing task executing the ReleaseResource() function needs to check the priority of the task at the head of the resource queue. If the priority of the task at the head of the resource queue is higher than the system ceiling (the priority the core is currently executing its tasks on) a context switch takes place.

The data-structures EE_th_resource_last[], EE_resource_locked[] and EE_resource_stack[] are used to keep track of the resource status. The value contained in the data-structures does not influence the behaviour of the system using the OO-kernel. The datastructures are used for error checking and might have an influence on the operation in other kernels. EE_th_resource_last[] contains the last resource the task accessed. EE_resource_locked[] contains 1 if the resource is locked and 0 if unlocked, this is not used for polling. EE_resource_stack[] contains a stack of resources that a task uses in case of nested resource access.

The data-structure EE_th_status[] is used to keep track of the status of the task. If a context switch is performed the status of the releasing task is changed to READY. The activated task gets the status RUNNING. The status of a task is not used to influence the behaviour. The status is only updated to keep track of the status and generate error messages.

<table>
<thead>
<tr>
<th>Data-structure</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EE_Global_mutex (static)</td>
<td>0x8000 0000</td>
</tr>
<tr>
<td>EE_sys_ceiling</td>
<td>0x0</td>
</tr>
<tr>
<td>EE_th_dispatch_prio[(r_i)]</td>
<td>f((r_i))</td>
</tr>
<tr>
<td>EE_resource_oldceiling[(R_i)]</td>
<td>0x0</td>
</tr>
<tr>
<td>EE_resource_ceiling[(R_i)]</td>
<td>0x0</td>
</tr>
<tr>
<td>EE_th_resource_last[(r_i)]</td>
<td>-1</td>
</tr>
<tr>
<td>EE_resource_locked[(R_i)]</td>
<td>0</td>
</tr>
<tr>
<td>EE_resource_stack[(R_i)]</td>
<td>-1</td>
</tr>
<tr>
<td>EE_th_status[(r_i)]</td>
<td>3 (SUSPENDED)</td>
</tr>
</tbody>
</table>

Table 5.3: Initial values of the data-structures
CHAPTER 5. THE ERIKA OS

Listing 5.7: The pseudo code of the function ReleaseResource()

```plaintext
ReleaseResource(Rq){
    Mask off the MSB, that indicates whether Rq is a global or a local resource
    Check whether the identifier of the resource Rq does not exceed the number of resources
    Check whether the priority of the active task does not exceed the system ceiling
    Disable interrupts
    Update an array that keeps track of which task uses which resource
    Update an array that keeps track which resource is in use
    In case this is a global resource, unlock the others cores (spin_out())
    Retrieve the task at the head of the ready queue
    Return the system ceiling to the priority the system had before locking the resource
    If the ready queue is not empty{
        If the priority of the task at the head of the queue is higher than the system ceiling{
            //we have to schedule a ready thread
            The status of the active thread becomes READY (indicating it has a stack)
            The status of the task at the head of ready queue becomes RUNNING
            The context switch takes place
            - Preempting the active task and putting it onto the stack
            - Removing the task from the head of the ready queue
            - Pointing the program counter to the task that was at the head or ready queue
            - The execution of the task starts
        }
    }
    Enable interrupts
}
```

Listing 5.8: The c-code of function ReleaseResource()

```c
StatusType ReleaseResource(ResourceType ResID){
    EE_TID rq, current; EE_UREG isGlobal; register EE_FREG flag;
    isGlobal = ResID & EE_GLOBAL_MUTEX; /\determine whether the resource is global
    ResID = ResID & ˜EE_GLOBAL_MUTEX; /\determine the local resource identifier
    if (ResID >= EE_MAX_RESOURCE){return E_OS_ID;} /\check whether the resource ID is out of bounds
    current = EE_stkfirst; /\retrieve the task identifier
    if (EE_th_ready_prio[current] > EE_resource_ceiling[ResID]) {return E_OS_ACCESS;} /\check the system ceiling
    flag = EE_hal_begin_nested_primitive(); /\the interrupts get disabled
    // remove the last entry from the data structure
    EE_th_resource_last[current] = EE_resource_stack[EE_th_resource_last[current]];;
    EE_resource_locked[ResID] = 0; /\data-structure that keeps track of the resource is updated
    if (isGlobal) EE_hal_spin_out(ResID); // if this is a global resource, unlock the others cores
    rq = EE_rq_queryfirst(); //retrieve the task identifier
    EE_resource_locked[ResID] = 0; //data-structure that keeps track of the resource is updated
    EE_resource_locked[ResID] = 0; //data-structure that keeps track of the resource is updated
    if (rq != EE_NIL) { // check if there is a preemption
        if (EE_sys_ceiling < EE_th_ready_prio[rq]){ // we have to schedule a ready thread
            EE_th_status[current] = READY; // the running task is now suspended
            EE_th_status[rq] = RUNNING; // and another task is put into running state
            EE_sys_ceiling|=EE_th_dispatch_prio[rq]; //the system ceiling gets increased again
            EE_hal_ready2stacked(EE_rq2stk_exchange()); //retrieved & remove the task id at the head of the ready queue
            EE_hal_end_nested_primitive(flag); //the interrupts get enabled again
        }
    }
    return E_OK;
}
```
5.6.3 Resource access: Introduction to MSRP

In a spin-based approach, a task essentially busy waits until a condition becomes true (i.e., holds). In MSRP, a task requiring access to a global resource busy waits non-preemptively until (i) it is the first in line (first-in-first-out) waiting for the resource and (ii) the resource is free. Conceptually, a first-in-first-out (FIFO) queue of tasks can therefore be associated with every global resource. Because spinning in MSRP is non-preemptive, at most one task per core can spin on a global resource. It is therefore also possible to associate a FIFO-queue of cores with every global resource.

Existing implementation of MSRP in ERIKA

To implement MSRP, ERIKA essentially maintains a distributed polling-bit queue for each global resource i.e., a (non-empty) FIFO queue of polling bits used by cores that want to access that global resource. The polling bits are stored as global data and the addresses of queue elements are stored in local data. To enable access to the queue, the tail of the queue is also stored as global data, containing the address of the global polling bit that needs to be inspected by the next core that requires access to the global resource. Access to the tail of the queue requires mutual exclusion.

1. Global data:
   (a) A global polling bit\(^4\) per core per global resource. This bit is written (toggled) by the core when it releases the resource and inspected (read) by the next core that requires access to the resource.
   (b) A global "spin-status" per global resource, that contains
      - the bit value that represents "what-means-locked" for the global polling bit;
      - the tail of the (non-empty) FIFO-queue i.e., the address of the global polling bit that shall be inspected by the next core that requests access to the global resource to determine whether or not the resource is locked.

2. Local data (i.e., the spin-status of the global resource at the time of the request, if any):
   (a) A local polling address per core, that contains the address of the global polling bit that is inspected by (a task on) that core, if any, during busy-waiting\(^5\).
   (b) A local "what-means-locked" bit per core, that provides the interpretation of the value of the global polling bit currently inspected, if any.

Initially, all global resources are unlocked i.e., not in use. All global polling bits are set to zero and the bit value in the global spin-status that represents the availability of a resource are all set to one i.e., initially one means locked, so the value zero of the global polling bit means unlocked. The address of the global polling bit in the global spin-status is set to an arbitrary core i.e., by default core 0. The local data is only relevant when a task on that core is busy waiting for a global resource.

The behavior of spin_in() and spin_out() is now as follows:

- **spin_in(R):**
  1. Mutual exclusive access:
     (a) The global spin-status of R is retrieved and stored in local data.
     (b) The global spin-status of R is updated with the address of the global polling bit of this core for R and the bit value of the "what-means-locked" is set to the current value of the global polling bit.
  2. A busy wait is performed till the value of the global polling bit of which its address was recorded in the core’s local data as "local polling address" becomes unlocked based on what has been recorded in the local bit for what means unlocked.

- **spin_out(R):**
  1. Toggle the global polling bit of R of the core itself (!)

Concluding remarks

A core (or task) that is accessing a global resource is unaware of the fact that another core may (or may not) have requested the same resource i.e., it is unaware of successors in the polling-bit queue. A core that is waiting for a resource to become available is aware of the core in front of it in the polling-bit queue. Notification upon resource release is therefore without any knowledge who needs to be notified. For an interrupt-based version of MSRP, the notifying core shall know which core to inform. The "knowledge" therefore has to be reversed (from requesting to releasing core).

\(^4\)In implementation denoted as spin_value

\(^5\)The global polling address is the address of the polling bit related to "the core that has issued the most recent request (to the same resource, if any) before this core".
5.6.4 Resource access: spin\_in() & spin\_out()

Here we explain the function spin\_in() and spin\_out() which are part of the GetResource function. These functions (spin\_in() and spin\_out()) contain the resource queue management and the spinning on a global variable. To implement a new RAP we will need to change the functions “spin\_in() and spin\_out()”.

Data-structures and initialization

The mutex hardware lock is a basic test-and-set lock, it is a bit having two values: locked and unlocked. To lock a resource an atomic instruction is used to implement a test-and-set operation. Each core continues to test and set a bit in shared memory until it finds that the value is zero (unlocked). The core that locks the resource changes the value to locked. The hardware lock is made in such a way that if multiple cores try to acquire the lock at the same time only one of them can actually lock the mutex. This principle allows for mutual exclusive access, but the access is not ordered. To allow for ordered resource access ERIKA provides functions as spin\_in() and spin\_out() that use the non-ordered access to the mutual exclusive hardware lock. The ordering is achieved by means of a queue. The access to the queue is protected by a mutex (unordered). Once registered in the queue the access to the shared resource is done by order of the queue.

ERIKA uses queues to make the shared resource access in order of arrival (of resource access request). Figure 5.14 illustrates an abstract view of the FIFO-queue. The queue consists of 3 parts:

1. The **global resource queue**, called "spin\_value" which is a matrix of size "(#cores) times (#resources)",

   as Figure 5.14 illustrates.

2. The **global tail of the queue**, called "spin\_status" which is an array of size (#resources), as Figure 5.14 illustrates. Hence, there is a queue per shared resource, where every queue can hold at most the number of cores as elements.

   This array contains in each entry previous spinning address + previous spin\_lock status.

3. The **local preceding cores pointer and polling value**

   A waiting core knows where to look in the "spin\_value" matrix to poll on the preceding polling bit, as Figure 5.14 illustrates.

   Note that the toggle bits are in global memory while the order consists of each core only knowing its direct predecessor. This means that a distributed implementation of the FIFO queue has been implemented.

---

This is a local value containing a pointer to a field in the global data "Spin\_value"
The execution of the functions spin\_in() and spin\_out()

In Figure 5.15 the call-graph of the function spin\_in() is given. The function spin\_in() is called from GetResource() and implements the busy waiting of the spin-lock protocol.

The polling address related to \( P_0 \) is the pointer to spin\_value with index \( P_0 \) (spin\_value\[R_q]\[P_0\]).

The polling value related to \( P_0 \) is the content inside spin\_value with index \( P_0 \) (spin\_value\[R_q]\[P_0\]).

The polling address related to preceding core is given by the tail pointer spin\_status upon entering the queue (spin\_status\[R_q\]), which later is stored in the local wait address.

The polling value related to preceding core is the content of the cell pointed by the pointer to the previous core (*spin\_status\[R_q]*/).

\[\text{Figure 5.15: The call-graph of spin\_in() \[84x770]\}]

Steps taken in spin\_in() are as follows:

1. Determines its own spinning address + spinning value.
2. Acquire the hardware lock.
3. Read the preceding spinning address + previous spinning value.
4. Store its own spinning address + spinning value at the tail of the queue.
5. Release the hardware lock.
6. Determines the previous spinning address (by removing last bit).
7. Determines the previous spinning value (by checking last bit).
8. Polls by comparing "the content of the previous spinning address" with "previous spinning value".

Steps taken in spin\_out():

1. Toggle its own polling value

A task requesting access checks the tail of the queue called "spin\_status" and determines the core that precedes the current core in the resource queue. The tail contains the value that represents locked (0 or 1) and the address in global memory where the preceding task will change the lock-status to unlocked upon release. In case the "preceding core’s spinning value" and "the contents of the preceding core’s spinning address" are equal the resource is locked. Initially and when the resource is unlocked the "preceding core its spinning value" and the "the contents of the preceding core its spinning address" differ.

The next step of the task requesting access is to add itself to the queue. It then places the address of its own polling field bitwise added with the value of its own polling field at the tail of the queue "spin\_status". The address of its own polling field is where the succeeding task will look while polling. The value of its own polling field is the value the succeeding task will use to compare with.

Finally the task will start spinning by continuously checking the polling bit/value at the address it is supposed to spin on in global memory with the value that means locked. As soon as the value gets toggled due to release by preceding task the access will be granted.

---

7Note that the steps taken are the same as presented in Section 5.6.3 on p. 35. The reason for providing the steps on p. 35 is to introduce the concept of spin\_in() and spin\_out() without considering the implementation details e.g. omitting variable names. The reason for repeating the steps on the current page 38 is to capture the behaviour of the code. The steps are pseudo code, for each line of c-code a step is presented. The same pseudo code is also used in the measurement section.

8Its own spinning address is the address where the current core \( P_k \) will let the next core eligible that requested access \( P_q \) know that the resource became available. The spinning value is the value that represents locked. The requesting core \( P_k \) adds itself to the resource queue at the tail of the resource queue i.e., in spin\_status. The bitwise combined spinning address and value are stored in spin\_status.

9The difference between items 6,7 and 3 is: At item 3 the combined value is read from the global data. Since the data is global it must be protected by the hardware locks to guarantee mutual exclusive access. To already split the data in the spinning address and spinning value would mean the hardware lock is unavailable for a longer amount of time. Since the hardware lock is shared between cores the access times determine the blocking time. While after the hardware lock is release (at item 5) the execution of the code does not contribute to the blocking time of other cores. Item 3 is reading the combined value, item 6,7 is splitting it into its respective parts.

10In code named what\_means\_locked.
Example spin\_in() and spin\_out(); initialization and request access by \textit{P}$_1$

The RT-druid initializes a lot of the data-structures. Figure 5.16a illustrates what the content of the data-structures used by spin\_in() and spin\_out() are at the moment of initialization. The spin\_value matrix contains the toggle bits for all cores and resource combinations. Initially all toggle bits are set to zero. Initially the spin\_status[\textit{R}$_q$] contains (address of spin\_value[\textit{R}$_q$][\textit{P}$_0$]) + 1. Which is the spinning address of core \textit{P}$_0$ for each resource. Since the entry of the spinning address (0) and the spinning value (1) differ the resource is unlocked. The arrows in the Figure 5.16a depict the pointers.

Each memory address is separated by 4 places, the binary representation does not change the last 2 bits. For instance address 0x4000 is binary 0100.0000.0000.0000 and the next address 0x4004 is 0100.0000.0000.0100. The reason for increasing by 4 instead of 1 is the size of an integer. The integer size is 4 bytes. Each address in memory contains only one byte. Thus if you would want to read the next value in an integer array the pointer needs to be increased by 4 bytes or 4 address places. Those 2 bits that are not used for indexing can be used to store some additional information. In our case it stores whether a 0 or a 1 means locked. Thus the value 4001, the initial value of spin\_status[0] is pointer 4000 + 1 (1 means locked).

Let us assume that a task on core \textit{P}$_1$ requests access to resource \textit{R}$_0$, as Figure 5.16b illustrates. Core \textit{P}$_1$ determines the new lock value, which will be used by the next task that requests access. The new lock value is “the current core its spinning address” + “the current core its spinning value” (0)\(^{11}\), thus 4004. Therefore 4004 will be used by the next task in the queue.

Core \textit{P}$_1$ reads the spin\_status[0] which is 4001, this is the pointer to the tail of the queue and the what means locked value. The 4001 is split in a pointer to the toggle field called wait\_address 4000 and the what means locked value 1. Now core \textit{P}$_1$ has in its local data a pointer to a toggle field of the task ahead of it in the queue and a value that represents locked.

For the next task in the queue the global data spin\_status[0] gets updated to the new lock value 4004. The core \textit{P}$_1$ starts spinning on the global address 4000 and compares it with the value what means locked. On core \textit{P}$_1$ the evaluation of (1==0) gives false the core directly escapes the while loop and access the resource \textit{R}$_0$.

In Figure 5.16a and 5.16b the local data contains a “-” sign. The local data is part of the function spin\_in() and only exists within the scope of the function. Only if the core has a request to access a resource the local data is available in memory. The “-” sign does not mean that the data is 1, 0 or undefined, it means that the data is non-existent anywhere in memory at that moment in time.

\(^{11}\)The current core its spin value should be the same as the value contained in the spinning address at the moment the task starts spinning (and during resource access). The value 0 or 1 is irrelevant as long as both values match. It is 0 due to initialization. Note that in Listing 5.9 line 5 EE\_hal\_spin\_value[\textit{m}][\text{EE\_CURRENTCPU}] is added as a binary value. This binary value is the spinning value of the current core and represents the value that denotes locked.
Example \texttt{spin\_in()} and \texttt{spin\_out()}; Access by \textit{P} 1, requests by \textit{P} 2 and \textit{P} 3

Figure 5.17a is shows what the content of the data-structures is after core \textit{P} 1 still has access to resource \textit{R} 0 and core \textit{P} 2 also requests access to the resource. Core \textit{P} 2 creates the new \texttt{lock\_value} that will be used by the next core in line. The new \texttt{lock\_value} is \textit{"the current core its spinning address" + "the current core its spinning value"}, 4008+0. Core \textit{P} 2 reads the spin\_status left behind by the core at the tail of the queue i.e., \texttt{(P)}, 4004. Core \textit{P} 2 determines that the \texttt{wait\_address} at which it will poll is 4004 and the value to which it will compare which represents \textit{"what means locked"} is 0. Core \textit{P} 2 updates the global data \texttt{spin\_status[0]} with its new \texttt{lock\_value} 4008. Core \textit{P} 2 starts comparing the value of the \texttt{wait\_address}, i.e., \texttt{*wait\_address with what\_means\_locked (0==0) is true}. The core is unable to escape the while loop and will keep spinning until the value gets changed by core \textit{P} 1. We now have a queue of \textit{P} 1 at the head accessing the resource and \textit{P} 2 waiting for release.

Figure 5.17b illustrates what the content of the data-structures is after core \textit{P} 1 still has access to resource \textit{R} 0, Core \textit{P} 2 is waiting and core \textit{P} 3 also requests access to the resource. Core \textit{P} 3 creates the new \texttt{lock\_value} that will be used by the next core in line. The new \texttt{lock\_value} is the pointer to its own toggle field + the bit that represents locked, 4012+0. Core \textit{P} 3 reads the spin\_status left behind by the Core at the tail of the queue i.e., \texttt{(P)}, 4008. Core \textit{P} 3 determines that the \texttt{wait\_address} on which it will poll is 4008 and the value to which it will compare is 0. Core \textit{P} 3 updates the global data \texttt{spin\_status[0]} with its new \texttt{lock\_value} 4012. Core \textit{P} 3 starts comparing *wait\_address with what\_means\_locked (0==0) which is true. The core is unable to escape the while loop and will keep spinning until the value gets changed by core \textit{P} 2. We now have a queue of \textit{P} 1 at the head accessing the resource, \textit{P} 2 and \textit{P} 3 waiting for release.

![Figure 5.17: The data-structures used by \texttt{spin\_in()} and \texttt{spin\_out()}](image-url)
Example spin_in() and spin_out(): P1 releases access, in queue are P2 and P3
Figure 5.18a illustrates what happens when core P1 releases the resource R0. The only thing that happens is that core P1 toggles the global data in spin_value[0][1], e.g., here from 0 to 1. We now have a queue of P2 at the head which has no access to the resource yet and P3 waiting for release.

(a) Core P1 releases R0, core P2 and P3 wait for release Figure 5.18: The data-structures used by spin_in() and spin_out()

(b) Core P2 uses R0, core P3 waits for release

Figure 5.18b illustrates what happens when core P2 reads that the resource is released. Core P2 actively polls on the global data spin_value to check whether the value pointed by wait_address stored in its local data has changed. Since core P1 just changed the value to 1 the comparison comparing *wait_address with what means locked (1 == 0) is false. The core P1 is now able to escape the while loop and acquires access to the resource R0 We now have a queue of P2 at the head accessing the resource and P3 waiting for release.

Example spin_in() and spin_out(): Access by P2, requests by P3 and P1
Figure 5.19 illustrates what the content of the data-structures is after core P2 still has access to resource R0. Core P3 is waiting and core P1 also requests access to the resource. Core P1 creates the new lock_value that will be used by the next core in line. The new lock_value is the pointer to its own toggle field + the bit that represents locked, 4004 + 1. Note that 1 is added, this means that for core P0 the value representing locked has changed from 0 to 1. Core P1 reads the spin_status left behind by the core at the tail of the queue (P3), 4012. Core P1 determines that the wait_address on which it will poll is 4012 and the value to which it will compare is 0. Core P1 updates the global data spin_status[0] with its new lock_value. Core P1 start comparing *wait_address with what means locked (0 == 0) is true. The core is unable to escape the while loop and will keep spinning until the value gets changed by core P3. We now have a queue of P2 at the head accessing the resource, P3 and P1 waiting for release.
Ordered or unordered access

To allow for ordered resource access we provide functions as spin_in() and spin_out() that use the non-ordered access to the mutual exclusive hardware lock. The effect is that if multiple cores request access to the same resource, the entering to the queue is done randomly. If 2 tasks $\tau_1$ and $\tau_2$ request access to $R_1$ at the same time, one of the tasks for instance $\tau_1$ locks the mutex and can put itself in the resource queue. The request of task $\tau_2$ is not lost, the task keeps trying to acquire the lock i.e., the task is blocked. If a task $\tau_3$ requests access to the resource before $\tau_1$ releases the mutex hardware lock (not the resource), it is possible that $\tau_3$ locks the mutex before $\tau_2$ has a chance to acquire the hardware lock. The queue becomes $\tau_1$, $\tau_3$ and $\tau_2$. Once a task is in the queue the order is preserved.
CHAPTER 5. THE ERIKA OS

5.7 Remote notification

This section contains the description of the functions used for remote notification (RN). Communication between cores can be done via shared memory or by inter core interrupts (which also use shared memory). Both the sending and the receiving of an interrupt is explained in this section. It is beneficial to understand how the inter-core interrupts work since they will be used in the RAP that we implement in later sections of this document.

\texttt{rn\_send()} p. 45 The function \texttt{rn\_send()} is used on the sending core to send an RN to another core. Sending of an RN is done non-interruptible. The sending core acquires a spin lock on the RN data-structures of the receiving core. The RN data-structures consist of a control part and an RN message data part. Once the control and data is adjusted accordingly the lock is released. An interrupt is send to the receiving core by a rising edge on the I/O port.\footnote{The Type of RN to be send are: 0x1= EE\_RN\_COUNTER will increase a counter on the receiving core, 0x2= EE\_RN\_EVENT will inform the receiving core of a event that happened remotely, 0x4= EE\_RN\_TASK will activate the task on the receiving core, 0x8= EE\_RN\_FUNC will start executing a function on the receiving core. Some other types of RN are reserved (0x10 and 0x20) in other kernel types but are unavailable in our kernel (allow for remote assigning of server budgets). Information on how the type of RN is send can be found on page 44.}

\texttt{rn\_handler()} p. 47 The function \texttt{rn\_handler()} is used on the receiving core to handle the incoming RN. Handling of an RN is done non-interruptible. The receiving core acquires a spin lock on the RN data-structure of the core itself. By setting a bit in the control part of the RN data-structures any sending core will write on a different place than where the receiving core reads its RN messages. The spin lock is released. The core reads the RN messages and starts executing them. After handling the RN the receiving core checks whether a new RN has occurred in the meanwhile.

The RN message buffer contains a dual buffer. A receiving core controls in which of the 2 buffers a sending core puts its message. By not allowing other cores to influence one of the two message buffers a receiving core can read the buffer without a mutex (it already has exclusive access).

\texttt{rn\_execute()} p. 50 The function \texttt{rn\_execute()} performs the action given by the RN. In case of a remote activation of a task, the task is put into the ready queue and possibly becomes the executing task.
Hardware related to RN

Figure 5.20 illustrates the hardware connections between the cores. Each core has one input to receive its interrupts and 4 outputs to send to each other core. The hardware allows to send an interrupt to the core itself, the software will perform a check and prevent this from happening. The 4 outputs are already by interconnects combined inside the hardware description. The connection between input and output is done in our setup by a cable but could be done in VHDL code of the hardware description.

Listing 5.10 shows the sending of the signal from output to input pin. Listing 5.11 shows that the function contains the writing of a value into a register. The function IOWR_ALTERA_AVALON_PIO_DATA() is part of the HAL layer, and makes an abstraction of what happens in the hardware. What actually takes place is writing on address: “base” the value: “data”. The first argument given is a register connected to a driver that controls the I/O. The data put there is the cpu bit shifted, thus 0b 1000 for P3 and 0b 0001 for P0.

The two instructions in Listing 5.10 first make the signal of the output pin of the sending core, and as a result the input pin of the receiving core, high. The second instruction makes the signal low, creating both a rising and falling edge on the input of the receiving core. The hardware description is configured in such a way that a rising edge on its input will generate an interrupt on the receiving core. In case interrupts are enabled the interrupt handler (IH) is executed. In case interrupts are disabled the moment the interrupt arrives, a flag is set to indicate an interrupt had occurred. When interrupts are enabled again the core starts handling the interrupts flags in order of priority.
CHAPTER 5. THE ERIKA OS

RN Data-structures

Figure 5.21 illustrates the data structures used by sending and receiving RN. All data in Figure 5.21 is shared between cores. For sending and receiving messages ERIKA makes use of 2 message buffers. The reason is that when a core is reading the received messages in lets say buffer 1, a sending core can add messages by putting them in buffer 2.

The RN data-structures consist of a control part and a message part. The control part is shown on the left. Each core has a bit-pattern consisting of 3 bits that contains the RN status of the core.

1. The 1st bit indicates the RN message buffer a sending core should write on. Only a receiving core \( P_k \) sets the 1st bit of \( \text{EErn.switch}[P_k] \), other cores read which buffer can be used to store new RN.
2. The 2nd bit indicates whether the receiving core is already in the RN handler handling RN requests. Core \( P_k \) sets the 2nd bit of \( \text{EErn.switch}[P_k] \) upon entering the RN and reset it upon exit. Sending cores read the bit to determine whether they should send an interrupt to the receiving core.
3. The 3rd bit indicates whether a new RN has arrived in the meanwhile. Sending cores \( (P_y \neq x) \) sets the 3rd bit of \( \text{EErn.switch}[P_k] \) when the receiving core is already in RN handling (its 2nd bit was set). The receiving core \( P_k \) checks the 3rd bit upon leaving the RN handler. If the 3rd bit (toggles the 1st bit) is set the core \( P_k \) resets the 3rd bit and performs the RN handler again. If the 3rd bit is not set the core \( P_k \) exits the RN.

The message part of the RN data-structures is duplicated. The data-items contain an index \( \text{SW} \) which is the 1st bit of \( \text{EErn.switch}()[ ] \) the control bit pattern. The value of \( \text{SW} \) determines whether the first half of the message buffer is used or the second half. The variable \( \text{rn} \) denotes the target task (its thread id). The RN buffer consists of \( \text{EErn.first}[\text{SW}] \), \( \text{EErn.next}[\text{rn}][\text{SW}] \), \( \text{EErn.type}[\text{rn}][\text{SW}] \) and \( \text{EErn.pending}[\text{rn}][\text{SW}] \). The ordering of the RN messages is determined by \( \text{EErn.first}[\text{SW}] \) (the last added RN msg) and \( \text{EErn.next}[\text{rn}][\text{SW}] \). The RN buffer is a linked list and \( \text{EErn.next}[\text{rn}][\text{SW}] \) keeps track of the order by means of \( \text{rn} \). \( \text{EErn.type} \) stores bit-wise the types of RN. Type of RN \( \in \{ \text{EE_RN_COUNTER(1)}, \text{EE_RN_EVENT(2)}, \text{EE_RN_TASK(4)}, \text{EE_RN_FUNC(8)} \} \). \( \text{EErn.pending} \) contains the number of pending requests. If a core tries to send an RN to a core of a target task that already exist then the pending counter will be increased.

RN initialization

The RN status \( \text{EErn.switch}()[ ] \) contains all zero, which indicates the use of message buffer 0 and the core is not in an RN and no RN has happened yet. The RN message buffer ordering \( \text{EErn.first}[\text{SW}] \) and \( \text{EErn.next}[\text{rn}][\text{SW}] \) contains -1 to indicate there is no pending RN. The RN message buffer type \( \text{EErn.type}()[ ] \) contains 0, there does not exist an RN of type 0 therefore 0 is a good initial value. The RN message buffer amount of pending requests \( \text{EErn.pending}[\text{rn}][\text{SW}] \) contains 0, there are no pending request initially.
5.7.1 Remote notification: \texttt{rn\_send()}

Here we explain how the ERIKA implementation of \texttt{rn\_send()} works. The \texttt{rn\_send()} function is used to send a remote notification (RN) to another core. To implement a new RAP we later decide to use the \texttt{rn\_send()} function.

\texttt{rn\_send(rn, t, par)}

\texttt{rn} is the task ID of the target task (for example the task to be activated in case of remote activation)
\texttt{t} the type of RN \( t \in \{ \text{COUNTER, EVENT, TASK, FUNCTION, BIND, UNBIND} \} \)
\texttt{(we do not use any of the provided \texttt{rn} types in our system model)}
\texttt{struct par{}
    \texttt{pending} the amount of pending RN for task \texttt{rn} of type \texttt{t}
    \texttt{ev} bitmask of the event type, used for remote events, (we do not use ev)
    \texttt{vers} contract type used for binding to assign a budget to a server and task (we do not use vers)
}\texttt{}}

Figure 5.22a illustrates the execution of the \texttt{rn\_send()} function. Figure 5.22b illustrates the call-graph of the \texttt{rn\_send()} function. The \texttt{rn\_send()} function is used to communicate between cores. The \texttt{rn\_send()} function is non-preemptive. First the message buffer of the receiving core is filled. To get access to the RN message buffer of a receiving core the sending core acquires a spin lock on the RN message buffer of a receiving core. The RN message is stored is the RN data-structures illustrated in Figure 5.21 and the spin-lock is released.

The sending core checks in the RN control data of the receiving core whether the core is already handling a RN. If the receiving core is not yet handling a RN then an inter core interrupt is send (Figure 5.20). If the receiving core is already handling a RN it will check the message buffer anyway, and the sending core does not need to send an additional RN. The interrupts get enabled.
CHAPTER 5. THE ERIKA OS

Listing 5.12: The pseudo code of the function rn_send()

```c
rn_send(index RN data-structures, type of RN, parameter of RN){
    Disable interrupts
    Determine the receiving core ID.
    If we try to send an RN to the core itself{
        Enable interrupts
        Return to function that called rn_send()
    } Otherwise {
        Acquire spin lock of RN data-structures of the receiving core
        Determine which of the 2 data-structures of the RN message is not in use by receiving core
        Check if the receiving core is already inside the IH, else we raise a new inter-core interrupt
        If the receiving core is already executing its IH than it has to go through the cycle again
        If there is no RN request the RN type present{
            Update the linked list that keeps track of the pending type of RN by linking the first RN type
            Put the RN-type we try to send at the head of the linked list (rn_first)
        }
        Increase the pending counter
        Set the bit in a bitmask contain the type of RN send to the receiving core.
        Release spin lock of RN data-structure receiving core
        If the receiving core is not handling RN send an RN
    } Enable interrupts
}
```

Listing 5.13: The c-code of the function rn_send()

```c
int EE_rn_send(EE_TYPERN rn, EE_TYPERN t, EE_TYPERN_PARAM par){
    register EE_UINT8 cpu;
    register EE_TYPERN_SWITCH sw;
    register int newIRQ;
    register EE_FREG flag;
    flag = EE_hal_begin_nested_primitive();
    cpu = EE_rn_cpu[rn]; /* The core to which the RN will be send
    if (cpu == EE_CURRENTCPU) { /* The core is trying to send an RN to itself
        EE_hal_end_nested_primitive(flag);
        return 1;
    } else {
        EE_hal_spin_in(EE_rn_spin[cpu]); //spin Lock acquisition
        sw = EE_rn_switch[cpu] & EE_RN_SWITCH_COPY;
        /* Check if the receiving core is already inside the IH, else we raise a new inter-core interrupt
        newIRQ = !(EE_rn_switch[cpu] & EE_RN_SWITCH_INSIDEIRQ) && EE_rn_first[cpu][sw] == -1;
        /* If the receiving core is already executing its IH than it has to go through the cycle again
        if (EE_rn_switch[cpu] & EE_RN_SWITCH_INSIDEIRQ){ EE_rn_switch[cpu] |= EE_RN_SWITCH_NEWRN;}
        if (!EE_rn_type[rn][sw]) { //request is not yet queued, insert into the pending requests
            EE_rn_next[rn][sw] = EE_rn_first[cpu][sw];
            EE_rn_first[cpu][sw] = rn;
        }
        EE_rn_pending[rn][sw] += par.pending; //increase the pending counter
        EE_rn_type[rn][sw] |= t; //set the type in the remote notification
        EE_hal_spin_out(EE_rn_spin[cpu]); //spin Lock release
        /* Inter-processor interrupt. We raise an interprocessor interrupt only if there is not a
        similar interrupt pending. Note that the irq is raised before releasing the spin lock */
        if (newIRQ){ EE_hal_IRQ_interprocessor(cpu);}
    }
    EE_hal_end_nested_primitive(flag);
    return 0;
}
```
5.7.2 Remote notification: \texttt{rn\_handler()}

Here we explain how the ERIKA implementation of \texttt{rn\_handler()} works. The \texttt{rn\_handler()} function is used to handle the RN on a receiving core. The function \texttt{rn\_handler} receives the RN messages, calls an \texttt{rn\_execute()} function to perform the execution of the RN and afterwards checks if additional RN-messages have arrived. To implement a new RAP we later decide to use the \texttt{rn\_handler()} function.

Listing 5.14 shows the code that starts executing the moment the interrupt arrives. During the initialization (Section 5.2) we registered the function \texttt{EE\_nios2\_IIRQ\_handler()}. The function \texttt{EE\_nios2\_IIRQ\_handler()} first disables most interrupts (allows only interrupts with a higher priority than the IPIC interrupt). The function \texttt{EE\_nios2\_IIRQ\_handler()} calls the \texttt{rn\_handler()} and before returning enables all the interrupts again.

```c
1 /* During startup the interrupt routine is registered in the right interrupt register. 
2 When an interrupt happens from the IPIC input the core starts executing EE_nios2_IIRQ_handler 
3 alt_irq_register( EE_IPIC_IRQ, 0, EE_nios2_IIRQ_handler); 
4 */ The implementation of EE_nios2_IIRQ_handler 
5 static void EE_nios2_IIRQ_handler(void *arg, alt_u32 intno)
6 alt_u32 pending;
7 pending = alt_irq_interruptible(EE_IPIC_kIRQ); 
8 rn_handler(); 
9 alt_irq_non_interruptible(pending); 
10 }
```

Listing 5.14: Handling of a remote notification

![Diagram of rn\_handler()](a) Diagram of rn\_handler()

![Call-graph of rn\_handler()](b) The call-graph of rn\_handler()

Figure 5.23: The rn\_handler() function
CHAPTER 5. THE ERIKA OS

Figure 5.23a illustrates the steps the rn_handler() goes through. Figure 5.23b illustrates the call graph of the function rn_handler(). The receiving core has a control data-structure containing the status of the RN handling and message data-structures containing the incoming RN. The receiving core disables its interrupts completely and acquires a spin lock on its rn data-structures. The control bits are adjusted to indicate the core is currently handling RN and any sending core should use its other RN message buffer. The spin-lock is released and interrupts are enabled again (remember only interrupts with a priority higher than the IPIC).

The core can now access its message buffer without read write problems with other cores. Any sending core will use the other message buffer (as illustrated in Figure 5.21). The function rnexecute is called to execute each of the RN in the RN message buffer. Once all the RN of the current message buffer are finished the head of the message buffer (a linked list) is set to -1 to indicate an empty list.

All the interrupts get disabled. The core acquires the hardware lock on its own RN data-structure. Now the core checks in the RN control data whether any additional RN requests arrived. If there are new RN requests in the second RN message buffer then the rn_handler() function will stay in the while loop and restart the rn_handler() function from the beginning. If there are no pending RN the rn_handler will reset the RN control bit to indicate the core is no longer handling RN. The IPIC interrupt flag in the interrupt table is reset such that new IPIC interrupts can arrive.

RN handling is done in a last-in-first-out fashion rather than first-in-first-out. Hence, the order is not preserved. Listing 5.15 shows the pseudo code of the rn_handler() function. Listing 5.16 shows the c-code of the rn_handler() function.

```c
rn_handler()
While there remain pending RN{
    Retrieve the current filled RN message buffer index SW
    Disable interrupts
    Acquire_spin_lock
    The RN control data-structure rn_switch[Ci] toggles the COPY bit and sets the INSIDEIRQ bit.
    /* To indicate that any sending core should use the other message buffer
    * and that the core is currently handling RN
    */
    Release_spin_lock
    Enable interrupts
    For as many as there are pending RN in the linked list RN message buffer indexed with SW{
        Execute the RN
    }
    Make the index to the beginning of the linked list RN message buffer equal to -1 indicating empty
    Disable interrupts
    Acquire_spin_lock
    If another core has queued additional RN request{
        Reset the bit in the RN control data, that indicates that a new RN arrived while inside the RN
        There remain pending RN
    } Otherwise{
        There remain NO more pending RN
        Reset the bit in the RN control data, that indicates that were inside the interrupt routine.
        Reset the interrupt routine flag to signal the IH has been handled
    }
    Release_spin_lock
    Enable interrupts
}
```

Listing 5.15: The pseudo code of the function rn_handler()
void rn_handler(void) { /* this function is only called when there are pending RN for the processor */
    register EE_TYPERN current;
    register EE_TYPERN_SWITCH sw;
    int redo = 1; /* indicates whether there are multiple pending RN */

    /* EE_rn_switch[CPU] contains information about the queuing data-structure should be used by the rn_send function. Init value=0. Flags: -EE_RN_SWITCH_COPY (bit 0) which copy of the data structures -EE_RN_SWITCH_INSIDEIRQ (bit 1) inside or not an interrupt handler -EE_RN_SWITCH_NEWRN (bit 2) a new remote notification arrived while inside an interrupt handler */
    while (redo) {
        sw = EE_rn_switch[EE_CURRENTCPU] & EE_RN_SWITCH_COPY;
        /* The RN data structures are duplicated, sv indicates which data-structure is to be used. */
        EE_hal_IRQ_begin_primitive(); // all interrupts get disabled
        spin_in(EE_rn_spin[EE_CURRENTCPU]); // Spin Lock acquisition
        /* switch pending requests and set the inside irq flag */
        EE_rn_switch[EE_CURRENTCPU] &= (EE_rn_switch[EE_CURRENTCPU] & EE_RN_SWITCH_COPY) | EE_RN_SWITCH_INSIDEIRQ;
        spin_out(EE_rn_spin[EE_CURRENTCPU]); // Spin Lock release
        EE_hal_IRQ_end_primitive(); // all interrupts get enabled
        /* the EE_rn_switch[EE_CURRENTCPU] is now ready for other cores to store additional RN. */
        /* Note: from now on, we can assume that all the data-structures indexed by sw can be accessed without need of interrupt disabling or spin locks. The only requirements that must be fulfilled is that this function cannot be interrupted again by itself (i.e., once an interprocessor interrupt is raised, it cannot be raised again until this function has completed. */
        /* Execute the rn requests */
        for (current = EE_rn_first[EE_CURRENTCPU][sw]; current != -1; current = EE_rn_next[current][sw]) {
            EE_rn_execute(current, sw); // here the activation of the task takes place
        }
        EE_rn_first[EE_CURRENTCPU][sw] = -1;
        /* now we have to guarantee that we can exit from the Interprocessor interrupt. That is, the other cores should not have queued any other request. To do that, we have to reacquire again the spin lock to check the appropriate conditions. Moreover, we have to do that with interrupt disabled, and we have to be sure that we will exit the handler with INTERRUPT DISABLED. */
        EE_hal_IRQ_begin_primitive();
        spin_in(EE_rn_spin[EE_CURRENTCPU]);
        /* if the other processor has queued another request */
        if (EE_rn_switch[EE_CURRENTCPU] & EE_RN_SWITCH_NEWRN) {
            /* reset the newRN flag */
            EE_rn_switch[EE_CURRENTCPU] &= ~EE_RN_SWITCH_NEWRN;
            /* redo the dispatching of the remote notifications */
            redo = 1;
        } else {
            /* we can exit the interrupt */
            redo = 0;
            /* set that we are no more inside the interrupt */
            EE_rn_switch[EE_CURRENTCPU] &= ~EE_RN_SWITCH_INSIDEIRQ;
            EE_hal_IRQ_interprocessor_served(EE_CURRENTCPU);
        }
        /* Spin Lock release */
        spin_out(EE_rn_spin[EE_CURRENTCPU]);
        /* note: we end the handler with interrupt disabled */
        if (redo) EE_hal_IRQ_end_primitive();
    }
}

Listing 5.16: The c-code of the function rn_handler()

The whole function EE_rn_handler() shown in Listing 5.12 can be separated into 3 parts.
1. Reseting the EE_rn_switch[cpu] to be able to receive additional RN (line 13-26).
2. Execute all the pending RN (line 28-32).
3. Check if in the meantime an RN has happened, then restart the function otherwise exit (line 34-57).
5.7.3 Remote notification: \texttt{rn\_execute()}

Here we explain how the ERIKA implementation of \texttt{rn\_execute()} works. The \texttt{rn\_execute()} function is used to execute the received RN. To implement a new RAP we later decide to use and change the \texttt{rn\_execute()} function. In the requirement, specification and design part of this document we will add an additional type of RN to be executed in the function \texttt{rn\_execute()}.

The function \texttt{rn\_execute} is a case distinction on the type of RN. Depending on the type of kernel (in our case OO) certain types of RN are supported. The type of RN is determined and the according action is performed. For example if the type of RN is a remote activation of a task then the \texttt{ActivateTask()} function is called.

Listing 5.17 shows the pseudo code of the \texttt{rn\_execute()} function. Listing 5.18 shows the c-code of the the \texttt{rn\_execute()} function.

```c
void rn_execute(EE_TYPERN rn, EE_UINT8 sw){
  register EE_UREG pend;
  if (EE_rn_type[rn][sw] & EE_RN_TASK) {
    for (pend = EE_rn_pending[rn][sw]; pend; pend--) {
      ActivateTask(EE_rn_task[rn]);
    }
    EE_rn_pending[rn][sw] = 0;
    EE_rn_type[rn][sw] &= ~EE_RN_TASK;
  }
  #ifdef __RN_FUNC__
  if (EE_rn_type[rn][sw] & EE_RN_FUNC) {
    for (pend = EE_rn_pending[rn][sw]; pend; pend--) {
      ((void (*)(void))(EE_rn_func[rn]))();
    }
    EE_rn_pending[rn][sw] = 0;
    EE_rn_type[rn][sw] &= ~EE_RN_FUNC;
  }
  #endif
}
```

Listing 5.18: The c-code of the function \texttt{rn\_execute()}

Thursday 30$^\text{th}$ June, 2016 18:15 50 Performance of resource access protocols
Chapter 6

PROJECT

Embedded real time systems should produce the correct output in time with constrained resources. Resource access protocols (RAP) regulate the mutual exclusive access of tasks to resources. Access to a local resource (shared among tasks on one core) or global resource (shared among tasks on different cores) may lead to additional delay due to blocking. Originally two approaches existed i.e., spin-based and suspension-based resource access. Under a spin-based RAP the task $\tau_i$ requesting access to a locked resource $R_q$ places its request in $R_q$’s resource queue and waits non-preemptive (highest priority HP). Under a suspension-based RAP the task requesting access to a locked resource places its request in the resource queue and releases the core (lowest priority LP). Several new flexible spin-lock RAP (priority between LP and HP) are proposed by Sara et al. in [2, 3]. The aim of this thesis is to specify, design, implement and measure the RAP in the range of CP (highest global resource ceiling) to HP (Highest Priority). The measurement consists of measuring execution times of the kernel code that is required to implement the flexible spin-lock protocol with priorities in the range of CP to HP. We measured overhead terms that occur due to resource access, we did not measure the completion times/schedulability of task-sets.

Figure 6.1a illustrates the different parts of the report. The ERIKA kernel layer consists of 6 function groups represented by the blocks 6.2 till 6.7 denoting the sections they are explained. In Section 5.1.3 p.24 Figure 5.3 is illustrated what the content is of each function group. Creating the new RAP is done incrementally in three separate steps. An introduction to those steps is given in this chapter. The design method we use during the project is the v-model or ISO 12207 standard (and ISO/IEC/IEEE standard 29148:2011). To create the final system we go through the V-model iteratively for each of the three design steps.

![Diagram](image.png)

(a) Explanation of symbols used in the top right corner of the page  
(b) The v-model

Figure 6.1: Steps that need to be implemented in the ERIKA kernel

- 5.2 Initialization
- 5.3 Task handling
- 5.4 Ready queue
- 5.5 Stack & context switch
- 5.6 Resource access
- 5.7 Remote Notification

1. STEP 1 Flexible priority tool
2. STEP 2 Spinning on a local spin_lock
3. STEP 3 Context switch

- Requirements
- Specification
- Design
- Implementation
- Check invariants
- Measurements

In the "V"-model you typically have tests in the right leg of the V (rather than "measurements" and "check invariants"). Note that a verification step unit-testing is grey. We do not perform the unit-testing as an explicit section in the report. The unit testing can be considered the testing of the incremental steps (1,2,3) of the project. In each of those steps as a part of the specification we provide invariants. After the design of each step we check whether the invariants hold. The measurements can be considered a system test, the test is not extensive to proof all properties of the system.

---

1A dedicated resource queue is associated with every global resource.
6.1 Project overview
Here we present an overview and introduction of the new RAP. We decided to split the work up in different parts
and make a requirement, specification, design, invariant check and measurements for each of those 3 steps. The steps are:
1. A tool to set the flexible spin priority. 2. Sending a remote notification to the task at the head of the resource queue (if any) to notify the release of a shared global resource. 3. Granting the task that requested the global resource non-preemptive access to the resource, optionally causing a context switch between the currently running task and the blocked task. After the three steps are completed measurements of the new flexible spin-lock protocol are performed.

Figure 6.2a illustrates the time-line as a result of the MSRP/HP protocol. Figure 6.2b illustrates the timeline
as expected for the flexible spin-lock protocol. The task set consists of 4 tasks, \( \tau_0 \) on core \( P_0 \) and \( \tau_1, \tau_2 \) on core \( P_1 \). Note that the priority of task \( \tau_i \) is \( \rho_i \). For this example we have chosen to make the priority of the tasks equal to the task ID. A higher priority task has a higher number. In ERIKA the priority of a task \( \tau_i \) with priority \( \rho_i=j \) is represented by \( 2^j \). The system ceiling in the figures is additive, if \( \tau_i \) with \( \rho_i=2^2=0x1 \) gets pre-empted by \( \tau_1 \) with \( \rho_1=2^1=0x2 \) then the new system ceiling becomes 0x3. Changes to the priority of the executing task take place by changing the system ceiling. When the task becomes non-preemptive due to resource access or in the flexible spin-lock implementation a task spins on a priority different from the dispatch priority, the system ceiling is adjusted accordingly. A task becomes non-preemptive by setting 0x80 the 8th bit in the system-ceiling.

In the MSRP/HP time-line \( \tau_0 \) requests a resource which is unavailable. Thus the task increases its priority by increasing the system ceiling above the non-preemptive priority 0x80. The system ceiling becomes \( \rho_{\max} + \rho_0 = 0x80 + 0x1 = 0x81 \). Task \( \tau_1 \) stays the executing task even though tasks \( \tau_1 \) and \( \tau_2 \) arrive and end up in the ready queue. The resource is released at time \( t=8 \) by \( \tau_3 \) on core \( P_1 \) by setting a bit in global memory. Since task \( \tau_0 \) is active and continuously spinning on the global spin_lock it notices the release. Thus \( \tau_0 \) locks the spin-lock at time \( t=8 \) and accesses the resource at time \( 8 < t < 12 \). Task \( \tau_0 \) releases the spin-lock at time \( t=12 \) and returns the system ceiling to the priority before locking the resource (0x1).

Since there exists a task \( \tau_2 \) at the head of the ready queue that has a higher priority than the system ceiling, task \( \tau_0 \) gets preempted. Task \( \tau_2 \) becomes the executing task at time \( t=12 \) and the system ceiling is increased to priority \( \rho_2 + \rho_0 = 0x4 + 0x1 = 0x5 \). Task \( \tau_2 \) finishes its execution at time \( t=16 \) and the system ceiling is returned to the priority of the task on the stack \( \rho_0 = 0x1 \). The task at the head of the ready queue \( \tau_1 \) has a priority higher than the system ceiling \( 0x2 > 0x1 \). The task \( \tau_1 \) becomes the executing task and the system ceiling is increased to \( \rho_1 + \rho_0 = 0x2 + 0x1 = 0x3 \). The task \( \tau_1 \) requests access to the resource at time \( t=17 \) and since the resource is available it locks the spin-lock and increases the system ceiling. Task \( \tau_1 \) accesses the resource at time \( 17 < t < 19 \) and releases the spin-lock and returns the system ceiling to the priority before locking the resource at time \( t=19 \). Task \( \tau_1 \) completes its execution at time \( t=21 \) and the system ceiling is returned to the priority of the task at the head of the stack \( \tau_0 \) i.e., \( \rho_0 = 0x1 \). When task \( \tau_0 \) completes the system ceiling returns to 0x0 and the main program becomes the active thread (no task is running).

A task starts spinning in the original ERIKA-kernel with a non-preemptive priority. Instead we want a task to spin with a priority between \( CP \) and \( HP \). In Figure 6.2b \( \tau \) spins with a priority \( \rho_{spin} = 0x2 \).
Figure 6.2b illustrates the flexible spin-lock time-line, at time \( t = 3 \) the resource is requested by \( \tau_0 \) but it is not available. The task increases its priority by increasing the system ceiling to the spin priority of \( \tau_0 \), \( \rho_{\text{spin,0}} + \rho_0 = 0x2 + 0x1 = 0x3 \). Task \( \tau_0 \) stays the executing task even though task \( \tau_2 \) arrives at time \( t = 4 \) and ends up in the ready queue. When task \( \tau_2 \) arrives at time \( t = 6 \) it is put at the head of the ready queue. Since \( \tau_2 \) has a higher priority than the system ceiling, \( 0x4 > 0x3 \), the executing task \( (\tau_0) \) gets preempted and \( \tau_2 \) becomes the executing task. The system ceiling gets increased to \( \rho_{\text{spin,0}} + \rho_2 + \rho_0 = 0x4 + 0x2 + 0x1 = 0x7 \).

At time \( t = 8 \) the resource is released on core \( P_0 \) and an interrupt is sent to \( P_1 \) to denote the release. The task requiring the resource, i.e., \( \tau_0 \) is not the active task on \( P_0 \). Thus a context switch takes place to preempt the executing task \( \tau_2 \) and resume the spinning task \( \tau_0 \). The system ceiling representing the priority the spinning task is executing on gets increased to the highest (non-preemptive) priority \( \rho_{\text{max}} + 2 = 0x80 + 0x7 = 0x87 \). Task \( \tau_0 \) locks the spin-lock at time \( t = 8 \) and accesses the resource at time \( 8 < t < 12 \), releases the spin-lock at time \( t = 12 \) and resets the bits in the priority bit-mask related to the spin and the non-preemptive priority. The system ceiling becomes \( \rho_2 + \rho_0 = 0x5 \).

The system ceiling is higher than the priority of the task at the head of the ready queue \( 0x5 > 0x2 \), i.e., the executing task or an other task on the stack has the highest priority. The dispatch priority of the executing task is lower than the system ceiling \( \rho_0 < 0x5 \) thus the task that was last preempted needs to be resumed.

A context switch takes place preempting task \( \tau_0 \) and resuming \( \tau_2 \). The task \( \tau_2 \) becomes the executing task at time \( t = 12 \) and the system ceiling remains unchanged at \( \rho_2 + \rho_0 = 0x5 \). The task \( \tau_2 \) finishes its execution at time \( t = 14 \) and the system ceiling is returned to the priority of the task on the stack \( \rho_0 = 0x1 \). The task at the head of the ready queue \( \tau_1 \) \( (\rho_1 = 0x2) \) has a priority higher than the system ceiling \( 0x1 \). Task \( \tau_1 \) becomes the executing task and the system ceiling is increased to \( \rho_1 + \rho_0 = 0x3 \). Task \( \tau_1 \) requests access to the resource at time \( t = 15 \) and locks it since it is available at time \( t = 15 \). Task \( \tau_1 \) accesses the resource at time \( 15 < t < 17 \), releases the spin-lock and returns the system ceiling to the priority before locking the resource at time \( t = 17 \). Task \( \tau_1 \) completes its execution at time \( t = 15 \). The system ceiling is returned to the priority of the task at the head of the stack \( \tau_0 \). When task \( \tau_0 \) completes the system ceiling returns to \( 0 \) and the main program becomes the active thread.

**Step 1. Flexible priority tool** Chapter 8 p.62 When task \( \tau_0 \) in Figure 6.2b starts spinning on the resource at \( t = 3 \) the system ceiling should get increased with the flexible spin priority. The flexible spin priorities need to be derived from the task characteristics. We decided to provide a tool that reads the priorities of the tasks and resource usages from the system files. The tool calculates the \( CP \), \( CP \) and \( HP \) priorities and shows them to the user. The user subsequently sets the spinning priority for each task from the range of \( [CP, HP] \).

**Step 2. Spinning on a local spin_lock** Chapter 7 p.55 In the original ERIKA kernel, any task that becomes blocked on a resource gets the non-preemptive priority. Thus the moment a resource becomes blocked on a resource gets the non-preemptive priority. Thus the moment a resource becomes available the task requiring the resource is active spinning. The task releasing the resource \( R_0 \) on another core unlocks \( R_0 \) by toggling the global spin_lock. The blocked task is still continuously spinning on the global spin_lock to check whether the resource becomes available. Since the toggle bit is checked right after it is unlocked the waiting task can directly resume its execution.

In the new situation as shown in Figure 6.2b there is not necessarily a task continuously spinning on the local spin-lock, e.g. task \( \tau_0 \) is not spinning at the moment the resource gets released at time \( t = 18 \). Task \( \tau_2 \) needs to be preempted and task \( \tau_0 \) should get resumed. Since the task on core \( P_0 \) is not checking the (global) spin_lock it needs to be informed in another way. We want to do this by means of an interrupt from the releasing core \( (P_0) \) to the requesting core \( (P_1) \). There is already functionality present in the ERIKA kernel to send and receive an interrupt by means of a remote notification.

**Step 3. Context switch** Chapter 9 p.88 The moment the interrupt arrives at the receiving core it needs to preempt task \( \tau_2 \) and resume task \( \tau_0 \). To facilitate this preemption the core should increase the priority of the blocked task \( \tau_0 \) to the non-preemptive priority. In the original ERIKA kernel the checking of the ready queue and preemption of a task only happens when a task arrives (by ActivateTask()) or when a shared resource is released. In MSRP only the core that releases the shared resource might require a preemption, not the requesting core. In the new situation as illustrated in Figure 6.2b a preemption may need to take place on the core requiring the released resource. This additional checking for preemption and resumption is not yet implemented and needs to be added.

Note that this report addresses the implementation of the preemptable spin-locks that have a spin-lock priority within those used by \( CP \) and \( HP/MSRP \). The RAP used by MSRP could suffice with a single stack, every task would be completed and terminated before a lower priority task was resumed. In a RAP with priorities below \( HP \) it might be necessary to resume a task that is not at the head of the stack. To facilitate the resumption of a preempted spinning task we require at least two stacks.
6.2 Additional stacks

Figure 6.3 illustrates in the top what happens when a task is preempted and put on the stack (in HP). In the middle of Figure 6.3 is the (CP) scenario illustrated where a lower priority task should preempt a higher priority task because its resource is released. A stack has two principal operations: push, which adds an element to the collection, and pop, which removes the most recently added element that was not yet removed. To retrieve an item halfway the stack we need to remove the items on top of it. After an item is removed from the stack it should either be used by the active task or the data is lost. Thus the middle picture is not a feasible solution. Instead we require at least two stacks (we later solved this by providing each task with a separate stack).

The bottom time-line of Figure 6.3 illustrates the operation of the stack when it is split into two separate stacks. Task \( \tau_0 \) starts executing. During the execution of task \( \tau_0 \) the task adds data to stack 0. (the registry has only 30 entries anytime data or instructions are used by the processor and they do not fit in the registries, registries are freed by storing items on the stack.) The task \( \tau_0 \) request access to the resource while it is in use and starts spinning. When a task \( \tau_2 \) arrives with a higher priority than the system ceiling i.e., spinning priority, a context switch takes place preempting the spinning task \( \tau_0 \) and starting the task \( \tau_2 \). Tasks arriving after the premption of the spinning task and before resuming it add their items to stack 1. When the resource becomes available and task \( \tau_0 \) needs to be resumed its possible to do so. The data of \( \tau_0 \) is accessible from the top of stack 0. We only need 2 stacks for the flexible spin-lock model in theory, however in this project we used a separate stack for each task.

---

2We only need 2 stacks for the flexible spin-lock model in theory, however in this project we used a separate stack for each task.
Chapter 7

Flexible priority tool

To make the new RAP we need to change the priority of the spinning task.

1. Derive $CP$ and $\hat{CP}$ and $HP$ from the OIL-file and present those to the user. A user has subsequently to select the desired spin-priority from the range $[CP, HP]$. (Chapter 7)
2. Make the user-selected spin-priority available in the implementation. (Chapter 7)
3. Apply the priority in the resource access protocol (Chapter 9)

Chapter overview

Requirements p.58 The requirements regarding the flexible priority tool consist of: reading the OIL file, processing the OIL file, interaction with the user and writing the spin-priority to the output file spin_priority.h.

Specification p.58 An OIL file is provided to the flexible priority tool. The tool derives the $CP$, $\hat{CP}$ and $HP$ priorities per task and shows those to the user. The user can specify a flexible spin-lock priority in the range $[CP, HP]$. The flexible spin priorities are written to an output file such that the kernel layer can access them.

Design p. 59 The data in the OIL file consists of strings, thus tasks and cores have a name instead of an ID. The data is transformed to information that is easy to process. Several mappings are created of the information. The user provided priorities are checked, to make sure they are within bounds. The priorities are stored as data-structures in a .h file in the project folder.

Check invariants p.61 The tool should calculate the $CP$ and $\hat{CP}$ priorities. We check whether the steps taken in the code match with the definitions of $CP$ and $\hat{CP}$ provided in [3].

---

1Places to store the spin-priorities: 1. In the kernel folder or in the project folder and 2. in existing file or new file. Since the generated spin-priorities are specific for a certain task set its natural to store those in the project folder. By appending to a existing file the tool should take care that it will not append the priorities twice. Either the tool should prevent this from happening/remove previous entry (additional complexity) or the user should not try to use the tool for the same project twice (inconvenient). Additional problem is the RT-druid generates the files eecfg.c, eecfg.h and common.c, and does so each time the project is build (overwriting the spin-priorities). We choose to store spin-priorities in the file spin_priorities.h in the same folder as the input OIL file (in the correct project folder).
CHAPTER 7. FLEXIBLE PRIORITY TOOL

Introduction

The spin priority of a task is dependent on the protocol that is used, e.g. $CP$ or $\tilde{CP}$. Since these priorities do not yet exist in the MSRP ($HP$) implementation the priorities need to be derived. The spin priority $\rho_{\text{spin}}$ of tasks on core $P_k$ is a function of: the tasks, the tasks priority, shared resources and cores. This information can be derived from the OIL file by the RT-Druid tool of ERIKA as is illustrated in Figure 7.3a.

Figure 7.1: Specification of the priority of spinning task

Figure 7.1 illustrates when the priority of a task is changed to the spin priority.

The RT-druid writes the dispatch priority to the Eecfg.c (Figure 7.3b) file which is part of the kernel-layer. From the ready priority alone it is not possible to derive the $CP$ or $\tilde{CP}$ spin priority. The ready priority is the priority of a task the moment the task arrives (gets activated).

Figure 7.2: A tool is added parallel to RT-Druid to extract the $CP$ and $\tilde{CP}$ priorities from the OIL file

Figure 7.2 illustrates how to make the spin priorities available in the kernel-layer. The information about (tasks, priorities, shared resources and cores) is present in the OIL file. We use a tool that reads the CONFIG.oil file derives the $CP$, $\tilde{CP}$ and $HP$ spin priorities and lets the user choose a priority from the range $[CP, HP]$. The spin priorities are written into the Eecfg.c kernel file.
In the OIL-file the tasks and resources are defined. Each task gets a priority by the user. Also a mapping
between tasks and resources is given. Listing 7.1 shows how a task and a shared resource is declared.

```c
TASK task0 {
    CPU_ID = "cpu0";
    APP_SRC = "task0.c";
    PRIORITY = 1;
    ACTIVATION = 1;
    SCHEDULE = FULL;
    AUTOSTART = FALSE;
    STACK = SHARED;
    RESOURCE = mutex0;
};

RESOURCE mutex0 {
    RESOURCEPROPERTY = STANDARD { APP_SRC = "resourcedata.c"; }
};
```

Listing 7.1: Part of an OIL-file

RT-Druid reads the OIL-file and generates C-code. The initial data-structures are filled with the
information about the cores, tasks, resources and priorities. As an example we generate the data-
structures for a situation with 2 cores \( P_0 \) & \( P_1 \) and resources \( R_0 \) & \( R_1 \). Where \( P_0 \) has only one task \( \tau_0 \) while Core \( P_1 \) has five task:

- \( \tau_1 \) with priority 0 and uses resource \( R_0 \).
- \( \tau_2 \) with priority 1 and uses resource \( R_0 \).
- \( \tau_3 \) with priority 2 and uses resource \( R_1 \).
- \( \tau_4 \) with priority 3 and uses no resource.
- \( \tau_5 \) with priority 4 and uses no resource.

```c
/* ready priority core 0 (Eecfg.c in directory of core0)*/
const EE_TYPEPRIO EE_th_ready_prio[EE_MAX_TASK] = {
    0x1 /* thread task0 */
};

/* ready priority core 1 (Eecfg.c in directory of core1)*/
const EE_TYPEPRIO EE_th_ready_prio[EE_MAX_TASK] = {
    0x1, /* thread task1 */
    0x2, /* thread task2 */
    0x4, /* thread task3 */
    0x8, /* thread task4 */
    0x10 /* thread task5 */
};
```

Listing 7.2: The generated C code by RT-Druid for core \( P_1 \)
CHAPTER 7. FLEXIBLE PRIORITY TOOL

7.1 Requirements

1. Reading input file
   (a) Locate the oil file on the system
   (b) Read the core, task, resource and priority information from the OIL file.

2. Process input file
   (a) Determine the mapping between tasks and cores.
   (b) Determine the mapping between tasks and resources.
   (c) Determine the mapping between tasks and priorities.
   (d) Determine per resource whether it is local or global
   (e) Determine per core the maximum resource ceiling among local resources.
   (f) Determine per core the maximum resource ceiling among resources.
   (g) Determine the $CP$, $\hat{CP}$ and $HP$ priority for each core.

3. User interaction
   (a) Show the $CP$, $\hat{CP}$ and $HP$ priority for each core via GUI.
   (b) Ask and read user input to choose the priority for each core.
   (c) The provided priority should be in the range $[CP-HP]$.
   (d) The provided priority should be in the range $[0-7]$.

4. Process user interaction
   (a) Transform priority $\rho^{dec} = i$ to $\rho^{bin} = 2^i$ (ERIKA format)
   (b) Write the flexible spin-lock priority for each core to the output file.

7.2 Specification

Figure 7.3a illustrates the input OIL file that contains the information required to determine the $CP$ and $CP$ priorities. The tool should read the OIL file, determines the $CP$, $\hat{CP}$ and $HP$ priorities and requests the user to provide a priority per task out of the available range. Figure 7.3b illustrates how the output should be provided (in Figure the ready priority is given instead of spin-priority. Each core should have an array of $EE_{th\_spin\_prio}[]$ with a priority per task as a power of 2.

1 TASK task1 {
2   CPU_ID = "cpu1";
3   PRIORITY = 1;
4   RESOURCE = R1;
5   ....
6 }
7
8 TASK task2 {
9   CPU_ID = "cpu1";
10  PRIORITY = 6;
11  RESOURCE = R2;
12  ....
13 }
14
15 ....

(a) OIL file

1 const EE_TYPEPRIO EE_th_ready_prio[EE_MAX_TASK] = {
2   0x1, /* thread task1 */
3   0x10, /* thread task2 */
4   0x2, /* thread task3 */
5   0x4, /* thread task4 */
6   0x8 /* thread task5 */
7 };

(b) Eecfg.c file

Figure 7.3: Part of the input and output file of RT-Druid

Definitions:

1) The $CP$ spin-lock priority for a core is the highest priority among tasks that use a global resource on the core.
2) The $\hat{CP}$ spin-lock priority for a core is the highest priority among tasks that use a (global or local) resource on the core.
3) The tool should show the priorities to the user via the terminal and allow to provide a priority ranging from $CP$ to $HP$.

We will check the definitions as if they were invariants.

Thursday 30th June, 2016 18:15
7.3 Design
The design consists of the following steps:
1. Read the information about "cores, resources, and priorities" grouped per task from the oil file.
2. Cores, tasks, and resources all are represented by strings, to be able to work with them we store them by numbers, e.g. "string_P0" becomes 0 (in Figure 7.5 represented with $P_0$).
3. Create a mapping of tasks to cores.
4. Create a mapping of tasks to resources.
5. Create a mapping of tasks to priorities.
6. Reduce the number of priorities, e.g. $\rho = 0,3,5$ becomes $0,1,2^3$.
   (the priority is not yet translated to powers of 2).
7. Determine per resources whether it is global or local.
8. Determine per task the set of global resources.
9. Determine the $CP$ priorities, per core $P_k$: The maximum priority among tasks that use a global resource on core $P_k$.
10. Determine the $\hat{CP}$ priorities, per core $P_k$: The maximum priority among tasks that use any resource on core $P_k$.
11. Determine the $HP$ priorities, per core $P_k$: The maximum priority among tasks on core $P_k$.
12. Provide the user with a range of priorities to choose from between $CP$ and $HP$ per core.
13. Write the spin-priorities per core in spin_priorities.h file in the project folder.

7.3.1 The interface
Figure 7.4 illustrates what the tool looks like.

---

3 This is also done by the RT-Druid to come up with the dispatch priorities.
4 We choose to make the spin priority of a core without global resource access to be equal to 0x0. The actual number would not matter in this case. The core that has no global resource access will not spin on a resource and thus never use its spin-priority.
CHAPTER 7. FLEXIBLE PRIORITY TOOL

Figure 7.5: Behavioral diagram and data-structures

Thursday 30th June, 2016 18:15 60 Performance of resource access protocols
7.4 Check Invariants

1) The CP spin-lock priority for a core is the highest priority among tasks that use a global resource on the core.

Figure 7.5 illustrates how the CP.priority[P_k] is calculated.

Task2Priorities[τ_i], is the priority of task τ_i, e.g. ρ_i.
Tasks2Resources[x][τ_i], is the set of resources accessed by jobs of task τ_i, e.g. RS_i.
Tasks2Cores[τ_i], is the core to which task τ_i is mapped, e.g. τ_i ∈ TP_k

Resource_scope[R_q], denotes 1 if the resource is global else 0, e.g. R_q ∈ RS^G

Resource_scope[R_q] = If ((Tasks2Resources[x][τ_i] == R_q) && (Tasks2Resources[y][τ_i] == R_q) && Tasks2Cores[τ_i] != Tasks2Cores[τ_j])

R_q ∈ RS^G ⇐⇒ (R_q ∈ RS_i ∧ R_q ∈ RS_j ∧ (τ_i ∈ TP_k ∧ τ_j ∈ TP_l ∧ P_k ≠ P_l))  

(7.1)

Tasks2GlobalResources[x][τ_i], is the set of global resources accessed by jobs of task τ_i, e.g. RS^G_i.

Tasks2GlobalResources[x][τ_i] = If(Resource_scope[Tasks2Resources[x][τ_i]])

R_q ∈ RS^G_i ⇐⇒ (R_q ∈ RS_i ∧ R_q ∈ RS^G ∧ τ_i ∈ TP_k ∧ R_q ∈ RS^G ∧ R_q ∈ RS_i)

(7.2)

CP.priority[P_k], is the maximum priority among tasks that uses a global resource on core P_k, e.g. rc^G_P_k.

CP.priority[P_k] = max{if(Tasks2GlobalResources[0][τ_i] != empty && Tasks2Cores[τ_i] == P_k)}

Task2Priorities[τ_i] }

rc^G_P_k = max{ρ_i | τ_i ∈ TP_k ∧ RS^G_i ≠ ∅}.

(7.3)

Note that the derived proposition denoting the outcome of the function is the same as the Definition 3.1 given by Sara et al. in [3].

2) The CP hat priority of a core is the highest priority among tasks that uses a (global or local) resource on the core.

CP_hat.priority[P_k] = max{ If(Tasks2Resources[0][τ_i] != empty && Tasks2Cores[τ_i] == P_k)}

Task2Priorities[τ_i] }

rc^L_P_k = max{ρ_i | τ_i ∈ TP_k ∧ RS^L_i ≠ ∅} = max{ρ_i | τ_i ∈ TP_k ∧ RS^G_i ∪ RS^L_i ≠ ∅}.

(7.4)

3) The tool should show the priorities to the user via the terminal and allow to provide a priority ranging from CP to HP.

Trivially satisfied. Figure 7.4 gives an illustration of the interface that allows to provide a priority. A check is performed whether the priority is out of bounds. In the figure is a priority of 5 provided to task of which the range contains only 1, therefore the priority is requested again.

```c
#if EE_CURRENTCPU == 0 //pre processor makes sure data is only stored in core P0
int spin_priority =0;
#endif

#if EE_CURRENTCPU == 1 //stored in P1
int spin_priority =4;
#endif
```

Listing 7.3: The output code
Chapter 8

Spinning on a local spin_lock

To make the new RAP we need to notify the release of a resource to a waiting core by means of an interrupt.

Chapter overview

Requirements p.64 The resource access by spinning on a local value (bit) instead of a global spin_lock contains requirements regarding mutual exclusive access. It should not be possible for multiple tasks to access a resource at the same time. There are also constrains on the access of the resource queue.

Specification p.65 The specification contains some invariants, which are rules about the system that should not be violated. The purpose is to keep those rules in mind during the design process and check the invariants after the design is complete.

Design p.66 The design of the resource access by spinning on a local instead of global spin_lock contains changes to the resource access and the remote notification functions. The spinning task checks a local variable that gets updated by receiving an interrupt from a releasing core.

Check invariants p.74 After the design is finished the invariants are checked. An informal reasoning is given why the properties hold.

Measurements p.77 The measurements section contains the execution times in cycles of parts of the code that are used during the resource access. Some parts of the code might take a manyfold of the measured time in worst-case scenarios. For the RAP analysis the time overhead of the implementation is measured.
Introduction

A step towards the flexible spin-lock model is implementing the HP/MSRP protocol by spinning on a local instead of global spin_lock. Resource access protocols regulate the access to resources. Tasks should access a resource mutual exclusive. Anytime a resource is available an access requesting task can lock the resource. In case the resource is unavailable the requesting task can either spin (active waiting) or suspend (let other tasks execute during the time the resource is unavailable). By setting the tasks spinning priority (in ERIKA’s case the system ceiling) it can be controlled whether a task spins or suspends.

In case the requesting task spins (either preemptive or non-preemptive), the task continuously checks the availability of the resource. When the resource becomes available the task can directly lock the the requested resource. In case the requesting task suspends and lets another task execute until the resource becomes available the suspended task needs its execution to resume the moment the resource gets released. Since the suspended task is not actively checking the availability of the resource the suspended task should get notified. The task releasing the resource should notify a requesting task on a remote core.

The input to a processor can be categorized into polling and interrupt based. The task requesting access might be suspended and will not periodically check the spin_lock instead we use interrupt based notification. A releasing task sends an interrupt to the core containing the requesting task. On the receiving core an interrupt will interrupt the executing task, the core will start its interrupt handler and register the release of the resource.

Raising an interrupt routine on a remote core is possible in our hardware platform by connecting I/O pins in the hardware description. The ERIKA OS already provides a mechanism of sending and receiving an RN as is explained in section: 5.7. The functions rn_send() (for sending), rn_handler() and rn_execute() (for receiving) are available for remote notifications. A potential problem by remote invocation of interrupts is sending of interrupts while a receiving core already has its interrupt flag set. It is possible for the receiving core to miss additional interrupts. By using those functions provided by ERIKA we already have a proven solution for sending an RN. The only adjustment we need to make is to allow for an additional type of RN to be send denoting the release of a shared resource.

Figure 8.1a illustrates the behaviour of the RAP based on spinning on a global spin_lock. Figure 8.1b illustrates interrupt-based notification of releasing the shared resource. In both cases the resource access is in order of arrival (of access requests), which means that we need to keep track of that order by means of a queue. When a task requests access to the shared resource the request is stored into the resource queue. 1) In the polling scenario, the resource is locked by \( \tau_0 \) (not depicted). The requesting task \( \tau_1 \) continuously spins on a spin_lock in global memory. The releasing task \( \tau_0 \) unlocks the global spin_lock in the resource queue as illustrated in Figure 8.1a. The requesting task \( \tau_1 \) notices the release and acquires access (\( C_1 \) spin_lock is already locked). 2) In the interrupts scenario, the spinning task \( \tau_1 \) checks its local spin_lock continuously which is initially locked (not depicted). The releasing task \( \tau_0 \) sends an interrupt to core \( P_i \) containing the requesting task to indicate the release. On arrival of the interrupt the interrupt handler (IH) of the receiving core \( P_i \) unlocks the local spin_lock. The spinning task on the receiving core checks the local spin_lock which indicates whether the resource is available. Properties to consider:

1. Spinning on a local instead of a global spin_lock.
2. Releasing task must know which core to inform.

(a) Global variable  
(b) Local spin_lock

Figure 8.1: Spinning on a global variable vs a local spin_lock
CHAPTER 8. SPINNING ON A LOCAL SPIN_LOCK

8.1 Requirements

1. Initial condition
   (a) At start-up the resource queue is empty (does not contain any task).
   (b) At start-up no task has access to a resource.
   (c) At start-up the local spin_lock in each core denotes unlocked (0).

2. Requesting Access
   (a) When a task requests access to a resource the task is put at the tail of the resource queue.
   (b) If two tasks request access to the resource at the same clock tick both should end up in the resource queue.
   (c) A task that requests access to a resource that is in use by another task locks (1) its local spin_lock.

3. In resource queue
   (a) The resource queue is FIFO ordered (in order of arrival of access requests).
   (b) A spinning task checks a local variable not a global variable.
   (c) A task that does not require access to a shared resource is not in the resource queue.
   (d) A task cannot occur twice in the resource queue.
   (e) At most one task per core can occur in the resource queue of each resource.
   (f) A task can occur in at most one resource queue at any time (no nested resource access).

4. During access
   (a) No two tasks can have access to the shared resource at the same time.
   (b) When a task has locked the resource no other task can access the resource.
   (c) The local spin_lock of the accessing core is unlocked (0).

5. Releasing Access
   (a) A task that releases the shared resource is removed from the resource queue.
   (b) A releasing task must know which task to notify of its release.
   (c) The notification should take place via an interrupt.
   (d) A task that releases a resource sends an interrupt to the core containing the task that is next in the resource queue.
   (e) A core receiving an interrupt denoting the release of the resource its task is spinning on, sets its local spin_lock to denote unlocked (0).

Requirements that already hold for the original implementation by spinning on a local spin_lock are: 1a, 1b, 2a, 2b, 3a, 3c-f, 4a-c and 5a.
Additional requirement that are required for the interrupt-based implementation are: 1c, 2c, 3b and 5b-e.
8.2 Specification

Any task requesting access to a resource checks the resource queue to determine whether there is a preceding task.

- If the queue is empty such as on start-up, a task $\tau_i$ that requests access to the resource adds its task ID at the tail of the resource queue. In a queue of one item the tail of the queue is also the head of the queue. No resource is locked thus the requesting task $\tau_i$ sets its local spin_lock to unlocked (0). Task $\tau_i$ can access the resource. When the task finishes its access it removes its task ID from the resource queue and notifies the task next in the resource queue in case there exists a waiting task.
- If a task $\tau_j$ would request access while the resource queue is not empty, the requesting task is added to the tail of the resource queue. The task $\tau_j$ sets its local spin_lock to locked (1) and waits until it is notified by the releasing task one place ahead in the resource queue.

8.2.1 Invariants initial condition
1. The number of received interrupts regarding R on a core is identical to the number of processed interrupts regarding R on a core.
2. The resource queue is empty (no task ID occurs in any resource queue).
3. All local spin_lock’s are unlocked (0).

8.2.2 Invariants during runtime
4. A task can occur in at most one resource queue (we ignore nested resource access).
5. At most one task per core can occur in any resource queue.
6. At most one task has access to a resource $R_q$.
7. If the local spin_lock of core $P_k$ is locked (1), then $\tau_i | \tau_i \in T_{P_k}$ has no access to the resource.
8. If the local spin_lock of core $P_k$ is unlocked (0) and $\tau_i | \tau_i \in T_{P_k}$ occurs at the head of the resource queue of $R_q$, then $\tau_i$ has access to $R_q$.
9. After GetResource($R_q$) and before ReleaseResource($R_q$) a task has access to $R_q$.
10. The number of received interrupts regarding $R_q$ on a core is at most 1 larger than the number of processed interrupts regarding $R_q$ on a core.
11. If the local spin_lock is locked (1), a task on the core occurs in a resource queue.
12. If the local spin_lock of core $P_k$ is unlocked (0), either no task $\tau_i | \tau_i \in T_{P_k}$ occurs in any resource queue or a task has access to a resource $R_q$.
13. If the number of received interrupts regarding resource access on a core $P_k$ larger is than the number of processed interrupts, then the local spin_lock of $P_k$ is locked (1), $\tau_i | \tau_i \in T_{P_k}$ occurs at the head of the resource queue of $R_q$ (but $\tau_i$ has no access to $R_q$ yet).

---

1 The status of spin_lock local to core $P_k$ is unlocked (0) if the resource is available for access by $P_k$. The spin_lock is locked (1) if the resource is locked by another core $P_j$ where $j \neq i$. 

Performance of resource access protocols 65 Thursday 30th June, 2016 18:15
8.3 Design

Here we present the design for implementing the MSRP/HP protocol by spinning on a local instead of a global spin_lock. We implement the new RAP in three steps of which changing from global to local spinning is the first step. The design consist of a new way of handling the resource queue and the changes that are needed to send and receive the interrupts.

We need to implement changes at the core that requests and the core that releases the access to a resource as follows:
1. A queue in shared memory containing all the waiting tasks, instead of only toggle bits.
2. Make an interrupt routine to resume the spinning task and lock the resource.
   - When the task requiring the resource is still the active task.
   - When the task requiring the resource has been preempted by another task. (For now we assume that this situation will not occur, it will be described in the Chapter 9)

Recap of the properties we require in the queues data-structure are as follows:
- The order of arrival (of access requests) must be preserved.
- Adding to and removing from the queue must be atomic.
- The used memory should be small.

To keep track of the tasks that are waiting for the release of a resource we store the taskID’s in the resource queue. The taskID is unique to a task, there exist no two tasks with the same ID on any core in the system. It is not possible that core P_0 and P_1 both have a task τ_2. The ERIKA system already keeps track of the mapping of tasks to cores which is used in sending the RN function rn_send(). By keeping track of a requesting task also the information of the requesting core is available. Some data-structures used in the RN functions are indexed by means of task ID’s. Any RN message send is related to a task. Therefore its convenient to store requesting tasks instead of cores.

Actions that should occur regarding a task τ_i requesting access:
1. Task τ_i is stored on the place the preceding task will look at the moment of releasing the resource.
2. Task τ_i updates the tail of the resource queue to the entry in the resource queue τ_i will look at the moment of releasing the resource to find out which succeeding task it needs to inform.
3. Task τ_i sets its local spin_lock to locked (1) in case the core is not at the head of the queue otherwise τ_i sets the local spin_lock to unlocked (0).
4. Task τ_i should when it receives a remote notification indicating the release of the shared resource: set its local spin_lock to unlocked (0).

Actions that should occur regarding a task τ_i releasing access (explained in section 8.3.2)
1. Task τ_i should send an interrupt to the next task in the resource queue, to notify the release of the resource.
2. Task τ_i should remove itself from the head of the resource queue.
8.3.1 Changes to spin_in()

A task requests access to resource \( R_q \) by the function call GetResource(\( R_q \)). GetResource(\( R_q \)) increases the system ceiling to a non-preemptive priority and calls spin_in(\( R_q \)) to spin on a global resource. To go from spinning on a global to a local spin_lock we need to change the spin_in() function.

We want a waiting core to be spinning on a local spin_lock instead of global spin_lock. The waiting core will be interrupted from spinning to update the local data by an interrupt routine. A core that releases a lock on the shared resource should notify the waiting core about its release. This implies that the releasing core should know which core is next in the resource queue.

The access to the resource queue must be mutual exclusive. If two tasks would write their task ID at the same instance on the same place in the resource queue, one of those entries gets overwritten. By only allowing one task to access the resource queue of \( R_q \) at the same time any task can complete its request before another task can interfere. Tasks requesting access to different resources can execute their requests simultaneously.

The access to the resource queue does not require the interrupts to be disabled. Let in this paragraph \( \tau_i \) denote a task executing an critical section and \( \tau_j \) a task that gets activated by an interrupt any time during the execution of \( \text{spin_in()} \) by \( \tau_i \). Before a task \( \tau_i \) tries to enter the resource queue the caller of \( \text{spin_in()} \) e.g. GetResource(), raises the system ceiling to a non-preemptive priority. The task starts with non-preemptive spinning until it receives access to the resource/critical section. A interrupt as a result of a timer would halt the execution of \( \tau_i \) and start handling the interrupt. The function registered as interrupt handler to specific interrupt, in this case the function ActivateTask(\( \tau_j \)) is executed. The function ActivateTask(\( \tau_j \)) would put \( \tau_j \) in the ready queue. The priority \( \rho_j \) would be lower than the system ceiling and the core would return to the executing task \( \tau_i \) registering its request resource. The priority of tasks, resources, system ceilings is of no concern in the function \( \text{spin_in()} \) that is all taken care of by the caller GetResource().

In the original \( \text{spin_in()} \) function the releasing core notifies the release by toggling a global spin_lock. We can reuse the core’s own global toggle field to store the taskID of the requesting task. Anytime a core releases a resource it should check the place in the resource queue entry with an index related to the task releasing the resource not the index of the task requesting the resource to find the succeeding task in the resource queue. Either there exists a succeeding task and the releasing core sends an interrupt to that core or there is no waiting task and the core can continue its operation.\(^2\)

\[
\text{spin_in}(R_q) \{ \\
\quad /\text{let } \tau_i \text{ denote the task calling } \text{spin_in}(R_q) \\
\quad \text{Acquire_hardware_lock} \\
\quad \text{Check if the resource } R_q \text{ is locked and we need to poll} \\
\quad \quad \text{Set its local spin_lock to locked (1)} \\
\quad \quad \text{Store } \tau_i \text{ in the resource queue where the task one place ahead in the resource queue will check} \\
\quad \quad \text{Let the tail of the resource queue point to the place where } \tau_i \text{ will look upon releasing the resource} \\
\quad \quad \text{Put } \tau_i \text{ at the tail of the queue} \\
\quad \text{Release_hardware_lock} \\
\quad \text{While (local spin_lock is locked (1));} \\
\}
\]

Listing 8.1: Pseudo code of \( \text{spin_in()} \), spinning on a local spin_lock

---

\(^2\)The original version of \( \text{spin_in()} \) is explained in 5.6.4.

\(^3\) Thus we store task ID’s in the matrix called \( \text{spin_value}[R_q][P_k] \) that originally contained only a toggle bit.
8.3.2 Changes to spin_out()

A task releases resource $R_q$ by the function call $\text{ReleaseResource}(R_q)$. $\text{ReleaseResource}(R_q)$ returns the system ceiling to the priority the system ceiling had before the resource access request and calls $\text{spin\_out}(R_q)^4$ to unlock the spin_lock on a global resource. A releasing task should send an interrupt to inform a requesting task of its release.

We make some design decisions regarding the resource queue used in spin_in() and spin_out().

1. The value 0xa0 denotes "an empty entry, not using the resource".
2. The task its taskID denotes "the task itself was the last requester, no need to inform other tasks".
3. A task $\tau_j$ checking the resource queue observing $\tau_i$ with $i \neq j$ denotes there exists a waiting task.

We identified the following actions that take place when a resource is released:

1. Acquire mutual exclusive access to the resource queue of $R_q$.
2. Look in the resource queue which task is awaiting the resource release and should be informed.
3. Remove its entry in the resource queue (either its own task ID or the task ID it needs to notify).
4. Release mutual exclusive access to the resource queue of $R_q$.
5. If the waiting task ID differs from the task ID releasing the resource{ send an RN to the waiting core}.

The original implementation of spin_out() by ERIKA only needed to flip a bit in shared memory to signal the release of a resource and could do so without mutex protection. We need to store and read more information than just the status of the resource in the resource queue. By storing and reading the task ID in the ready queue it is possible to inform a waiting core when the resource becomes available.

The reason for removing the entry from the resource queue after the resource is released is the possibility to detect the empty queue. Any time a task enters the resource queue its taskID is stored and the tail pointer is moved. When the last task that requested access to the resource releases the resource the tail still points to the task entry in the resource queue, but the entry itself becomes 0xa0. Any task that checks the resource will determine that the queue is empty and can directly lock the resource.

A task requesting access to a resource at a time $t_0$ checks whether there is a preceding task and at time $t_1$ stores its taskID (used to get notified). If the access to the resource queue by the releasing task is unprotected by mutex it is possible for a releasing task to reset the entry in the resource queue between $t_0$ and $t_1$. To prevent this from happening the resource queue is protected by mutex.

The required changes are updating the queue and sending an interrupt to the waiting core. Listing 8.2 illustrates the pseudo code used to implement the interrupt-based spin_out().

```
spin_out(R_q) {
    //let $\tau_i$ denote the task calling spin_out(R_q)
    Acquire_hardware_lock
    Determine the waiting Task
    Remove the task from the resource queue /*either the waiting task if any, else $\tau_i$*/
    Release_hardware_lock
    If the waiting task differs from $\tau_i$ {
        Send a remote notification
    }
}
```

Listing 8.2: Pseudo code of spin_out(), spinning on a local spin_lock

The sending of the interrupt in Listing 8.2, line 9, is outside the hardware lock. Inside the hardware lock (line 2-5) the succeeding (waiting) task is identified. Inside the lock two actions happen: 1) the succeeding access requesting task is read if any and 2) the task resets its entry to 0xa0. After these two actions the lock is released, and there is either a core waiting, or not. In the case there is a core waiting the interrupt will be send (8.2 line 9) to the waiting core regardless of what happens inside the resource queue. In the case there was no waiting core, no interrupt is send. The next core that wants to access the resource reads 0xa0 denoting that the resource is available.

---

The original version of spin_out() is explained in 5.6.4.
8.3.3 Changes to \texttt{rn\_execute}

The task that releases a shared resource sends an interrupt to inform the task next in the resource queue of its release. The core receiving the RN about the release of the resource starts executing the interrupt routine. The RN handler is registered in the interrupt table and reads the RN message. The \texttt{rn\_execute()}\footnote{The original version of \texttt{rn\_execute()} is explained in 5.7.3.} is called to perform the action contained in the RN message. In the new implementation by spinning on a local bit an additional type of RN is send denoting the release of a resource.

We can add an RN in addition to the already existing: remote task, remote function, binding and unbinding. The new RN will set the local spin\_lock to unlocked(0) to notify the release of a resource to the task which is spinning. Listing 8.3 presents the pseudo code of the interrupt handler that is added.

\begin{verbatim}
\begin{lstlisting}[language=C]
void spin\_in(EE\_TYPESPIN m){
    /* EE\_hal\_spin\_value[resource][cpu] initialized with 0xa0
    /* EE\_hal\_spin\_status[resource] initialized with &EE\_hal\_spin\_value[m][0]
    /* spin\_lock initialized with 0
    EE\_altera\_mutex\_spin\_in();
    if(*(int *)EE\_hal\_spin\_status[m] != 0xa0){
        spin\_lock=1; /* The resource is locked and we need to spin
        *(int *)EE\_hal\_spin\_status[m]=GlobalTaskID[EE\_stkfirst]; /*Store taskID in preceding core’s queue
    }
    EE\_hal\_spin\_value[m][EE\_CURRENTCPU]=GlobalTaskID[EE\_stkfirst]; /*Store taskID in current core’s queue
    EE\_hal\_spin\_status[m]=(int) &EE\_hal\_spin\_value[m][EE\_CURRENTCPU]; /*Update the tail of the queue
    EE\_altera\_mutex\_spin\_out();
    while (spin\_lock == 1);}
}
void spin\_out(EE\_TYPESPIN m){
    EE\_altera\_mutex\_spin\_in();
    int task2notify=EE\_hal\_spin\_value[m][EE\_CURRENTCPU]; /*Determine the waiting task
    EE\_hal\_spin\_value[m][EE\_CURRENTCPU]=0xa0; /*Remove taskID from head of queue
    EE\_altera\_mutex\_spin\_out();
    if(task2notify!=GlobalTaskID[EE\_stkfirst]){ /*If there exists a waiting task, send an interrupt
        register EE\_TYPERN\_PARAM par;
        par.pending = 1;
        EE\_rn\_send(task2notify, RN\_ReleaseResource, par );
    }
}
\end{lstlisting}
\end{verbatim}

Listing 8.4: C-code of \texttt{spin\_in()} and \texttt{spin\_out()}, spinning on a local spin\_lock
8.3.4 Example spin_in() and spin_out(); initialization & request access by \( \tau_{11} \) on \( P_1 \)

Figure 8.2a illustrates the content of the data-structures used by spin_in() and spin_out() at the moment of initialization. The data-structures are:

1. **spin_status\([R_q]\)**, global data that points to the tail of the resource queue. Any task on core \( P_k \) requesting access to \( R_q \) puts its task ID at the place in the resource queue pointed by \( \text{spin\_status}[R_q] \) and updates the tail pointer to the place in the ready queue where \( P_k \) checks upon release of \( R_q \).

2. **spin_value\([R_q]\)[\(P_k\)]**, global data where \( P_k \) will look for a waiting task to signal the release of \( R_q \).

3. **task2notify**, local data that contains the taskID read from \( \text{spin\_value}[R_q][P_k] \) of the task that needs to be notified when core \( P_k \) releases \( R_q \).

4. **spin_lock** local data stores the spin-lock value. A waiting core sets its spin_lock to locked (1) and spins on the local spin_lock. Any releasing core sends an interrupt to remotely update the spin_lock to unlocked (0).

The initial content of \( \text{spin\_value}[R_q][P_k] \) is 0xa0, which denotes that \( P_k \) does not use nor requests \( R_q \). The \( \text{spin\_status}[R_q] \) is a pointer to the tail of the resource queue. The arrows in the Figure 8.2a depict the pointers. Initially \( \text{spin\_status}[R_q] \) (tail of resource queue) points for any resource \( R_q \) to \( \text{spin\_value}[R_q][P_0] \) (the resource queue entry of core \( P_0 \)). It does not make any difference to which core the tail of the queue initially points as long as it denotes empty (0xa0). The reason for this is as follows: The first task that requests access will check the address pointed by the \( \text{spin\_status} \) (TailQueue). Since the value in the address pointed by the \( \text{spin\_status} \) (TailQueue) is \( \text{spin\_value}[R_q][P_0] \) (which is initialized with 0xa0), the requesting core will know that core \( P_0 \) is not using the resource. Thus the requesting core can access the resource. The initial value of the local spin_lock for each core is unlocked (0) (no core is spinning yet). The data-structure task2notify is local to the function spin_out() and is initially not available. When data is not available it is represented with a "-" sign in Figure 8.2a.

![Figure 8.2](image)

**Figure 8.2:** The data-structures used by spin_in() and spin_out()

Figure 8.2b illustrates the scenario that task \( \tau_{11} \) on core \( P_1 \) requests access to resource \( R_0 \). Task \( \tau_{11} \) acquires mutual exclusive access to the resource queue and is able to read and write from the global data \( \text{spin\_status} \) and \( \text{spin\_value} \). Task \( \tau_{11} \) reads the address of the tail of the queue in the array "spin\_status" which is 4000 and points to the resource queue entry of core \( P_0 \). Task \( \tau_{11} \) then reads whether core \( P_0 \) is currently using the resource by checking the value at address 4000 which is 0xa0. Task \( \tau_{11} \) identifies that the resource is currently not in use by the tail of the queue and the resource queue is empty.

Since the resource queue is empty, core \( P_1 \) does not need to lock its local spin_lock thus there is no need for \( \tau_{11} \) to store its taskID for the preceding core \( P_0 \). Task \( \tau_{11} \) puts itself at the tail of the resource queue by changing the \( \text{spin\_status} \) to 4004 which points to the core’s own \( \text{spin\_value}[R_0][P_1] \). The address of EE_hal\_spin\_value\([R_0][P_1]\) (4004) is the place where core \( P_1 \) will check for any task it should notify (task next in resource queue). Task \( \tau_{11} \) can let the next core know that it currently uses the resource by placing the taskID of \( \tau_{11} \) in \( \text{spin\_value}[R_0][P_1] \). Task \( \tau_{11} \) releases the hardware lock.
The local spin\_lock\(^6\) was not set (no preceding core) thus the spinning statement evaluates to false, the core exits the while statement and receives access to the resource. We now have a queue consisting of one core \(P_1\) which is at the head and has access to the resource.

8.3.5 Example: Task \(\tau_1\) on \(P_2\) requests access to \(R_0\) while \(\tau_1\) on \(P_1\) has access

Figure 8.3 illustrates what the content of the data-structures is after task \(\tau_1\) on core \(P_1\) still has access to resource \(R_0\) and task \(\tau_2\) on core \(P_2\) also requests access to the same resource. Task \(\tau_2\) acquires mutual exclusive access to the resource queue and is able to read and write from the global data spin\_status and spin\_value. Task \(\tau_2\) reads the address of the tail of the queue in the array spin\_status[\(R_0\)] which is 4004 and points to the field of core \(P_1\). Task \(\tau_2\) then reads whether core \(P_1\) is currently using the resource by checking the value at address 4004 which is 11 the taskID of \(\tau_1\) (not 0xa0, thus not empty). Task \(\tau_2\) determines that the resource is currently in use.

Since the resource queue is not empty, core \(P_2\) sets its local spin\_lock to locked (1). Task \(\tau_2\) stores its taskID in the field where the releasing core \(P_1\) will look to notify upon release of \(R_0\), which is EE\_hal\_spin\_value[\(R_0\)][\(P_1\)]. Task \(\tau_2\) puts itself at the tail of the resource queue by changing the spin\_status[\(R_0\)] to 4008 which points to the core’s own spin\_value[\(R_0\)][\(P_2\)]. The address of spin\_value[\(R_0\)][\(P_2\)] (4008) is the place where core \(P_2\) will check for any task it should notify (task next in resource queue). Task \(\tau_2\) lets the next task that will request access to \(R_0\) know that \(P_2\) is currently waiting (or has access to the resource) by placing the taskID of \(\tau_2\) in spin\_value[\(R_0\)][\(P_2\)]. Task \(\tau_2\) releases the hardware lock. The local spin\_lock of \(P_2\) is locked (1) (there exists a preceding core) thus the spinning statement evaluates to true, the core cannot exit the while statement and continues spinning. We now have a resource queue of task \(\tau_1\) at the head accessing the resource and \(\tau_2\) waiting for release of resource \(R_0\).

![Figure 8.3: Task \(\tau_1\) on \(P_2\) uses \(R_0\), task \(\tau_2\) on \(P_2\) waits for release of \(R_0\)](image)

8.3.6 Example: Task \(\tau_1\) on \(P_1\) releases access, \(\tau_2\) on \(P_2\) is in the resource queue

Figure 8.4 illustrates what happens when \(\tau_1\) on \(P_1\) releases the resource \(R_0\). Task \(\tau_1\) acquires the hardware lock. Task \(\tau_1\) is able to determine the succeeding core by looking at its own place in the spin\_value matrix, the task\_notify is task \(\tau_2\). task\_notify is local data and is used to send the RN. Task \(\tau_1\) removes itself from the queue by updating its spin\_value to 0xa0 to denote that the core \(P_1\) is not using nor waiting for the resource. Task \(\tau_1\) releases the hardware lock. Task \(\tau_1\) sends an RN to the waiting core \(P_2\) to signal the release of the resource \(R_0\). We now have a queue of \(P_2\) at the head that has no access yet.

Figure 8.4b illustrates the scenario that core \(P_2\) receives the RN that the resource is released. The IH changes the local spin\_lock from locked (1) to unlocked (0). The IH handles all RN and the core returns to the while(spin\_lock is locked) statement. Task \(\tau_2\) on \(P_2\) is now able to escape the while loop and acquires access to the resource \(R_0\). We now have a queue of task \(\tau_2\) on \(P_2\) at the head accessing the resource \(R_0\).

---

\(^6\)The local spin\_lock is initially unlocked (0), it gets locked (1) in Listing 8.1 line 4, unlocked (0) in \text{rn\_handler()} Listing 8.3 line 3 and checked in Listing 8.1 line 11. In the original implementation the global spin\_lock was part of the EE\_hal\_spin\_status, this is not the case anymore.
8.3.7 Example; Task $\tau_{12}$ on $P_2$ releases access to $R_0$, the queue becomes empty

Figure 8.5a illustrates the scenario where task $\tau_{12}$ on $P_2$ releases the resource $R_0$. Task $\tau_{12}$ acquires mutual exclusive access to the resource queue. Task $\tau_{12}$ is able to determine the succeeding core by looking at its own place in the spin value matrix, the task2notify is task $\tau_{12}$. Task $\tau_{12}$ removes itself from the queue by updating its spin value to 0xa0 to denote that the core $P_1$ is not using nor waiting for the resource. Task $\tau_{12}$ releases the hardware lock. Task $\tau_{12}$ is the task2notify itself, there is no need to send an RN. We now have a queue that is completely empty.

Figure 8.5b illustrates what the queue looks like when there are no tasks in the queue. The tail of the queue points to the last core that released access $P_2$, the content of spin_value[$R_0$][$P_2$] (=$0xa0$) represents empty thus the queue is empty.
Figure 8.6: Overview of locking and unlocking a resource

Main()
- StartOS()
- Activate_Task()
- While()

GetResource()
- Increase spin_count to non-preemptive
- Spin_in()

spin_in()
- Mutex_spin_in()
- Enter to resource queue
- True
- spin_locked
- False
- spin_locked

spin_out()
- Mutex_spin_out()
- While spin_locked

rn_execute()
- If RNI_release
- spin_locked

rn_handler()
- Disable interrupts
- spin_in_original()
- Switch the RNI buffer
- Enable interrupts

For pending request
- RNI_execute()

ReleaseResource()
- Get_Resource
- Use resource
- Release_Resource

Current task highest priority?
- Yes
- Context switch
- Enable interrupts

spin_out_original()
- Store RNI message

Interrupt results in execution of RNI_handler on another core
CHAPTER 8. SPINNING ON A LOCAL SPIN_LOCK

8.4 Check Invariants

The project is constructed in 3 steps. The first of those steps is changing from spinning on a global variable to spinning on a local spin_lock. Each step consists of describing the requirements, specification and design. In this section we validate the design by checking the invariants given in the specification 8.2.1 and 8.2.2.

8.4.1 Invariants initial condition

1) The number of received interrupts regarding R on a core is identical to the number of processed interrupts regarding R on a core.

The initialization phase consists of the function StartOS() as can be seen in Figure 8.6. Within the StartOS() function no GetResource() or ReleaseResource() requests are done. No interrupts concerning resources are send nor received. Thus the number of received and processed interrupts is zero.

2) The resource queue is empty (no task ID occurs in any resource queue).

The whole resource queue "EE_hal_spin_value" gets initialized with 0xa0 which is not the ID of a task nor of any core.

3) All local spin_lock’s are unlocked (0).

The local spin_lock gets initialized with a 0 which means unlocked.

8.4.2 Invariants during runtime

4) A task can occur in at most one resource queue (we ignore nested access).

Proof by contradiction. Assume a task occurs in two different resource queues. A task can only occur in any resource queue by entering the queue via spin_in() as illustrated in Figure 8.6 and Listing 8.4 line 8. The entry in the resource queue gets removed by function spin_out() as illustrated in Figure 8.6 and Listing 8.4 line 22. To occur in two resource queue’s a task needs to call spin_in() twice without a spin_out() in between. We derive a contradiction since we do not allow nested resource access.

5) At most one task per core can occur in any resource queue.

Proof by contradiction. Assume two tasks of the same core occur in any resource queue. In Figure 8.6 can be seen what steps a task has to go through to end up in a resource queue. When a task gets activated and requests a resource within the GetResource() function the system ceiling is increased to a priority level that is non-preemptive (its MSB gets set) Listing 5.6 line 17. Subsequently a function spin_in() is called, in which the task will start spinning if the resource was not available. The system ceiling ensures no task can preempt the spinning task. We derive a contradiction.

6) At most one task has access to a resource R_q.

Proof by contradiction. Assume that two tasks have access to the same resource. Figure 8.6 illustrates what steps a task has to go through to get access to a resource. The only way for a task to receive access to a resource is by the GetResource() function. A task releases access to a resource by use of ReleaseResource(). When a task issues the request to access a resource R_q by GetResource(R_q), the task is only able to read or write to R_q after the function GetResource(R_q) has terminated. For two tasks to have access to a resource, both tasks should request access to R_q by completing (executing until return) the function GetResource(R_q). Within the GetResource() function spin_in() is called, as shown in Listing 5.6 line 17. The task gets exclusive access to the resource queue "EE_hal_spin_status" by means of a mutex. A check is performed whether there is any task in the queue by checking the tail of the resource queue (Listing 8.4 line 6). Either 1) the resource is not in use and there are no waiting tasks or 2) at least one task is using/ requesting the resource. Proof by case distinction.

Case1: the resource is not in use and there are no waiting tasks. The check whether the preceding task has access to the resource Listing 8.4 line 6 is false and spin_lock stays at its initial value 0. Task ID is stored in the resource queue line 8. The tail pointer is updated to point to the task that just entered the resource queue line 10. The mutex gets released and the resource queue becomes available again (Listing 8.4 line 12). Since the local spin_lock is unlocked (0) the task gets access to the resource. Note that only one task has access not two, we derive a contradiction.

Performance of resource access protocols

Thursday 30th June, 2016 18:15
Case 2: the task requests access to the resource queue by means of GetResource($R_q$) while there exist at least one task in the queue. The check whether the preceding task has access to the resource Listing 8.4 line 6 is false and spin_lock becomes locked (1). The task is added at the tail of the resource queue (Listing 8.4 line 8). The mutex gets released and the resource queue becomes available again. Now the core will start to poll on the local spin_lock until it gets unlocked by an interrupt routine (as Figure 8.6 RN_execute() illustrates). The local spin_lock can only become unlocked by release of a preceding task. By releasing access by the preceding task there will again be only one task having access to the resource. It is therefore not possible for 2 tasks to have access at the same time. We derive a contradiction.

8) If the local spin_lock of core $P_k$ is unlocked (0) and $\tau_i \in T_{P_k}$ occurs at the head of the resource queue of $R_q$, then $\tau_i$ has access to $R_q$.

At initialization all the cores have their spin_lock unlocked (0), yet no task is at the head of the resource queue. For a task to occur in the resource queue at the head of the queue the task should request access to a resource as is illustrated in Figure 8.6. Within the GetResource() function spin_in() is called, as shown in Listing 5.6 line 17. The task gets exclusive access to the resource queue "EE_hal_spin_status" by means of a mutex. A check is performed whether there is any task in the queue by checking the tail of the resource queue (Listing 8.4 line 6). Either 1) the resource is not in use and there are no waiting tasks or 2) at least one task is using/ Requesting the resource. Proof by case distinction.

Case 1: the resource is not in use and there are no waiting tasks. The check whether the preceding task has access to the resource Listing 8.4 line 6 is false and spin_lock stays at its initial value unlocked (0). Task ID is stored in the resource queue line 8, since it is the first item the resource queue the task occurs at the head. The tail pointer is updated to point to the task that just entered the resource queue line 10. The mutex gets released and the resource queue becomes available again (Listing 8.4 line 12). Since the local spin_lock is unlocked (0) the task gets access to the resource.

Case 2: the task $\tau_i$ requests access to the resource queue by means of GetResource($R_q$) while there exist at least one task in the queue. The check whether the preceding task has access to the resource Listing 8.4 line 6 is false and spin_lock becomes locked (1). The task is added at the tail of the resource queue (Listing 8.4 line 8). The mutex gets released and the resource queue becomes available again. Now the core will start to poll on the local spin_lock until it gets unlocked by an interrupt routine (as Figure 8.6 RN_execute() illustrates). The local spin_lock can only become unlocked by release of a preceding task. By releasing access by the preceding task the local spin_lock of $P_k | \tau_i \in T_{P_k}$ becomes unlocked (0) $\tau_i$ ends up at the head of the queue and access is granted.

9) After GetResource($R_q$) and before ReleaseResource($R_q$) a task has access to $R_q$.

In Figure 8.6 in the body of task() can be seen that a task is allowed access after GetResource() and before ReleaseResource().

10) The number of received interrupts regarding $R_q$ on a core is at most 1 larger than the number of processed interrupts regarding $R_q$ on a core.

Proof by contradiction, assume more than one difference between the number of received and processed interrupts regarding $R_q$ on a core. Figure 8.6 illustrates what steps are taken during the Request and Release of a resource. The only function that can send an interrupt regarding a resource is the spin_out() function. A. Thus either the same core should send an interrupt to the receiving core before processing it. B. Or two different cores should send the interrupt to the receiving core. The sending core sends the interrupt to the core that is waiting on a resource, as can be seen in Listing 8.4 line 23. Thus the receiving core should be a waiting core.

Case distinction
A) Assume that one core sends both interrupts. This can be done because the core first releases a resource, accesses a resource and then releases it again. If it accesses the same resource twice the core will end up in the resource queue where the waiting task is already present, which did not process its interrupt thus that resource is not available yet. If the sending core access different resources, then the waiting core can only be in one of those waiting queues since we do not allow nested resource access. (In practice its possible to make use of nested resource access, but then there might be more unprocessed interrupts thus is the statement not true).
CHAPTER 8. SPINNING ON A LOCAL SPIN_LOCK

B) Assume that two cores send an interrupt. With nested resource access, you only spin on 1 resource at the time. In order to wait for multiple global resources simultaneously, you either need a dedicated primitive allowing you to do so or multiple tasks on a core waiting simultaneously for different resources. The former is not provided and the latter cannot happen, because spinning is non-pre-emptive. Finally, there can be at most one task accessing a resource (mutual-exclusive access), hence there cannot be multiple tasks that send the interrupt for the same resource.

11) If the local spin_lock is locked (1), a task on the core occurs in a resource queue.
In Listing 8.4 line 7 the local spin_lock gets locked (1), it is the only place in the code where the value of the local spin_lock can become locked (1). The next line of code that is executed puts the task into the resource queue. Since the access to the resource queue on line 7 and 8 are protected by a mutex it is conceptually as if both actions happen at the same instance.

12) If the local spin_lock of core $P_k$ is unlocked (0), either no task $\tau_i | \tau_i \in TP_k$ occurs in any resource queue or a task $\tau_i$ has access to a resource $R_q$.
Proof by contradiction. Assume that the local spin_lock is unlocked (0) and task $\tau_i | \tau_i \in TP_k$ occurs in any resource queue while $\tau_i$ has no access to resource $R_q$. The local spin_lock can only be unlocked (0) by initialization, or if it becomes unlocked (0) by an interrupt routine.

For a task to occur in the resource queue at the head of the queue the task should request access to a resource as is illustrated in Figure 8.6. Within the GetResource() function spin_in() is called, as shown in Listing 5.6 line 17. The task gets exclusive access to the resource queue "EE_hal_spin_status" by means of a mutex. A check is performed whether there is any task in the queue by checking the tail of the resource queue (Listing 8.4 line 6). Either 1) the resource is not in use and there are no waiting tasks or 2) at least one task is using/ requesting the resource. Proof by case distinction.

Case1: the resource is not in use and there are no waiting tasks. Task $\tau_i$ ends up at the head of the resource queue. In Listing 8.4 line 14, the check whether the local spin_lock is locked escapes the while loop directly and the task receives access. We derive a contradiction.

Case2: the task $\tau_i$ requests access to the resource queue by means of GetResource($R_q$) while there exist at least one task in the resource queue. The check whether the preceding task has access to the resource Listing 8.4 line 6 is false and spin_lock becomes locked (1). We derive a contradiction.

13) If the number of received interrupts regarding resource access on a core $P_k$ larger is than the number of processed interrupts, then the local spin_lock of $P_k$ is locked (1), $\tau_i | \tau_i \in TP_k$ occurs at the head of the resource queue of $R_q$ (but $\tau_i$ has no access to $R_q$ yet).
Note that we already proved in 11 that the number of interrupts received can be at most one larger than the number of processed interrupts. We can thus conclude that the number of received interrupts is one more than the number of processed interrupts. Proof by contradiction we assume a core has received one interrupt more regarding R on a core $P_k$ than it has processed, and either the spin_lock denotes unlocked (0) or the task does not occur at the head of the resource queue of $R_q$.

A releasing task checks which task it needs to notify about the release of a resource as presented in Listing 8.4 line 19. Any core $P_k$ can only receive an interrupt regarding release of resource $R_q$ if a task $\tau_i | \tau_i \in TP_k$ requested access to $R_q$ and ended up in the resource queue. For any task $\tau_i$ end up in a resource queue (line 8 Listing 8.4 it needs to request access to the resource. As a part of the function spin_in() a check is performed whether there is any task in the queue by checking the tail of the resource queue (Listing 8.4 line 6). Either 1) the resource is not in use and there are no waiting tasks or 2) at least one task is using/ requesting the resource. Proof by case distinction.

Case1: the resource is not in use and there are no waiting tasks. The task ends up at the head of the resource queue 8.4 line 8 and we derive a contradiction.
Case2: the task $\tau_i$ requests access to the resource queue by means of GetResource($R_q$) while there exist at least one task in the queue. The check whether the preceding task has access to the resource Listing 8.4 line 6 is false and spin_lock becomes locked (1). We derive a contradiction.
8.5 Measurements

Here we present the measurement plan and the measurement results in time and memory used by the RAP. The purpose of the measurement is to make the analyses a better representation of the system. The analysis does not yet take the overhead introduced by the implementation in consideration.

We need to determine the time and space (overhead) required by the protocols CP, ČP and HP or any protocol in that range [CP, HP]. The time we want to measure is the amount of cycles it takes to perform the actions required to execute the RAP. The theoretical model of ČP outperforms HP, all task-set scheduable under HP are scheduable under ČP, while ČP can schedule tasks-set that HP cannot schedule. In practise execution of the algorithm implementing ČP might not always be shorter than HP. Our interest is the time it takes to execute the additional actions required to implement ČP. The time measurements provide the time in cycles the execution takes of several parts of the kernel code. The memory we want to measure is the amount of bytes used by data-structures or used by the stack. We decided to specify, design and measure the changes to the resource access protocol in parts. We consider three stages and deal with each stage separately (incrementally). As mentioned in Chapter 6 we divided the implementation into 3 parts. Of which spinning on a local spin_lock in the MSRP-protocol is the first step.

The board provides two measurement types. A timer can be used with a precision of 1µs accuracy. The FPGA runs with a frequency of 50MHz. To make a comparison, a read instruction to memory takes about 100 cycles, thus 2µs. Another measurement with more accuracy is using a high precision component in the hardware designer (Quartus). This will provide clock tick precision. Directly starting and stopping the timer requires 3 cycles. The bad property is that this timer cannot be read without stopping the timer. It is sufficient to measure for instance the time between release and completion of a task. It is not possible to start the timer and read the time at various events without stopping the timer in the meanwhile. There are at most 8 of these timers available thus it is possible to do 8 measurements in parallel.

The preferred set-up contains one core printing values (measurement data) via UART. Thus all the measurement data needs to be saved in a shared data-structure. Core P₀ can print the data after all the measurement data is collected.

7Be aware: we essentially made another versions of HP where spinning is done on a local (instead of a global) bit. The original HP and “new” HP will therefore already perform differently.
8.5.1 Measurement of memory usage

We can distinguish multiple sources of memory usage such as dynamic and static memory usage. Examples of static memory are the size of the binary file we load into the board. The size of data-structures that are defined and do not change over time are also static memory. An example of dynamic memory usage is for instance the memory used for the stack. Also data-structures that grow over time or are created in runtime contribute to dynamic memory usage.

Size of program file

A binary file is used to store the program code in the memory of the FPGA or in off-chip memory. At the end of compiling the program code the size of the binary is displayed per core. Thus we can determine the program size for several configurations as a function of for instance # of tasks \( n \) and # of cores \( m \).

Size of static data-structures

A lot of the data-structures are defined in .\( h \) files. These data-structures already exist at startup of the OS and do no change in size. The size of these data-structures can be calculated (offline) without running the code. The difference between the MSRP with interrupts and without interrupts can be determined by the additional or reduced data-structures.

Size of stack

When the hardware description is configured it is possible to set the amount of memory available for each core. For instance each core gets 64 kbyte of memory to its disposal. If the program memory (read only memory and read write memory) required is 30 kbyte then the stack and heap have effectively the remaining 64-30=34 kbyte to their disposal. This will not change over time. The stack will start pointing at a defined starting place in the remaining 34 kbyte. Anytime a part of the remaining executing program needs to be saved to be able to continue later the code is put on the stack. An example is the preemption of a task. In case the stack pointer keeps incrementing due to a growing stack it is possible that the heap gets overwritten and later even the program memory. To measure the stack size we can print the index of the stack pointer. We can do this at preemption and resuming a task. This is not enough to derive the worst-case, because the nesting of functions of the task at the top of the stack is not shown.

Size of dynamic data-structures

Within functions often variables are created such as counter variables for loops. All data that only exist within the scope of a function is only required during the runtime of such a function. After the function returns can those bytes be used to store other data-structures. We can identify the data size used in a function. Due to mutual exclusion only few of the functions require their values to be stored at the same time. These "dynamic" variables are stored on the (statically reserved) stack.

---

\( ^8 \)A fixed amount of memory will be reserved for the stack(s). Hence, although the memory is used during run-time, the allocation is fixed.
8.5.2 Measurement of time overhead

Figure 8.7 illustrates what the ideal (in the sense that all overhead is ignored) MRSP implementation timeline looks like, where requesting, acquiring and releasing of the resource is done without any overhead. Of course this is not possible in practice. Figure 8.8 illustrates what overheads are introduced in the ERIKA implementation of the MSRP RAP. Requesting, acquiring and releasing the resource takes time. Figure 8.9 illustrates what overheads are introduced in the local spinning implementation of the MSRP RAP. The actions that contribute to overhead are: requesting, acquiring and releasing access to the resource. In the implementation of HP based on local spinning the releasing is done by means of an interrupt which induces a more complex form of overhead.

![Figure 8.7: HP/MSRP without overhead](image)

![Figure 8.8: HP/MSRP spinning on a global spin_lock](image)

![Figure 8.9: HP/MSRP spinning on a local spin_lock](image)

The original ERIKA OS and hardware description makes use of one mutex that controls the access to both the resource queue and the interrupt message buffer. It is no problem to extend the ERIKA OS and hardware description to use two mutexes for mutual exclusive access. The new design uses two mutexes, this allows the overhead of the resource queue and the interrupt message to be independent from each other. The maximum value among access times of the resource queue is used for worst-case analysis. The maximum value among access times of the message buffer is used for worst-case analysis.

The measured terms would not vary in case we would allow for other routines to use the interrupt message and its mutex, such as remote activation of tasks. The activation of a remote task would just use the functions send_rn() and handle_rn(), its code would not change and the time it takes would not vary. The worst-case time any message buffer access takes is independent from the type of RN messages used in the system. The execution time of rn_execute depends on the type of RN messages.

---

The worst-case access time of the resource queue is the time required to add or remove a request to the queue and is independent of the time the resource access takes.
CHAPTER 8. SPINNING ON A LOCAL SPIN_LOCK

Measurement overhead original spin_in() and spin_out()

Let $\Delta_a$ denote the time (in cycles) that overhead of code\textsubscript{a} takes.
Let $\Theta_a$ denote the worst-case time that overhead of code\textsubscript{a} takes, including mutex interference of other cores.

$\Delta_a$ is the time it takes for the requesting task to put itself into the resource queue.
$\Delta_a$ is the time it takes to check the global spin-lock value (1 polling loop).
$\Delta_a$ is the time it takes a releasing task to unlock the global spin-lock value.

The code\textsubscript{a}, code\textsubscript{b} and code\textsubscript{c} in Listing 8.5 take $\Delta_a$, $\Delta_b$ and $\Delta_c$ cycles in Figure 8.8 and is part of spin\textsubscript{in()} and spin\textsubscript{out()} in Figure 8.11.

```
#include <locking.h>

EE_hal_spin_in(Rq){
    Determine its own spinning address + spinning value // 128 cycles
    Acquire_hardware_lock
    Determine the previous spinning address + previous spinning value
    Store its own spinning address + spinning value
    Release_hardware_lock //code\textsubscript{a} takes $\Delta_a = 234$ cycles
    Determine the previous spinning address //42 cycles
    Determine the previous spinning value //48 cycles
    while (*spinning address == spinning value); //code\textsubscript{b} takes $\Delta_b = 50$ cycles
}

EE_hal_spin_out(Rq){
    Toggle its own spinning value //code\textsubscript{c} takes $\Delta_c = 129$ cycles
}
```

Listing 8.5: Pseudo code original spin\textsubscript{in()} and spin\textsubscript{out()}

Note that Figure 8.8 and 8.10 are the same apart from the colour. The purpose of colours in Figure 8.8 is to show which parts share a mutex. The purpose of the colours in Figure 8.10 is to shown which code is mapped to which time overhead.

Two things can happen when the core tries to acquire the hardware lock on line 4 of Listing 8.5:
1. The hardware lock is available and the core locks the mutex and continues its operation.
2. It might be the case that the hardware lock (mutex) is already locked by another core. Each time another core adds itself to the resource queue the registering in the queue for the current core is delayed by overhead $\Delta_a$.

In [40] is described how the mutex operates, its only possible to do a set and test instruction, there is no ordering. Stated differently, it is possible that a core $P_0$ may have to wait twice for a core $P_1$ before acquiring the mutex. Note if you take this into account than the worst-case time becomes infinite. The original ERIKA implementation already had this phenomena and did not explicitly mention it. The effect can probably be ignored unless either $m$ increases or the time it takes to access the resource queue becomes longer than the critical section itself, in which case the response time becomes infinite independent of the RAP used but as a result of the hardware.
In Listing 8.5 the code checks whether the global spin lock is locked or unlocked. This continuously checking of a global data-structure might have influence on the access times to shared memory. In the best case situation checks the task the spin lock one clock tick after the spin-lock is unlocked. In the worst case situation checks the task the spin-lock one clock tick before the spin-lock gets unlocked and performs an additional checking cycle. The worst-case overhead is thus two cycles minus one clock tick e.g., $\Theta = 2\Delta_b - 1$.

In Listing 8.5 the code is the time it takes for a releasing task to notify the requesting task. The release involves setting the global spin lock to unlocked. To unlock the spin lock, no access to any mutex is required. The worst-case overhead is equal to $\Theta = \Delta_c$. The overhead will later be compared with the time the release takes in the implementation by spinning on a local spin lock.

The non-coloured code in Listing 8.5 shows the preparation of the value to be written in shared data and preparation of the pointer to the spinning address. The non-coloured code has no interaction with other cores. To reduce the number of colours used, we choose to not include these overheads in the time-line.

![Figure 8.11: Overview of locking and unlocking a resource (colors map to overhead)](image)
CHAPTER 8. SPINNING ON A LOCAL SPIN_LOCK

Mapping of overhead time-line to pseudo code revised spin_in() and spin_out()

Figure 8.12: Overhead is introduced by queue management and updating local spin_lock

\[ \Delta_d \] The time it takes for a requesting task to put itself into the resource queue.
\[ \Delta_e \] The time it takes to check the local spin-lock value (1 polling loop).
\[ \Delta_f \] The time it takes for a releasing task to read from the resource queue to identify which core has the next requesting task in the resource queue.
\[ \Delta_b, \text{send} \] The time it takes to fill a RN message buffer with information that the resource is released.
\[ \Delta_b, \text{receive} \] The time it takes to initiate the interrupt routine on receiving core.
\[ \Delta_i \] The time it takes to read the IH (inside interrupt handler) message buffer for the information about the IPIC inter core communication.
\[ \Delta_j \] The time it takes to unlock the local spin-lock.
\[ \Delta_k \] The time it takes to read the IH message buffer again to detect messages that took place in the meanwhile.

The code \( d \), code \( e \) and code \( f \) in Listing 8.6 and Listing 8.7 take \( \Delta_d, \Delta_e \) and \( \Delta_f \) cycles in Figure 8.12 and are part of spin_in() and spin_out() in Figure 8.11
The code \( g \) and code \( h \) in Listing 8.8 take \( \Delta_g \) and \( \Delta_h \) cycles in Figure 8.12 and are part of rn_send in Figure 8.11
The code \( i \), code \( j \) and code \( k \) in Listing 8.9 take \( \Delta_i \), \( \Delta_j \) and \( \Delta_k \) cycles respectively in Figure 8.12 and are part of rn_handler in Figure 8.11

Listing 8.6: Pseudo code spin_in() (Measurement of HP with local spin_lock)

```c
EE_hal_spin_in(Rq)
{ // let \( \tau_i \) denote the task calling spin_in(Rq)
  Acquire_hardware_lock
  Check if the resource Rq is locked and we need to poll{
    Set its local spin_lock to denote locked (1)
    Store \( \tau_i \) in the resource queue where the task one place ahead in the resource queue will check
  }
  Store \( \tau_i \) in the resource queue place of the current core (used to check existence of a waiting task)
  Release_hardware_lock //code \( d \) takes \( \Delta_d = 379 \) cycles
  while (local spin_lock is locked (1)); //code \( e \) takes \( \Delta_e = 50 \) cycles
}
```

Listing 8.7: Pseudo code spin_out() (Measurement of HP with local spin_lock)

```c
EE_hal_spin_out(Rq)
{ // let \( \tau_i \) denote the task calling spin_out(Rq)
  Acquire_hardware_lock
  Determine the waiting task
  Remove the waiting task from the head of the resource queue
  Release_hardware_lock //code \( f \) takes \( \Delta_f = 251 \) cycles
  If the waiting task differs from \( \tau_i \){ //40cycles
    Send a remote notification //Sum of times in Listing 7.4
  }
}
```

The functions spin_in() and spin_out() both use the resource queue to control the access to a shared resource. Both "code\( d \)" and "code\( f \)" share a mutex to provide the exclusive access. It is important to know the time \( \Delta_d \) and \( \Delta_f \) take, they determine the worst-case time.

\[ \Theta_d = \max(\Delta_d, \Delta_f) \ast (m - 1) + \Delta_d \]
\[ \Theta_f = \max(\Delta_d, \Delta_f) \ast (m - 1) + \Delta_f \]
CHAPTER 8. SPINNING ON A LOCAL SPIN LOCK

Note that in the original `spin_in()` function it was not necessary to exclusively access the resource queue upon releasing of a shared resource. Only the entering of the resource queue was regulated with a mutex. The reason that the original `spin_in()` allowed this was:

A task that requests access to the resource queue while it is in use will look at the preceding tasks spinning address. If the requesting task looks before release, the global spin_lock will represent locked and the requesting task will continuously poll. If the requesting task looks after release, the global spin_lock will represent unlocked and the requesting task will access the resource. The moment the releasing tasks updates the global spin_lock has no influence on the behaviour.

It the new situation the releasing core needs to notify the waiting core via an interrupt. The releasing task needs to know the next requesting task in the resource queue. The only way to do this is to check in the resource queue to identify which task to notify. The release contains two steps: first by reading the requesting taskID if any. Second is storing that the resource is released by removing the taskID (which is either the releasing or requesting task) from the resource queue. The following scenario could happen in case checking for a requesting task and removal from the resource queue would take place non-exclusive:

1. Releasing task would check the resource queue and see that there is no task waiting for the resource.
2. Requesting task would check the resource queue and see the resource is in use.
3. Releasing task would remove itself from the queue.
4. Requesting task would store its taskID to register it is waiting.

The releasing of a resource by the releasing core consists of 2 actions: checking and updating. Instead of only updating in the original `spin_in()`. The easiest way to guarantee that the actions happen in correct order is by using exclusive access.

```c
rn_send()
{
    Disable interrupts(); //73 cycles
    Original spin_in()
    Store that the receiving core interrupt handler has to do an additional cycle
    If request was not queued before, insert it into the pending requests
    Increase the pending counter
    Set the type in the remote notification
    Original spin_out(); //code\_p takes \( \Delta_p = 1414 \) cycles
    Check if we should raise a new inter-processor interrupt //code\_h takes \( \Delta_h = 418 \) cycles (sending)
    Enable interrupts() //51 cycles
}
```

Listing 8.8: Pseudo code of `rn_send()`, measurements

```c
void rn_handler(void){ //code\_h takes \( \Delta_h = 374 \) cycles (receiving)
    determine which RN message buffer we need (SW is 0 or 1) //54 cycles
    while (there are pending RN){
        disable interrupts //73 cycles
        spin_in();
        switch pending requests data-structure
        spin_out(); //code\_s takes \( \Delta_s = 542 \) cycles
        enable interrupts //51 cycles
        for (as many as there are pending RN) {
            rn_execute();
        }
        //code\_j takes \( \Delta_j = 341 \) cycles
        Update the first RN message to -1 (indicates empty) //71 cycles
        disable interrupts //73 cycles
        spin_in();
        switch pending requests data-structure
        spin_out() //code\_k takes \( \Delta_k = 605 \) cycles
        enable interrupts //51 cycles
        Determine whether there are pending RN //41 cycles
    }
}
```

Listing 8.9: Pseudo code of `rn_handler()`, measurements

```c
void rn_execute(RN\_type){
    If the RN is of type RN\_ReleaseResource indicating a resource is released then{
        Reset the local spin_lock to unlocked (0)
    }
}
```

Listing 8.10: Pseudo code of `rn_execute()`, measurements

Performance of resource access protocols 83 Thursday 30th June, 2016 18:15
CHAPTER 8. SPINNING ON A LOCAL SPIN LOCK

Figure 8.13: Overview of locking and unlocking a resource (colors map to overhead)
8.5.3 Measurement results

The measurement set-up consist of 2 cores in both hardware (containing only on-chip memory) and software. Each core has one task that once activated does not terminate. Both tasks consist of a while(1) loop that executes GetResource() directly followed by ReleaseResource(). Core 0 has also the statement printf("%d", measurement time). The measurements can be done by starting a timer (precision cycle counter) before the code that is going to be measured and stop the timer after the execution of the code. Each measurement is done around 10,000 times. There is no variance except for the case where both cores try to access the same resource simultaneously. In Figure 8.14a and 8.14b we present the timings we want to measure. To recall what each part of the diagram represents it is possible to look at Listing 8.5, 8.6 and 8.9 for the pseudo code and Figure 8.11 and 8.13 for the diagrams.

![Figure 8.14: A comparison of the approaches](image)

<table>
<thead>
<tr>
<th>Name</th>
<th>$\Delta_a$</th>
<th>$\Delta_b$</th>
<th>$\Delta_c$</th>
<th>$\Delta_d$</th>
<th>$\Delta_e$</th>
<th>$\Delta_f$</th>
<th>$\Delta_g$</th>
<th>$\Delta_h$</th>
<th>$\Delta_i$</th>
<th>$\Delta_j$</th>
<th>$\Delta_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>234</td>
<td>50</td>
<td>129</td>
<td>379</td>
<td>40</td>
<td>251</td>
<td>1414</td>
<td>418</td>
<td>374</td>
<td>542</td>
<td>341</td>
</tr>
<tr>
<td>Interference</td>
<td>$O(m)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(m)$</td>
<td>$O(1)$</td>
<td>$O(m)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(m)$</td>
<td>$O(1)$</td>
<td>$O(m)$</td>
</tr>
</tbody>
</table>

Table 8.1: The measured timing overhead in cycles, n is number of cores in system

Table 8.1 shows the measurement results given as a decimal amount of cycles. The interference row indicates whether the task might have to wait on other cores. The number cores that can interfere is $O(m)$ (big-O notation), where $m$ is the number of cores in the system. The actual amount of interference depends on how many of the cores have tasks that use a global resource and whether all the cores can access the primitive at the same time\textsuperscript{11}. The duration of the interference is worst-case the largest amongst $\Delta_4$ execution times that share the mutex

![Figure 8.15: The measurement results (allows to compare the relative size)](image)

\textsuperscript{11}By giving each resource queue $R_q$ a separate mutex, the worst-case access time would become a function of the number of cores that share the resource instead of just number of cores, the same holds for message buffer.
Figure 8.15a and 8.15b illustrate the measured overheads in cycles/10. Each overhead in the figures is the time the intrinsic overhead takes. Note that the interference of other cores has not been taken into account (in the figure). What can be observed is that the sending and receiving of the interrupt has the significant overhead. The overhead of queue management \( \Delta_a \) and \( \Delta_c \) in Figure 8.15a change very little to \( \Delta_d \) and \( \Delta_f \) in Figure 8.15b. The spinning on a local spin-lock \( \Delta_e \) is short compared to spinning on a global spin-lock \( \Delta_b \), that \( \Delta_e \) is almost not visible.

<table>
<thead>
<tr>
<th>Function</th>
<th>Listing code</th>
<th>Listing pseudo code with timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>spin_in() (original)</td>
<td>5.9</td>
<td>8.5</td>
</tr>
<tr>
<td>spin_out() (original)</td>
<td>5.9</td>
<td>8.5</td>
</tr>
<tr>
<td>spin_in() (local spin_lock)</td>
<td>8.4</td>
<td>8.6</td>
</tr>
<tr>
<td>spin_out() (local spin_lock)</td>
<td>8.4</td>
<td>8.7</td>
</tr>
<tr>
<td>rn_handler()</td>
<td>5.16</td>
<td>8.9</td>
</tr>
<tr>
<td>rn_execute()</td>
<td>5.16</td>
<td>8.9</td>
</tr>
<tr>
<td>rn_send()</td>
<td>5.13</td>
<td>8.8</td>
</tr>
</tbody>
</table>

Table 8.2: The code and pseudo code with timing of the functions used in the overhead measurement

**Worst-case overhead**

*It is possible to determine the worst case overheads for each term from the measured overheads.*

In original spin_in() code exclusively uses the mutex

\[ \Theta_a = \Delta_a \times m = 234 \times m. \]

The code\( e \) and code\( f \) share a mutex to control the resource queue access.

\[ \Theta_d = \Delta_d + \max\{\Delta_d, \Delta_f\} \times (m - 1) = 379 + 379 \times (m - 1) = 379 \times m \]
\[ \Theta_f = \Delta_f + \max\{\Delta_d, \Delta_f\} \times (m - 1) = 251 + 379 \times (m - 1) = 379 \times m - 128 \]

All \( m \) cores can simultaneous try to access the ready queue.

The code\( g \), code\( i \) and code\( k \) share a mutex to control the interrupt message buffer access.

\[ \Theta_g = \Delta_g + \max\{\Delta_g, \Delta_i, \Delta_k\} \times (m - 1) = 1414 + 1414 \times (m - 1) \]
\[ \Theta_i = \Delta_i + \max\{\Delta_g, \Delta_i, \Delta_k\} \times (m - 1) = 542 + 1414 \times (m - 1) \]
\[ \Theta_k = \Delta_k + \max\{\Delta_g, \Delta_i, \Delta_k\} \times (m - 1) = 605 + 1414 \times (m - 1) \]

All \( m \) cores can simultaneous try to request access to the message buffer.

In previous computation of \( \Theta_g \), \( \Theta_i \) and \( \Theta_k \) we assumed that other \( (m-2) \) cores can send interrupts due to activation of tasks. If we would only allow the sending of RN of type of remote release, the number of possible cores accessing the mutex decreases. When each core has its own mutex protecting its message buffer, the number of cores acquiring a mutex at the same time decreases even more.

A core releasing a resource requests access to the the message buffer of the core it needs to notify. Due to the range of priorities between \( CP \) and \( HP \) we know there is at most one task per core requesting a resource. Thus there exist at most one sending core trying to access the message buffer of a receiving core. After the RN is received the receiving core tries to access its message buffer. Since there can be at most one task requesting access at the same time (range \( CP-HP \)), the receiving core has also no contention to access its message buffer.\(^{12}\)

\[ \Theta_g = \Delta_g \]
\[ \Theta_i = \Delta_i \]
\[ \Theta_k = \Delta_k \]

The code\( e \), code\( h \) and code\( j \) take constant amount of time, worst-case is the same as average case.

\[ \Theta_e = \Delta_e \]
\[ \Theta_h = \Delta_h \]
\[ \Theta_j = \Delta_j \]

\(^{12}\)The worst-case scenario with multiple types of RN or priorities below \( CP \) would not only get worse, in case a receiving core is already handling RN it might save some time by handling many RN in series without having to lock and unlock its message buffer.
8.5.4 Measurement and analysis

We want to measure the overhead of the resource access protocol when it makes use of interrupts. The measurement makes it possible to include that overhead in the analysis. We give a short recap of the formulas given in [7]. This directly copied from their work as a reference. The purpose is to add a term of overhead.\textsuperscript{13}

The maximum priority among tasks that share a resource $R_q$ is called the resource ceiling.

$$\text{ceil}(R_q) = \max \{ \rho_i | \tau_i \text{ uses } R_q \}$$

The spin lock time every task allocated to processor $P_k$ needs to spend before accessing resource $R_q \in \mathcal{R}$ is bound by:

$$\text{spin}(R_q, P_k) = \sum_{p \in \{P-P_k\}} \max_{\tau_i \in \mathcal{T}_p, \forall b} \omega^b_{bh}$$

worst-case the task has to wait the duration of the longest critical section of each of the other processors before it can start accessing the resource.

The actual worst-case time $C_i'$ is the worst-case execution time increased with the time spend spinning.

$$C_i' = C_i + \sum_{\tau_i \in \mathcal{T}_h} \text{spin}(R_q, P_k)$$

The time spend spinning for task $\tau_i$ is summed over the instances $h$ the critical section occur each spinning on a resource $R_q$.

The blocking time for a task $\tau_i$ can be divided into local and global blocking time.

$$B_i = \max(B_i^\text{local}, B_i^\text{global})$$

$$B_i^\text{local} = \max_{j,h,q} \{ \omega^b_{jh} | (\tau_j \in \mathcal{T}_{P_i} \land (R_q \text{ is local to } P_i) \land (\lambda_i \geq \lambda_j) \land (\lambda_i \leq \text{ceil}(R_q))) \}$$

$$B_i^\text{global} = \max_{j,h,q} \{ \omega^b_{jh} + \text{spin}(R_q, P_k) | (\tau_j \in \mathcal{T}_{P_i} \land (R_q \text{ is global}) \land (\lambda_i \geq \lambda_j)) \}$$

The analysis is a model of the reality. Often it means the more realistic the model becomes the more complex it gets. We therefore decide to add the most significant term first which is the interrupt. Figure 8.16 illustrates what terms remain.

In Figure 8.13 can be seen that both sending as receiving an interrupt involve storing an interrupt message in shared memory. That data-structure is protected by a RAP (the original MSRP protocol). This means that the task releasing a resource cannot directly access the interrupt message and instead starts spinning on that shared resource. The number of cores the sending core has to wait before it can access the interrupt msg data-structure is equal to the number of cores -2. The sending core does not have to wait for itself and for the core to which the sending core will send its interrupt msg.

\textsuperscript{13}Note that the notation of the paper and of Sara are combined, to get the precise notation please use [7].
Chapter 9
Context switch

The design of the flexible spin-lock protocol is realised in three steps of which this chapter contains the third and final step. The flexible spin-lock protocol assigns a spin priority to a task that starts spinning (step 2 Chapter 8). When a task arrives with a higher priority than the spin-priority the spinning task gets preempted and ends up on the stack. Once the resource is released the core containing the spinning task receives an interrupt from the releasing core to notify the release (step 1 Chapter 7). In this chapter describing step 3 we make sure the task requesting the resource gets resumed.

Chapter overview

Requirements p.90 The system ceiling should contain the correct value throughout the operation. The context may have to take place when a resource becomes available and when released.

Specification p.91 The specification contains invariants, which are rules about the system that should not be violated. The invariants impose properties on the system ceiling and variables that track the executing & preemted tasks.

Design p.93 The design of the additional context switch to resume a preempted task needs to consider three aspects: 1) The system ceiling needs to contain bitwise the priority the active task is executing on and the priorities of the tasks on the stack(s). 2) The context switch needs to resume and preempt tasks that already contain a stack when a resource becomes available or is released. 3) The tasks should be stored on stacks in such an way that resuming and preempting is possible (we decided to use separate stacks).

Check invariants p.102 After the design is finished the invariants are checked. A non-formal reasoning is given why the properties hold.

Measurements p.105 The measurements section contains the execution times in cycles. Only three functions contain changes compared to the implementation by spinning on a local spin-lock: spin_in(), execute() and ReleaseResource(). In contrast to the measurement results on local polling of the spin_lock in step 1, the measured times only occur once per event. In the measurements of the context we measure the influence of the code we changed in this third step. Since we do not use any shared data in this step (already implemented in step 1) we do not need to synchronise between cores. The duration of the code execution in this phase of the project is only determined by the core switching, there is no interference from other cores.
9.1 Introduction

Here we present the specification of the third and last step of the changes we need to apply. In this step of developing the new RAP, we may need to resume a spinning task when the resource becomes available and may need to pre-empt it again after release.

To allow a task \( \tau_i \) to resume while a higher priority task has pre-empted the task \( \tau_i \) during spinning means we require at least two stacks. The ERIKA-kernel contains functionality that provides a separate stack for each task. The preemption of the spinning task is done automatically by ERIKA whenever a task arrives that has a priority higher than the system ceiling (including spin-priority of a spinning task). Resuming the spinning task means an additional context switch. We need to let the kernel initiate the context switch the moment the interrupt arrives denoting the release of a resource. A preemption check and switch is originally only done at the moment a task arrives, not when a resource becomes available. We need to make sure that the preemption check is also done when the resource becomes available.

Figure 9.1: The spinning task can be preempted and needs to be resumed in correct manner

Figure 9.1 illustrates a scenario where a task spins on a resource, gets preempted by a higher priority task and is allowed to process again as soon as the resource becomes available. To facilitate the context switching the following changes are applied:

- The priority of the spinning task needs to change according to the preemptive spin-lock protocol.
- A preemption check and context switch need to preempt and resume the spinning task.
- We need at least two stacks.

9.1.1 Spin priorities

We can distinguish the following moments a priority needs to change concerning resource access:

1. A task that requests access to a resource that is available should increase its priority to the maximum priority (by increasing the system ceiling).
2. A task should spin on the spin priority instead of the non-preemptive priority.
3. A task on the stack of which the resource becomes available should increase its priority to the maximum priority.
4. A task releasing access to the resource should return to its original priority.

9.1.2 Context switching

A spinning task \( \tau_i \) must get preempted as soon as a task \( \tau_j \) arrives that has a higher priority than the spinning priority. When a task arrives via function-call ActivateTask() the task is put into the ready queue in the correct order. If the arriving task \( \tau_j \) ends up at the head of the ready queue a preemption check is performed. If priority \( p_j \) is larger than the system ceiling (including spin priority), then \( \tau_j \) will preempt the spinning task \( \tau_i \).

This preemption check the moment a resource becomes available is not implemented in the ERIKA-OS yet. The context switch needs to resume a task that already contains a stack. In the original ERIKA OS containing MSRP only two types of context switches take place, either starting a task without a stack and terminating a task. We need a context switch that preempts the executing task stores its stack and another task containing a stack is resumed.

9.1.3 Additional stacks

The ERIKA OS provides functionality to allow for private stacks. It is allowed to provide each task with a private stack.
CHAPTER 9. CONTEXT SWITCH

9.2 Requirements

1. Initial condition
   (a) The system ceiling contains the priority 0x0.
   (b) Each task has a separate address space to store its stack.
   (c) Each stack is empty.

2. Requesting access to a (global) resource
   (a) The task requesting access sets a bit in the system ceiling corresponding to the spinning priority of the task.
   (b) The requesting task is added to the resource queue.
   (c) If the resource is available the task locks the resource, increases the system ceiling to non-preemptive and accesses the resource.
   (d) If the resource is not available the local spin lock gets locked (1).

3. Releasing access to a (global) resource
   (a) When a task releases access to a resource, the system ceiling resets the bits of the task its spinning priority and the non-preemptive priority (keeps the releasing task priority set).
   (b) If the task $\tau_i$ at the head of the ready queue has the highest priority, then $\tau_i$ becomes the active task.
   (c) If the executing task $\tau_i$ has the highest priority than $\tau_i$ remains the active task.
   (d) If the task $\tau_i$ the task that was last preempted without resuming has the highest priority than $\tau_i$ becomes the active task.
   (e) The releasing task sends an interrupt to the task next in the resource queue awaiting the release of the resource.

4. A task arrives
   (a) Any arriving task is put in the ready queue.
   (b) The ready queue is first ordered by priority then FIFO ordered.
   (c) If the arriving task ends up at the head of the ready queue, a pre-emption check is performed.
   (d) If the arriving task has a higher priority than the system ceiling, the executing task is put on its stack and the new task becomes the executing task.
   (e) If the arriving task becomes the executing task, the system ceiling is increased by setting the bit corresponding to the priority of the arriving task.

5. A task is terminated
   (a) The terminating task is removed from the stack and its stack is empty.
   (b) The priority bit of the terminating task is removed from the system ceiling.
   (c) If the task at the head of the ready queue has the highest priority, the task becomes the executing task and sets its priority bit in the system ceiling.
   (d) If a task at the stack has the highest priority, the task becomes the executing task.
   (e) If there exist no task on the stack nor in the ready queue, the idle task (main.c) becomes the executing thread.

6. Properties of system ceiling
   (a) The highest bit in the system ceiling that is set should be equal to the priority the active task is executing on (including dispatch priority, spin priority, non-preemptive access and resource ceiling).
   (b) The system ceiling bits should contain all the priorities of all tasks on any stack including their priority as a result of resource access.
   (c) The system ceiling of core $P_k$ should have the non-preemptive bit (0x80) set when a task $\tau|\tau_i \in T_{pk}$ accesses a global resource.
   (d) If there exits a task $\tau_i$ with priority $\rho_i = 2^x$ on any stack of core $P_k$ means the $x^{th}$ bit of the system ceiling of core $P_k$ is set.
   (e) If the $x^{th}$ bit of the system ceiling of core $P_k$ is not set than there exist no task $\tau_i$ on the stack with a priority of $2^x$ on any stack of core $P_k$. 
9.3 Specification

The specification of the third step of the resource access protocol describes the context switches that take place.

When the resource becomes available:
- The requesting task is still the executing task, no context switch needs to take place
- The requesting task is not the executing task, a context switch puts the active task on the stack and the requesting task gets resumed.

When the resource is released:
- No task was preempted by the spinning task
  - The executing task has the highest priority, no context switch takes place.
  - The task at the head of the resource has the highest priority, a context switch takes place.
- There exists a task that was preempted by the spinning task
  - The preempted task on the stack has the highest priority, a context switch takes place.
  - The task at the head of the resource has the highest priority, a context switch takes place.

**Tasks on the stack**

In Figure 9.2 illustrates the data-structures that keep track of the tasks:
- \texttt{EE\_stkfirst} used in original ERIKA OS to contain the local task ID of the executing task.
- \texttt{EE\_th\_next[\tau]} used in the original ERIKA to point to the task that is preempted by \tau.
- \texttt{Task\_exec} is the task for which the resource become available (while it was s). The task ID is read from the received RN message. For the duration that \texttt{EE\_stkfirst} does not contain the executing task, does \texttt{Task\_exec} contain the taskID of the executing task.

The tasks \texttt{EE\_stkfirst} and \texttt{EE\_th\_next[\tau]} are already maintained in the original ERIKA OS and keep track of the preemption order of tasks. Since we apply additional context switches a task might have preempted two tasks, it is not sufficient to store just one task ID of any preempted task. Thus \texttt{EE\_th\_next[\tau]} contains the task preempted by \tau. When a resource becomes available we resume the task \texttt{Task\_exec}. Which would mean that should store that it preempted two tasks namely \texttt{EE\_th\_next[\texttt{Task\_exec}]} and \texttt{EE\_stkfirst}. You could imagine the preemption list as a tree (Figure 9.2), in which consists of one path without branches or cycles.

Since we need to store more preemption relation information than fits in only the original data-structures we added \texttt{Task\_exec} which contains the ID of the executing task if \texttt{EE\_stkfirst} is not the executing task. The task \texttt{Task\_exec} preempted is always \texttt{EE\_stkfirst}.

---

**Figure 9.2:** The variables that track the preemption order and additionally the executing task when a resource requesting task is resumed.

<table>
<thead>
<tr>
<th>Executing task</th>
<th>(t=0)</th>
<th>(t=1)</th>
<th>(t=2)</th>
<th>(t=3)</th>
<th>(t=4)</th>
<th>(t=5)</th>
<th>(t=6)</th>
<th>(t=7)</th>
<th>(t=8)</th>
<th>(t=9)</th>
<th>(t=10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{Task_exec}</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>(\tau_1)</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>\texttt{EE_stkfirst}</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>\texttt{EE_th_next[\tau]}</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Figure 9.2 illustrates a time-line with 4 tasks. The table illustrates the value of the data-structures as a function of time. Initially \((t < 0)\) no task is executing and the data-structures all contain -1. Task \(\tau_0\) arrives \((t = 0)\) and starts executing which means \texttt{EE\_stkfirst} is updated to contain the executing task. The variable \texttt{EE\_stkfirst} points to the task that is last added to any stack and is not yet terminated. Task \(\tau_1\) arrives \((t = 1)\) and becomes the executing task. Since \(\tau_1\) preempts \(\tau_0\), \texttt{EE\_th\_next[\tau]} = 0. Task \(\tau_2\) arrives \((t = 3)\) and preempts \(\tau_1\), in this example the spin-priority is lower than \(\tau_2\), if the priority of \(\tau_2\) was lower than the spin priority than the task \(\tau_1\) remained the spinning task. Task \(\tau_3\) arrives \((t = 4)\) and preempts \(\tau_2\).
CHAPTER 9. CONTEXT SWITCH

The resource becomes available \((t = 5)\), the releasing task on a remote core signals the release by sending an RN to core \(P_1\). Upon receiving the RN indicating the release of the resource the task \(Task_{exe}\) for which the resource became available is read from the RN message buffer. A context switch takes place resuming \(Task_{exe}\). We keep the ordering of preempted stacks \(EE_{th_{\text{next}}}[\tau_i]\) as is.

The resource is released \((t = 6)\) by \(\tau_1\), and task \(EE_{stk\text{first}}\) is resumed. All the preempted tasks are resumed as soon as the preempted task gets terminated. At time \(t=7\) gets the task \(\tau_3\) terminated and is the task \(EE_{th_{\text{next}}}[\tau_3]\) resumed, which is \(\tau_2\). Note that \(EE_{th_{\text{next}}}[\tau_i]\) keeps its value even after \(\tau_i\) is terminated.

9.3.1 Invariants initial condition
1. All bits in the bit-mask \(EE_{sys\text{ ceiling}}\) are reset (0).
2. There exists no executing task (that preempted \(EE_{stk\text{first}}\) due to a remote resource release) i.e., \(Task_{exe}\) is empty (-1).
3. The stack of stacks is empty\(^1\), \(EE_{stk\text{first}}\) is \(EE_{\text{NIL}}\) (-1) and \(\forall, \; EE_{th_{\text{next}}}[\tau_i]\) is \(EE_{\text{NIL}}\) (-1).

9.3.2 Invariants during runtime
4. The highest bit set in \(EE_{sys\text{ ceiling}}\) is equal to the priority of the executing task (including resource access and possible spin-priority).
5. The priority of the task at the head of the ready queue is lower than the \(EE_{sys\text{ ceiling}}\).
6. An executing task instance (job) is not in the ready queue.
7. A preempted task instance (job) is not in the ready queue.
8. A task instance (job) that has arrived and is not yet terminated is either: "executing", "on the stack while not at the head of the stack" or "in the ready queue".

\(Task_{exe}\) contains the task ID of an access requesting task that was suspended and must become the executing task. \(Task_{exe}\) contains the task ID for the duration the requesting task is the executing task instead of the task \(EE_{stk\text{first}}\). When \(EE_{stk\text{first}}\) is the executing task, \(Task_{exe}\) is empty (-1).

We prove this by proving (9,10,11) separately:
9. \(Task_{exe}\) contains the task ID of an access requesting task that was suspended and must become the executing task.
10. \(Task_{exe}\) contains the task ID for the duration the requesting task is the executing task instead of the task \(EE_{stk\text{first}}\).
11. When \(EE_{stk\text{first}}\) is the executing task, \(Task_{exe}\) is empty (-1).
12. \(EE_{stk\text{first}}\) contains the task ID of the task that last preempted another task (excluding resuming and preempting the spinning task as a result of a remote release) (The task which was last to start its execution and did not yet terminate).
13. \(EE_{th_{\text{next}}}[\tau_i]\) contains the task ID that is preempted by \(\tau_i\) (excluding resuming and preempting the spinning task due to remote release).
14. A task \(\tau_i\) of which the resource is not locked by any task \(\tau\) and the task \(\tau_i\) occurs at the head of the resource queue is: "being notified by RN".

\(^1\)Each task has a separate stack, we keep track of the order in which tasks have started their execution (same a order of preemption modulo resuming the spinning task) i.e., a stack of tasks.

Thursday 30\(^{th}\) June, 2016 18:15

92 Performance of resource access protocols
9.4 Design

Here we present the design of the third and final step of developing the flexible spin-lock protocol. First we explain how the system ceiling keeps track of the priorities. Then we elaborate on how we provide each task with a separate stack. Furthermore we perform additional context switches when the resource becomes available and when the resource gets released.

9.4.1 Design: priority changes

The EE\textsubscript{sys}ceiling or system ceiling is a parameter that keeps track of the priority the system is currently operating at. The system ceiling is used for comparison with new arriving tasks to determine whether a context switch needs to take place. If the arriving task ends up at the head of the ready queue and has a higher priority than the system ceiling then it will become the executing task. It is necessary to have a correct system ceiling throughout the operation of the system. By updating the system ceiling to the correct priorities associated with the flexible spin-lock model we can make sure the context switches take place at the correct moments.

Figure 9.3: The spinning task can be preempted and needs to be resumed in correct manner

Figure 9.3 illustrates the moments the priority (the system is executing at) changes related to resource access. The priority might change upon requesting, acquiring and releasing access. Acquiring the access can occur if the resource is available at the moment the request is issued. Or the access might be acquired after another core releases access. Figure 9.4 illustrates how the system ceiling EE\textsubscript{sys}ceiling is updated.

1. GetResource() already increases the priority to the non-preemptive priority.
2. spin\textsubscript{in}() checks the availability of the resource, in case we need to spin the system ceiling is decreased and set to the spin-priority\textsuperscript{2}.
3. rm\textsubscript{execute}() changes the local spin\_lock to unlocked and the system ceiling to non-preemptive.
4. ReleaseResource(), starts with a bit-mask containing: all priorities of tasks on the stack, the spin\_priority of a task and the non-preemptive priority. By resetting the spin and non-preemptive priority we keep the priorities of tasks on the stack, we set the executing task’s bit, thereby covering the case where the spin-priority it had was identical to the priority of the executing task.

Figure 9.4: The moments the priority needs to change related to resource access

\textsuperscript{2}The benefit of setting the non-preemptive bit before calling spin\_in() and resetting the non-preemptive bit in case the local spin\_lock is locked, is that no context switch can happen while accessing the resource queue.
CHAPTER 9. CONTEXT SWITCH

GetResource(Rq){
  // mask off the LSB, that indicates whether Rq is a global or a local resource
  Check whether the identifier of the resource does not exceed the number of resources
  Check whether the priority of the active task does not exceed the system ceiling
  Disable interrupts
  Retrieve the active task identifier
  Update an array that keeps track of which task uses which resource
  Update an array that keeps track which resource is in use
  Set the non-preemptive bit in EE_sys_ceiling
  In case this is a global resource, lock the other cores (spin_in())
  Enable interrupts
}

Listing 9.1: Pseudo code of GetResource(), change system ceiling

spin_in(Rq){
  // let \( \tau_i \) denote the task calling spin_in(Rq)
  Acquire_hardware_lock
  Check if the resource Rq is locked and we need to poll
  Set its local polling spin_lock to denote locked (1)
  Store \( \tau_i \) in the resource queue where the task one place ahead in the resource queue will check
  }
  Store \( \tau_i \) in the resource queue place of the current core (used to check existence of a waiting task)
  Let the tail of the resource queue point to the place where \( \tau_i \) will look upon releasing the resource
  Release_hardware_lock
  If the local spin_lock is locked{
    Set the spin_k priority bit in EE_sys_ceiling
    Reset the non-preemptive bit in EE_sys_ceiling
  }
  While (local polling spin_lock is locked (1));
}

Listing 9.2: Pseudo code of spin_in(), change system ceiling

rn_execute(RN){
  If the RN is of type RN_ReleaseResource indicating a resource is released then{
    Set the non-preemptive bit in EE_sys_ceiling
    Reset the local spin_lock to unlocked (0).
  }
}

Listing 9.3: Pseudo code of spin_out(), change system ceiling

ReleaseResource(Rq){
  Mask off the LSB, that indicates whether Rq is a global or a local resource
  Check whether the identifier of the resource does not exceed the number of resources
  Retrieve the active task identifier
  Check whether the priority of the active task does not exceed the system ceiling
  Disable interrupts
  Update an array that keeps track of which task uses which resource
  Update an array that keeps track which resource is in use
  In case this is a global resource, unlock the others cores (spin_out())
  Retrieve the task at the head of the ready queue
  Retrieve the task that was preempted by the spinning task
  Reset the non-preemptive bit in EE_sys_ceiling
  Reset the spin_k priority bit in EE_sys_ceiling
  Set \( \rho_{spin,k} \), the dispatch priority bit in EE_sys_ceiling
  // it might be the case that the spinning priority and the active task have the same priority
  // \( \rho_{spin,k} = \rho_{spin,k} \) in which case we need to re-enable the bit of the active task
  If the priority of the task at the head of the queue is higher than the system ceiling{
    // we have to schedule a ready thread
    The status of the active thread becomes READY (indicating it has a stack)
    The status of the task at the head of the ready queue becomes RUNNING
    The EE_sys_ceiling gets increased with the priority of the task at the head of the ready queue
    The context switch takes place
  - Preempting the active task and putting it onto the stack
  - Removing the task from the head of the ready queue
  - Pointing the program counter to the task that was at the head of the ready queue
  - The execution of the task starts
  Enable interrupts
}

Listing 9.4: Pseudo code of ReleaseResource(), change system ceiling

\(^3\)The spin-priority of \( \mathcal{C}_P \) and \( \mathcal{C}_P \) is fixed per core hence \( \rho_{spin,k} \) could be used. The tool allows to specify a spin priority per task.
9.4.2 Design: private stacks

The ERIKA OS reads from the OIL file the properties of the operating system. Listing 9.5 shows the parts of the OIL file that are required to configure the OS to operate with multiple stacks. Selecting "KERNEL_TYPE= BCC2" and "MULTI_STACK = TRUE" makes sure that the file eecfg.c that is generated contains the definitions MULTI_STACK and BCC2. All of the code used to compile the ERIKA OS is conditional code, depending on whether a certain #define is declared. By defining MULTI_STACK and BCC2 all of the code related to stacks becomes available. For instance the execution of a task preemption will differ in case of MONO, or MULTILSTACK.

```plaintext
OS EE {
  ....
  CPU_DATA = NIOSII {
    MULTI_STACK = TRUE; /* Allows for multi-stacks, MULTI_STACK will be defined enabling parts of the code
    SYS_SIZE = 0x1000;  /* The total stack size for a core
                         /~The software assumes this size, regardless of whether it is available
    } .... /* The total stack should be larger than the sum of the stacks of tasks
    KERNEL_TYPE = BCC2; /* The BCC2 is the kernel type used for MRSP including stacks
  ....
  }; /* The total stack size should be larger than the sum of the stacks of tasks

TASK task0 {
  STACK = PRIVATE{ /* The stack should be defined as PRIVATE, it is possible to have some tasks
   SYS_SIZE = 0x100; /* private and others shared. Here we provide the stack size of the task
    } ; /* Note that this size should be smaller than the core SYS_SIZE. In this example
   .... /* the task stack starts at stackpointer=0, the cpu at stackpointer=0x100
```

Listing 9.5: Parts of the OIL file that are needed to configure a multi-stack OS

For each core it is necessary to define a stack size (SYS_SIZE is the stack-size). It is possible to combine shared and private stacks. All the shared stacks will be put on the same stack. All the private stacks will not share their stack. It is not possible (without changing the code) to put some tasks in one stack and some in another. If the total stack size of a core is 0x1000 and the core contains a task with a private stack of size 0x100. The first 0x100 addresses of the stack are used by the task, the rest is used by the core. When the task overflows its 0x100 memory places it will overwrite the other stack.
9.4.3 Design: context switch

Figure 9.5 illustrates the 3 primitives used for context switching: to activate a task which doesn’t have a stack yet, to terminate a task and to switch between tasks with stacks.

Some scheduling algorithms check periodically whether a context switch needs to take place. In the ERIKA OS there is no periodic check, but event driven switching. Which means that when a task arrives or when a resource is released the preemption check needs to take place.

Figure 9.6: Functions used to activate, terminate and switch between tasks (concerns the original code)

Figure 9.6 illustrates where the context switches are used in the original ERIKA OS. Note that only the ready2staked() and terminate_task() function are used in the MSRP protocol. For the MSRP protocol it is not required to switch between two tasks that already contain a stack.
1. Activating a task without a stack.
When task τ₁ arrives in Figure 9.5 the task is added to the ready queue with the function `RQ_insert()` as illustrated in Figure 9.6. The arriving task will be compared with the system ceiling `E_{E sys_ceiling}` of core `P_{k|τ₁ \in T_{P_k}}` to determine whether a context switch needs to take place. In the original MSRP, HP approach the system ceiling of a spinning task would increase to the non-preemptive priority. Instead in section 9.4.1 we already proposed changes to increase the system ceiling to contain the spin-priority. In the example the priority of task τ₁ is higher than the spin priority of task τ₂ the preemption check (in Figure 9.6) makes sure a context switch will take place. The task ID of the active task and the task at the head of the ready queue are retrieved and the context switch is performed with `Ready2stacked()`. (We keep the activation of tasks in the original ERIKA code as is).

2. Terminate a task with a stack.
At the end of the API-layer code of a task is the function `TerminateTask()`, which terminates the active task and reschedules the remaining tasks that have a stack. In case there is no task left to resume the function returns to thread 0 which is the main program. The behaviour of the original ERIKA kernel performs the correct context switch. To terminate the active task no additional code is required.

3. Context switch to resume the task of which the resource became available.
In chapter 7 we already implemented the HP protocol that notifies the release of a resource via an RN. The received RN unlocks the local spin_lock a task is spinning on. In the new RAP the receiving core might be executing a task different from the spinning task (task τ₁ instead of task τ₂ in Figure 9.5). When the receiving core starts its interrupt routine and handles the RN it should besides unlocking the local spin_lock perform a context switch to resume the spinning task.

```c
rn_execute(RN){
    if the RN is of type a shared resource is released{
        Set the non-preemptive bit in EE_sys_ceiling
        Reset the local spin_lock to denote unlock (0)
        Determine Task_exe
        If the requesting task (Task_exe) is not the executing task (EE_stkfirst) {
            Disable interrupts
            Perform the context switch to resume Task_exe, the task for which the resource became available
            Enable interrupts
        } else{ Task_exe is empty (-1)}
    }
}
```

Listing 9.6: Pseudo code of `rn_execute()`, context switch

Listing 9.6 presents the pseudo-code used to execute the release resource RN. Note that the context switch is done with interrupts disabled. The context switch functions consist of ASM code. In their description is given that the interrupts should be disabled. The context switch function has as argument the task it should resume. When a remote message is received indicating a shared resource became available the message already contains information on which task resume (Task_exe). Either Task_exe is still executing, e.g. spinning on the spin-lock or we perform a context switch and resume Task_exe.
CHAPTER 9. CONTEXT SWITCH

4. Context switch to preempt the spinning task.
When a task $\tau_i$ releases the resource the non-preemptive and spin-priority bit in EE_sys_ceiling get reset. Due to the decreasing the EE_sys_ceiling other tasks have might have a higher priority and should become the executing task. In the original ERIKA OS there where two options: either a task $\tau_j$ arrived during the resource access with a priority higher than EE_sys_ceiling, or the executing task $\tau_i$ has the highest priority and remains the executing task.

In the new RAP we have an additional possibility, there might exist a task $\tau_k$ that was preempted when a resource became available. Lets assume there does not exist a task $\tau_k$ that is preempted by a release of a resource, then the original two options remain valid (execute $\tau_j$ or $\tau_i$). Instead assume a task $\tau_k$ was preempted by $\tau_i$ when the resource became available. This implies that $\rho_k > \rho_i$, since $\tau_k$ did get activated while $\tau_i$ was spinning. Either task $\tau_i$ the preempted task or task $\tau_j$ the task at the head of the ready queue should become the executing task which ever has the highest priority.

```plaintext
ReleaseResource(R)
1 Mask off the MSB, that indicates whether R is a global or a local resource
2 Check whether the identifier of the resource does not exceed the number of resources
3 Retrieve the active task identifier
4 Check whether the priority of the active task does not exceed the system ceiling
5 Disable interrupts
6 Update an array that keeps track of which task uses which resource
7 Update an array that keeps track which resource is in use
8 In case this is a global resource, unlock the others cores (spin_out())
9 Retrieve the task at the head of the ready queue
10 Retrieve the task that was preempted by the spinning task
11 Reset the non-preemptive bit in EE_sys_ceiling
12 Reset the $\rho_{spin}$ priority bit in EE_sys_ceiling
13 Set $\rho_i$ the dispatch priority bit in EE_sys_ceiling
14 If Task_exe is not empty {-1} { 15 Task_exe becomes empty {-1}
16 if (the ready queue is empty) \{ $\rho_j < EE_sys_ceiling$ where $\tau_i$ at head of ready queue \}
17 A context switch takes place to preempt the releasing task and resume EE_stkfirst
18 }
19 }
20 if the ready queue is not empty{
21 if the priority of the task at the head of the queue higher is than the system ceiling{
22 //we have to schedule a ready thread
23 The status of the active thread becomes READY (indicating it has a stack)
24 The status of the task at the head of the ready queue becomes RUNNING
25 The EE_sys_ceiling gets increased with the priority of the task at the head of the ready queue
26 The context switch takes place
27 - Preempting the active task and putting it onto the stack
28 - Removing the task from the head of the ready queue
29 - Pointing the program counter to the task that was at the head of the ready queue
30 - The execution of the task starts
31 }
32 Enable interrupts
33 }
Listing 9.7: Pseudo code ReleaseResource(), context switch
```

Figure 9.7a illustrates where the additional context switches take place as a result of the flexible spin-lock protocol. Figure 9.7b, illustrates an example where task $\tau_1$ starts spinning at $t = 2$ by executing the spin_in() function. When the resource becomes available ($t = 5$) the function rn_execute() increases the system ceiling to non-preemptive (due to the resource access). The local spin_lock is unlocked (0). A check is preformed whether the requesting task Task_exe remained the executing task. In the example $\tau_1$ executes instead of $\tau_1$ thus we need to perform the context switch. The interrupts get disabled, the context is switched and interrupts get enabled again.

Task $\tau_1$ releases access to the resource ($t = 6$) by means of the ReleaseResource() function. We check whether there exists a preempted task (Task_exe is not empty (-1)). We reset Task_pre to empty (-1). Since we know that task exists a task with a higher priority on the stack, i.e. EE_stkfirst we should check whether it has an higher priority than the task at the head of the ready queue. In the example this is the case and a context switch takes place resuming EE_stkfirst. All the tasks that are preempted are resumed by the TerminateTask() function of ERIKA, we did not alter this function and since the EE_th_next[$\tau_i$] contain the correct information of which task preempted which the tasks are handled in correct manner.

Thursday 30th June, 2016 18:15 98 Performance of resource access protocols
Figure 9.7: `rn_execute()` and `ReleaseResource()` perform the additional context switches

Figure 9.9 illustrates the situation where tasks arrive during the resource access. Task $\tau_1$ arrives while $\tau_0$ is spinning on a resource. The resource becomes available and $\tau_0$ is resumed. During the non-preemptive access task $\tau_2$ and $\tau_3$ arrive. After the resource is released $\tau_0$ gets preempted, the task $\tau_3$ at the head of the ready queue and has a higher priority than the system ceiling. Thus $\tau_3$ is resumed instead of $\tau_1$ which is stored in `Task_pre` and in `EE_th_next[\tau_3]`. Task $\tau_4$ arrives preempts $\tau_3$, terminates and resumes $\tau_3$. When $\tau_3$ terminates it reduces the system ceiling. Task $\tau_2$ at the head of the ready queue has a higher priority than the system ceiling and becomes the active task. Since $\tau_3$ is removed from the stack and $\tau_2$ is put on the stack `EE_th_next[\tau_2]` becomes `EE_th_next[\tau_3]`. When $\tau_2$ terminates it resumes the task next on the stack $\tau_1$ which is the task that was preempted when the resource became available. Note that $\tau_1$ is now activated by the context switch `terminate_task()` instead of `stkchange()` and the variable `Task_pre` was not used to resume the task. A task that gets preempted by the release of a resource should get resumed directly after the resource access Figure 9.7a or the task will be resumed in the regular manner by removal of tasks on the stack as in Figure 9.9

Figure 9.8: Context switch example where tasks arrive during the resource access
CHAPTER 9. CONTEXT SWITCH

Figure 9.9: Total diagram

Performance of resource access protocols
9.4.4 In case EE_stkfirst is not the executing task.
The implementation of the context switches required for the CP flexible spin-lock protocol, we should keep track of the tasks to preemp and to resume. We choose to let EE_th_next[τ] track the task preempted by task τ. The task EE_stkfirst contains the task that last acquired a stack (it is the last task that started its execution which is not yet terminated). This means that the assumption that EE_stkfirst contains the taskID of the executing task is not true. We should check whether anywhere in the code the invalidation of this assumption might lead to problems.

We need to take care for correct execution in the function spin_out(). The task determines whether: the releasing task is the task that last requested access or if there exists a requesting task. If there exists a requesting task on an remote core the requesting task has over-written the entry in the resource queue (earlier explained in 8.3). The releasing task checks the entry and if its different from the executing task the releasing task sends an RN to the requesting task.

Due to the changes applied in this step (3. context switch) the EE_stkfirst might not contain the executing task, instead Task_exe might be the executing task. Thus we should make sure that in case Task_exe is the task ID stored in the resource queue we use Task_exe for comparison. Listing 9.8 and 9.9 present how to adapt the code such that the task entry in the resource queue is compared to the executing task.

```
1 EE_hal_spin_out(Rq){
2   let τi denote the task calling spin_out(Rq)
3   Acquire_hardware_lock
4   Determine the waiting task
5   Remove the waiting task from the head of the resource queue
6   Release_hardware_lock
7   //The releasing task τi might be Task_exe or EE_stkfirst
8   if the waiting task differs from τi{
9      Send a remote notification
10   }
11 }
```

Listing 9.8: Pseudo code spin_out(), check the executing task

```
1 EE_hal_spin_out(EE_TYPESPIN m){
2   EE_UINT32 task2notify=0;
3   EE_altera_mutex_spin_in();
4   task2notify=ResourceQ[m][EE_CURRENTCPU];
5   ResourceQ[m][EE_CURRENTCPU]=0xa0;
6   EE_altera_mutex_spin_out();
7   if(task2notify!=GlobalTaskID[EE_stkfirst] \\ (Task_exe == -1 || task2notify!=GlobalTaskID[Task_exe]){
8      register EE_TYPERN_PARAM par;
9      par.pending = 1;
10      EE_rn_send(task2notify, RN_ReleaseResource, par );
11      EE_hal_IRQ_interprocessor((EE_UREG)task2notify);
12 }
13 }
```

Listing 9.9: C-code spin_out(), check the executing task

| spin_out | + | Is adapted as presented in this subsection 9.4.4 |
| Resource is released thus EE_stkfirst gets updated |
| spin_in() | + | Must be extended with checking Task_exe in case of nested resource access |
| GetResource() | + | Must be extended with checking Task_exe in case of nested resource access |
| rn_execute() | + | Must be extended with checking Task_exe in case of nested resource access |

| EE_IRQ_end_instance() | A scheduler function (preemption check) at the end of the interrupt routine |
| Only called from EE_nios2_instacks ASM function used at initialization |

| ActivateTask() | Non-preemptive EE_sys_ceiling, thus execution will not reach EE_stkfirst |
| EE_rq2stk_exchange() | Non-preemptive EE_sys_ceiling, thus execution will not reach EE_stkfirst |
| TerminateTask() | A task should not get terminated before resource release |
| thread_end_instance() | A task should not get terminated before resource release |

Table 9.1: All occurrences of EE_stkfirst

Table 9.1 illustrates the functions that contain EE_stkfirst. A + sign indicates that the value of EE_stkfirst could be read when it contains the task at the head of the stack instead the executing task. The occurrence in spin_out() is taken care of in Listing9.9. The other occurrences can easily be solved by also checking Task_exe. Since we do not allow for nested access this is no issue.
9.5 Check Invariants

To facilitate the context switches we need to update the system ceiling and preemption order of tasks when a global resource becomes available/is released. For each of these variables we defined invariants and provide a reasoning why the invariants hold. In this section we validate the design by checking the invariants given in the specification 9.3.1 and 9.3.2.

9.5.1 Invariants initial condition

1) All bits in the bit-mask EE_sys_ceiling are reset (0).

 EE_TYPEPRIORITY EE_sys_ceiling= 0x0000; (eecfg.c)

The data is initialized with 0.

2) There exists no executing task (that preempted EE_stkfirst due to a remote resource release) i.e., Task_exe is empty (-1).

 EE_TID Task_exe = EE.Nil; (internal.h)

The statement shows the variable is initially empty (-1);

3) The stack of stacks is empty\(^4\), EE_stkfirst is EE.Nil (-1) and \(\forall i, EE.th.next[τ_i] \) is EE.Nil (-1).

 EE_TID EE_stkfirst = EE.Nil; int Task_pre = -1; (eecfg.c)

 EE_TID EE.th.next[EE.MAX_TASK] = {EE.Nil, EE.Nil, EE.Nil}; (eecfg.c)

The data is initialized with EE.Nil (-1);

9.5.2 Invariants runtime condition

4) The highest bit set in EE_sys_ceiling is equal to the priority of the executing task (including resource access and possible spin-priority).

Figure 9.9 illustrates where EE_sys_ceiling changes and where tasks are preempted/resumed. We proof the invariant by assuming the invariant holds before any action and prove that it is still holds after the action. We should check whether any time the system ceiling changes, the active task changes, a resource is requested, becomes available or is released: the EE_sys_ceiling represents the priority of the executing task. Case distinction:

A) The active task changes.

The active task changes as a result of activation, termination or switching between tasks on the stack. If a new task becomes the active task its priority is set in EE_sys_ceiling as part of the ActivateTask() function. If a task gets terminated its priority bit is reset in EE_sys_ceiling, either the task on the stack becomes the active task (its bit is already contained in the EE_sys_ceiling. Or the the task at the head of the ready queue becomes the active tasks in which case the its bit will be set in EE_sys_ceiling. Note that switching between tasks on the stack means that the priority of both tasks is already contained in EE_sys_ceiling. In all cases the execution matches the priority.

B) The resource is requested, becomes available or is released.

If the resource is requested and the task starts spinning in spin_in() the resource gets the spinning priority and resets the non-preemptive priority. If the resource becomes available the task receives an RN, rn_execute() increases the priority to the non-preemptive priority. If the resource is released the function ReleaseResource() resets the spin_priority, the non-preemptive priority and the sets the priority of the active task. In all cases the execution matches the priority.

C) The priority changes.

All of the instances a priority changes in Figure 9.9 are already covert.

\(^4\)Each task has a separate stack, to keep track of which task preempted which task there exists a linked list. The items in the list are LIFO-ordered (like a stack of tasks on the stack).
5) The priority of the task at the head of the ready queue is lower than the EE_sys_ceiling. Proof by contradiction. Assume that there exists a task at the head of the ready queue with a priority lower than EE_sys_ceiling. The task should either arrive at the head of the ready queue while it has a higher priority than EE_sys_ceiling, or the task should be at the head of the ready queue and EE_sys_ceiling should reduce. Case distinction:
   A) The task arrived at the head of the ready queue while it has a higher priority than EE_sys_ceiling. In the function ActivateTask() is a preemption check performed, we derive a contradiction.
   B) The task is at the head of the ready queue, EE_sys_ceiling reduces but the task at the head of the ready queue is not activated. EE_sys_ceiling can be reduced due to release of a resource or termination of a task. If a resource is released in ReleaseResource() and the task at the head of the ready queue has the highest priority it will get activated. If a task is terminated and the task at the head of the ready queue has the highest priority it will get activated. We derive a contradiction.

6) A executing task instance (job) is not in the ready queue. Proof by contradiction. Assume the executing task occurs in a in the ready queue. The only place a item is added to the ready queue is in the function ActivateTask(). A task can become the executing task while it already has a stack, or it has no stack yet. Proof by case distinction. A) The task has no stack, activation by activate task. The task is removed from the head of the queue to get activated (contradiction). B) The task has a stack. The task does not occur in the ready queue anymore. We derive a contradiction.

7) A preempted task instance (job) is not in the ready queue. Proof by contradiction. Assume the preempted task occurs in a in the ready queue. The only place a item is added to the ready queue is in the function ActivateTask(). Before a task can become a preempted task it should have executed. A task can become the executing task while it already has a stack, or it has no stack yet. Proof by case distinction. A) The task has no stack, activation by activate task. The task is removed from the head of the queue to get activated (contradiction). B) The task has a stack. The task does not occur in the ready queue anymore. We derive a contradiction.

8) A task instance (job) that has arrived and is not yet terminated is either: "executing", "on the stack while not at the head of the stack" or "in the ready queue". Assume a task to be available when it arrived and is not yet terminated. Proof by contradiction. Assume a task instance that is available is not executing nor on the stack nor in the ready queue. For a task to be available it needs to be activated. Once activated it occurs in the ready queue. The only way to get removed from the ready queue is by starting to execute the task. Once executing the task can only be preempted or terminated to stop executing. We derive a contradiction.

9) Task_exe contains the task ID of an access requesting task that was suspended and must become the executing task. Proof by contradiction. Assume Task_exe contains a task ID that is different from the access requesting task. The only place where Task_exe is assigned to a task ID (as illustrated in Figure 9.9) is in the run_execute() function where it gets the task ID of the access requesting task is assigned. The task of which the resource became available is either still spinning on the resource or the task is suspended. Proof by case distinction. Assume the task is still spinning, the Task_exe becomes empty (-1). We derive a contradiction. Instead assume the requesting task is suspended and should be resumed. The variable Task_exe contains the access requesting suspended task. We derive a contradiction.

10) Task_exe contains the access requesting task ID for the duration the requesting task is the executing task instead of the task EE_stkfirst. We should prove that when EE_stkfirst is not the executing task, the task Task_exe is the executing task. Proof by contradiction. Assume EE_stkfirst is not the executing task and Task_exe does not contain the executing task. The only moment the EE_stkfirst does not contain the executing task is when a suspended task is resumed. In the function run_execute() takes a context switch place, Task_exe becomes the executing task instead of EE_stkfirst. The task Task_exe remains the executing task until the function ReleaseResource(), where EE_stkfirst becomes the executing task. Concluding either Task_exe or EE_stkfirst contains the executing task. We derive a contradiction.
11) When \( EE_{\text{stkfirst}} \) is the executing task, \( Task_{\text{exe}} \) is empty (-1).

Proof by contradiction. Assume \( EE_{\text{stkfirst}} \) is the executing task and \( Task_{\text{exe}} \) is not empty (-1). The only duration \( Task_{\text{exe}} \) is assigned a value different from empty (-1) is after \( rn_{\text{execute}}() \) when \( Task_{\text{exe}} \) contains the executing task and before \( ReleaseResource() \). In 10) we proved that \( EE_{\text{stkfirst}} \) cant be the executing when \( Task_{\text{exe}} \) contains the executing task. We derive a contradiction.

12) \( EE_{\text{stkfirst}} \) contains the taskID of the task that last preempted a task (excluding resuming and preempting the spinning task as a result of a remote release)

Proof by contradiction. Assume \( EE_{\text{stkfirst}} \) contains the taskID of the task that is not last added to a stack. The only place that \( EE_{\text{stkfirst}} \) is assigned is in the functions \( ActivateTask() \), \( ReleaseResource() \) and \( Th_{\text{end}}(\text{instance}) \). Proof by case distinction. A) \( ActivateTask() \) starts a new task, that ends up on a stack and stores it in \( EE_{\text{stkfirst}} \) (contradiction). B) \( ReleaseResource() \) starts a new task, that ends up on a stack and stores it in \( EE_{\text{stkfirst}} \) (contradiction). C) \( Th_{\text{end}}(\text{instance}) \) removes a task from the stack and stores the next task in the linked list of task on the stack in \( EE_{\text{stkfirst}} \) (contradiction).

13) \( EE_{\text{th_next}}[\tau_i] \) contains the task that is preempted by task \( \tau_i \) (excluding resuming and preempting the spinning task as a result of a remote release). (In case there exits no task \( EE_{\text{th_next}}[\tau_i] \) contains \( EE_{\text{NIL}} \) (-1) ) Proof by induction.

**Base case:** The linked list of tasks tracking the preemption order is initialized with \( EE_{\text{NIL}} \) (-1) Listing: 9.10.

**Step case:** Assume task \( \tau_i \) is executing and has a task \( \tau_i' \) that is preempted by it, which is stored in \( EE_{\text{th_next}}[\tau_i] \). Whenever a task \( \tau_j \) preempts \( \tau_i \) it can do so either by arrival of the task or if a resource becomes available. Proof by case distinction:

A) Task \( \tau_j \) with a priority above the system ceiling arrives. In Figure 9.9 in \( ActivateTask() \) is illustrated that at the moment the context switch takes place, preempting \( \tau_i \) with \( \tau_j \) (temp), then \( EE_{\text{th_next}}[\tau_j] \) becomes \( \tau_i \).

B) Task \( \tau_i \) is preempted when it releases a resource. In Figure 9.9 in \( ReleaseResource() \) is illustrated that at the moment the context switch takes place, preempting \( \tau_i \) with \( \tau_j \) (temp), then \( EE_{\text{th_next}}[\tau_j] \) becomes \( \tau_i \).

13) A task \( \tau_i \) of which the resource is not locked by any task \( \tau \) and the task \( \tau_i \) occurs at the head of the resource queue is: "being notified by RN".

Proof by contradiction. Assume the resource is available but the task is not "being notified by RN" nor "executing". When the resource becomes available the releasing core will send an RN as a result of the \( ReleaseResource() \) function. The receiving core will handle the RN. Part of handling the RN is resuming the task that requested the task in the function \( rn_{\text{execute}}() \) and removing it from the queue. We derive a contradiction.
9.6 Measurements

*This chapter contains the measurement results of the complete flexible spin-lock protocol. A comparison in time between the code used in the ERIKA OS, and the flexible spin-lock code is made. In the end we provide a worst-case time analysis for both ERIKA’s MSRP and the flexible spin-lock protocol.*

Figure 9.9 illustrates a diagram of the flexible spin-lock protocol. In yellow are the parts depicted that changed compared to the original ERIKA OS. Only the functions spin\(_{in}()\), \(\text{rn}\_\text{execute}()\), ReleaseResource() and spin\(_{out}()\) differ from the original code. The changes to spin\(_{out}()\) where already applied (and measured) in changing to spinning on a local spin\(_{lock}\) chapter 7. All the access to shared data protected by mutex was already implemented in STEP 1. The most resent changes are related to allowing the context switch to take place. Since there are no changes related to mutexes, the time each action takes is a fixed amount of time. The code can be measured as a continuous block of code. Depending on the scenario the execution time differs. We measure the time of the changes applied to ReleaseResourc(), \(\text{rn}\_\text{execute}()\) and spin\(_{in}()\) under different scenario's.

**ReleaseResource()**

Figure 9.10 illustrates 3 scenarios that can occur while releasing the resource.

1. The releasing task should remain the executing task \(\Delta_m\).
2. The task at the head of the ready queue becomes the executing task \(\Delta_l\)
3. The task at the head of the stack becomes the executing task \(\Delta_n\)

Since the function ReleaseResource() was not altered while developing the spinning on a local spin\(_{lock}\) we need to measure the time it takes in the original code.

**\(\text{rn}\_\text{execute}()\)**

Figure 9.11 illustrates 2 scenarios that can occur while executing a remote release RN.

1. The task requesting the resource remained the executing task \(\Delta_n\).
2. The task requesting the resource is preempted and should get resumed \(\Delta_p\).

The original \(\text{rn}\_\text{execute}()\) would not be used for resource access. The measurement results of the remote release used in local spinning is given in Section 8.5.

**spin\(_{in}()\)**

Figure 9.12 illustrates 2 scenarios that can occur while checking the availability of the spin\(_{lock}\).

1. The resource is available, i.e. spin\(_{lock}\) is unlocked \(\Delta_q\)
2. The resource is unavailable, i.e. spin\(_{lock}\) is locked \(\Delta_r\).

The code measured does not yet occur in the original nor in the local spinning code.

<table>
<thead>
<tr>
<th>Table 9.2: Measurement results of the context switch in cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReleaseResource()</td>
</tr>
<tr>
<td>--------------------</td>
</tr>
<tr>
<td>Original</td>
</tr>
<tr>
<td>local spinning</td>
</tr>
<tr>
<td>Flexible spin-lock</td>
</tr>
</tbody>
</table>

In Table 9.2 is shown how long the measured durations are. The context switches to resume a spinning task \(\Delta_n\) and to preempt it again \(\Delta_p\) are the most significant changes to the original HP implementation. The context switches take more time then the terms measured in step 1 going from spinning on a global to spinning on a local polling bit. A clear observation is that the flexible spin-lock protocol as implementation has overhead which makes it so that HP can be more efficient than \(\hat{C}P\). Whether \(\hat{C}P\) or HP is more efficient depends on the duration of the critical sections compared to the duration of the context switch.

Only for large \(m\) should the access to the resource queue and message buffer should be considered.

\(\Delta_1\) is the execution of the code in ReleaseResource() before the function spin\(_{in}()\) i.e., disabling the interrupts, error checking and keeping track of resource usage (book keeping). \(\Delta\) The time it takes to release a resource if the task at the ready queue has the highest priority and should become the executing task (switch takes place).

\(\Delta_m\) The time it takes to release a resource if the executing task has the highest priority (no switch takes place).

\(\Delta_n\) The time it takes to release a resource if the task on the stack has the highest priority (EE\_stkfirst) (switch takes place).
Figure 9.10: Measurement of ReleaseResource()

$\Delta_o$: The time it takes to increase the priority and do a preemption check as a part of the interrupt handler (in this case while the requesting task is the executing task).

$\Delta_p$: The time it takes to increase the priority and do a preemption check and perform a context switch as a part of the interrupt handler (in this case while the requesting task is not the executing task).

Figure 9.11: Measurement of $\text{rn}\_\text{execute}()$

$\Delta_q$: The time it takes to check whether the local spin-lock is unlocked if the resource is available.

$\Delta_r$: The time it takes to check whether the local spin-lock is unlocked if the resource is not available and the system ceiling is set to non-preemptive.

Figure 9.12: Measurement of $\text{spin}\_\text{in}()$
Figure 9.13: The terms of the original ERIKA OS

Figure 9.14: The terms of the flexible spin-lock protocol, while the spinning task is not preempted

Figure 9.15: The terms of the flexible spin-lock protocol, while the spinning task gets preempted
9.6.1 A comparison of overhead

To make a fair comparison between the overhead occurring in MSRP and the flexible spin-lock model we should take all the overhead in consideration. The functions that implement the resource access are: spin_in(), spin_out(), GetResource(), ReleaseResource and in the new implementation also rn_handler() and rn_execute(). All those functions except GetResource() are already measured. The GetResource() function stayed unchained, but does add to the overhead terms. The time the GetResource() function takes 2184 cycles. The overhead \( \Delta_s \) denotes this GetResource() overhead. Any resource access overhead takes place when:

- Access to the resource is requested.
- Access to the resource is granted.
- Access to the resource is released.

Figure 9.16 illustrates the situation a task on core \( P_0 \) and \( P_1 \) access a shared resource.

![Figure 9.16: The overhead under MSRP and FSLM](image)

<table>
<thead>
<tr>
<th>Term</th>
<th>Calculation</th>
<th>Measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSRP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Request resource A</td>
<td>1399+234</td>
<td>( \Delta_s )</td>
</tr>
<tr>
<td>Access granted B</td>
<td>1534+234</td>
<td>( \Delta_s + \Delta_n + (m-1) \times \Delta_n )</td>
</tr>
<tr>
<td>Release resource C</td>
<td>2349+2090</td>
<td>( \Delta_s + \Delta_n + \Delta_m )</td>
</tr>
<tr>
<td>FSLP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Request resource A</td>
<td>1399+234</td>
<td>( \Delta_s )</td>
</tr>
<tr>
<td>Access granted B</td>
<td>1534+234</td>
<td>( \Delta_s + \Delta_n + (m-1) \times \Delta_n )</td>
</tr>
<tr>
<td>Release resource C</td>
<td>2349+2090</td>
<td>( \Delta_s + \Delta_n + \Delta_m )</td>
</tr>
</tbody>
</table>

Table 9.3: The overhead calculated by adding separate measured terms and by measuring the complete execution

Note that \( \Delta_M \) actually takes place after the sending of the interrupt, which means that Figure 9.16 is pessimistic.
Chapter 10

Conclusions

"Time waits for no one."

"Everything should be made as simple as possible, but not simpler."

Real-time systems are characterized by computational activities with timing constraints that must be met in order to achieve the desired behavior. Several algorithms and methodologies have been proposed in the literature to improve the predictability and schedulability of real-time systems. In real-time systems analysis, we want to capture time aspects of the computing system. To create a model of the system, it is necessary to make an abstraction of the reality. Too much detail and the model becomes cluttered and difficult. However, a too simple model does not represent the reality.

The purpose of the thesis is to identify the overheads that have not been considered in the model of preemptable spin-locks in [2] and identify which types of overhead are more significant based on measurement results. For this purpose, we have implemented the preemptable spin-lock model of [2] in ERIKA OS. Towards implementing this model, we needed to change the spinning technique to be based on interrupt notification rather than polling of a global variable. A task might get preempted during spinning and needs to be notified when it is granted access to the resource. We have changed the non-preemptive spin lock implementation of ERIKA OS to an interrupt-based technique.

The analysis suggested that the flexible spin-lock protocol would dominate MSRP/HP. The measured overhead of queue accesses and context switches are substantial. MSRP/HP works in practice quite reasonable due to the lack of overhead. Still, the flexible spin-lock protocol can probably schedule a set of tasks that could otherwise not be scheduled.

Instead of comparing the CP and $\tilde{C}P$ with MSRP it can be compared with FMLP, since it is a more similar protocol that sometimes spins and sometimes suspends. The FMLP protocol has as benefit that it applies spin based or suspension based blocking depending on the length of the critical sections. The cost of the context switch overhead should be saved by reduced time spend spinning. To be efficient the duration of the blocking should take longer than the switching time. Using suspension based blocking if the resource access time is long makes the blocked time vs switching time ratio is favourable. Probably its possible to combine FMLP by Brandenburg et al. in [16] with CP or $\tilde{C}P$ by Sara et al. in [3]. To have short resource access done by HP and apply CP or $\tilde{C}P$ on access of long resources. Additionally, instead of a spin priority per task it may be possible to have a spin priority per resource instead.
Bibliography


Appendix A

Final measurements

To come up with the measurement data used in the paper we used tightly coupled memory in the Quartus hardware designer. The benefit of tightly coupled memory is that a core is connected by faster interconnects to its memory. We used tightly coupled memory for the instruction memory and the data memory. Note that tightly coupled memory is core specific thus not shared.

![Figure A.1: The tightly coupled data and instruction memory interconnects](image1)

Figure A.1 illustrates what the interconnections are between the tightly coupled memory and the core. Both data and the instruction the tightly coupled memory can be found as “System contents window → memories and memory controllers → On-Chip → On Chip Memory (RAM or ROM)”.

![Figure A.2: The properties of the core, the cache and memory interface](image2)

Figure A.2 illustrates the settings of the core under the Cache and Memory Interface tab. By setting the memory as tightly coupled additional connections on the component will appear. Select 1 port for both instruction and master. We chose to use an instruction cache size of 4kbyte and data cache of 2 kbytes with a line size of 4 bytes.

Figure A.3a illustrates the configuration of the instruction memory. Select dual-port access. Read during write mode is DONT_CARE. Block type is Auto. Initialize memory content is checked. We chose for
APPENDIX A. FINAL MEASUREMENTS

(a) The properties of the instruction memory

6144 Bytes. The more space is available on chip the larger this can be set. Figure A.3b illustrates the configuration of the instruction memory. Do not select dual-port access. Block type is Auto. Initialize memory content is checked. We chose for 2048 Bytes. The more space is available on chip the larger this can be set. The same procedure is repeated for each of the cores used in the set-up.

Figure A.3 illustrates the events that we measured. The overhead A denotes the time it takes to request the resource. Overhead B denotes the time it takes to lock the resource. Finally C denotes the time to release the resource.
APPENDIX A. FINAL MEASUREMENTS

In the implementation of MSRP takes the same time for executing A, B and C invariant of the scenario. While the implementation of FSLM contained some scenario specific core that is executed depending on the availability of the resource and whether the task acquiring the resource is still the executing task.

Figure A.4 illustrates the obtained measurement results. The values 79, 146 and 94 is the time it takes due to accessing the resource queue. Note that in MSRP the resource queue is only accessed upon request, while in FSLM it is upon request and release. The values 538 and 560 is the time it takes due to sending and receiving the remote notification. This overhead only occurs in FSLM. The value 700 is the time it takes to perform a context switch.

The measurement data is acquired by measuring parts of the code in a specific scenario around 1000 times. It is possible to construct an overhead $C_4$ in a scenario where the the RN needs to be communicated and that the task of which the resource becomes available is not the executing task. The overhead would become 366+94+560+700.

What can be noticed is that the MRSP and FSLM differ due to the RN and the additional context switching. Requesting, acquiring and releasing takes 433 cycles for MSRP and 3433 cycles. Thus in case the resource access time is larger than 3000 cycles it becomes beneficial to use MSRP. In case the access time is less than the MSRP should be used. For further research it is probably possible to make sure that the overhead of RN and additional context switches only take place in case the performance improves. By using notification via data instead of interrupts in case the requesting task is still the executing task. Only using FSLM for long resources and MSRP for short resources.

<table>
<thead>
<tr>
<th></th>
<th>MSRP</th>
<th>FSLM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request</td>
<td>A 160+79</td>
<td>A 189+146</td>
</tr>
<tr>
<td>Access</td>
<td>B 18</td>
<td>B$_1$ 127+538</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B$_2$ 140+538+700</td>
</tr>
<tr>
<td>Release</td>
<td>C 255</td>
<td>C$_1$ 322+94+560</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C$_2$ 255+94</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C$_3$ 366+94+700</td>
</tr>
</tbody>
</table>

Figure A.3: The measurements performed on MSRP and FSLM

Figure A.4: The measurements results

Thursday 30th June, 2016 18:15

114 Performance of resource access protocols
Appendix B

Memory organization

We want to create a multiprocessor systems as is shown in Figure B.1. We want to run the software for more than one processor out of the same physical memory device. Software for each processor must be located in its own unique region of memory [41], but those regions are allowed to reside in the same physical memory device. For instance, imagine a twoprocessor system where both processors run out of SDRAM. The software for the first processor requires 128Kbytes of program memory, and the software for the second processor requires 64Kbytes. The first processor could use the region between 0x0 and 0x1FFFF in SDRAM as its program space, and the second processor could use the region between 0x20000 and 0x2FFFF.

In Figure B.2 is shown what addresses are used. Its possible to directly write to such an address. For instance the address 0x016000 is in the on-chip memory. By using a int * p = 0x16000; *p=1; we can put the value 1 on that place in the memory. As such we can be sure to read or write from a memory item shared between cores.
In Figure B.3 it is shown how the components comprise the system in memory, and consists of instructions and data. The stack works like a stack, thus its possible to push and pop items from the stack. In our case the items are program instructions. A processor consists of few pipeline stages, for instance:

1. Instruction and cache registers to temporarily store the instruction and data.
2. Arithmetic units like adder and ALU, to perform the operation.
3. Local memory to read or write the data.

The registers contain for instance 32 entries depending on the chip. That could be 16 instructions and 16 data items. The program that is currently being executed builds a sequence of instructions that it is going to execute. Not all of those instructions fit into registers and thus are stored in memory. That datastructure is called the stack. Since the sequence of instructions grows, for instance an if statement results that additional instructions need to happen, that information is then pushed on the stack.

The heap is memory set aside for dynamic allocation. Unlike the stack, there is no enforced pattern to the allocation and deallocation of blocks from the heap; you can allocate a block at any time and free it at any time.

The rwdata contains the data-structures that can be read and written by the program, for instance int i=0; The rodata contains the data-structures that can only be read by the program, for example #define max=30; The text is the actual program, it contains all the instructions the program needs to perform. Its the c-code after its compiled and transformed to asm-instructions.
B.1 Avalon_bus

In case of the Altera FPGA the processor fetch its data and instructions by use of the Avalon_bus. In Figure B.4 the topology of the bus is shown.

In Nios-based systems [42], the Avalon bus connects the Nios processor(s) and other Avalon peripherals via active logic and interconnects inside an Altera® programmable logic device. The system does not have shared bus lines like traditional microprocessor-based systems. Instead, each masterslave pair has a dedicated connection between them. When a peripheral must accept data from multiple sources, such as a Nios processor that receives data from multiple memory devices, multiplexers (not tri-stages) feed the appropriate signal into the peripheral. If a master never needs access to a particular slave, a connection between the two is not generated, saving hardware resources.

![Figure B.4: Overview of bus topology.](image)

Because master and slave peripherals are connected with dedicated paths, multiple masters can be active at the same time and can simultaneously transfer data to their slaves. This simultaneous multi-master architecture offers great throughput performance advantages compared to a traditional, shared bus architecture. Master peripherals do not have to wait to access a target slave peripheral, as long as another master does not access the same slave at the same time. Unlike a shared bus, a simultaneous multi-master architecture with two masters offers up to twice the throughput; with three masters, it offers up to three times the throughput. The throughput improvement depends on how often all three masters are active simultaneously. A simultaneous multi-master system still requires arbitration, but only when two masters contend for the same slave. This arbitration is called slave-side arbitration, because it is implemented at the point where two (or more) masters connect to a single slave. For Nios-based systems using the Avalon bus, the SOPC Builder implements slave-side arbitration entirely inside the Avalon bus module. Every slave peripheral that can be accessed by multiple masters has an arbitrator. The situation of slave arbitration is depicted in Figure B.5.
APPENDIX B. MEMORY ORGANIZATION

The following Figure B.5 shows several slave read transfers [43]. The slave is pipelined with variable latency. In this figure, the slave can accept a maximum of two pending transfers. The slave uses waitrequest to avoid overrunning this maximum.

The numbers in this timing diagram, mark the following transitions:
1. The master asserts address and read, initiating a read transfer.
2. The slave captures addr1.
3. The slave captures addr2.
4. The slave asserts waitrequest because it has accepted a maximum of two pending reads, causing the third transfer to stall.
5. The slave asserts data1, the response to addr1. It deasserts waitrequest.
6. The slave captures addr3.
7. The slave captures addr4. The interconnect captures data2.
8. The slave drives readdatavalid and readdata in response to the third read transfer.
9. The slave captures addr5. The interconnect captures data3.
10. The interconnect captures data4.
11. The slave drives data5 and asserts readdatavalid completing the data phase for the final pending read transfer.

If the slave cannot handle a write transfer while it is processing pending read transfers, the slave must assert its waitrequest and stall the write operation until the pending read transfers have completed. The Avalon-MM specification does not define the value of readdata in the event that a slave accepts a write transfer to the same address as a currently pending read transfer.
B.2 SDRAM

The Nios core is connected to the Avalon bus see Figure B.4. The data request is send over the interconnect to the SDRAM controller. The slave side has arbitration to determine which master it’s serving. The SDRAM controller generates the logic signals needed for communication with the SDRAM. Analog drivers amplify the signal to output it of chip. And the signals arrive at the chip shown in Figure B.7.

![Figure B.7: The architecture of the SD-ram memory.](image1)

The data storage area is divided into several banks, allowing the chip to work on several memory access commands at a time. In Figure B.8 is shown how the banks are implemented.

![Figure B.8: Each bank is like a matrix of rows and columns.](image2)
APPENDIX B. MEMORY ORGANIZATION

A Bank active
Before executing a read or write operation \[44\], the corresponding bank and the row address must be activated by the bank active (ACT) command. An interval of tRCD is required between the bank active command input and the following read/write command input.

B Read with auto-precharge
In this operation, since pre-charge is automatically performed after completing a read operation, a pre-charge command need not be executed after each read operation. The command executed for the same bank after the execution of this command must be the bank active (ACT) command. The next ACT command can be issued at the later time of either tRP after internal pre-charge or tRC after the previous ACT.

C SDRAM time
These timings need to be considered regarding the response time of the SDRAM memory \[45\]:
1. CAS Latency (tCL) - This is the most important memory timing. CAS stands for Column Address Strobe. If a row has already been selected, it tells us how many clock cycles we’ll have to wait for a result (after sending a column address to the RAM controller).
2. Row Address (RAS) to Column Address (CAS) Delay (tRCD) - Once we send the memory controller a row address, we’ll have to wait this many cycles before accessing one of the row’s columns. So, if a row hasn’t been selected, this means we’ll have to wait tRCD + tCL cycles to get our result from the RAM.
3. Row Precharge Time (tRP) - If we already have a row selected, we’ll have to wait this number of cycles before selecting a different row. This means it will take tRP + tRCD + tCL cycles to access the data in a different row.
4. Row Active Time (tRAS) - This is the minimum number of cycles that a row has to be active for to ensure we’ll have enough time to access the information that’s in it. This usually needs to be greater than or equal to the sum of the previous three latencies (tRAS = tCL + tRCD + tRP).

D Variable read or write times.
What makes the read or write times to SDRAM variable. The SDRAM has an refresh cycle, which makes sure there is enough charge in the circuit available to retain its data. Depending on the moment of reading or writing the recharge might take place resulting in an additional delay. The data is split up in four banks, as can be seen in Figure B.7. Those banks need to be activated to be able to access them. Switching from bank takes time. Also switching from reading to writing takes time. Pipelined reading of the same bank is most efficient. For instance the instructions that need to be fetched from a different bank can delay the writing to SDRAM. As such the time a write to SDRAM takes is dependant on how efficient the processor stores the instructions in its registers or needs to fetch them.

SDRAM Time of read to read fast Time of write to write fast Time of read to write slow, the system needs to switch.
The memory controller can use two different policies to manage the row-buffer: close page that precharges the row after every access and open page that leaves the row in the row-buffer open for possible future accesses to the same row.
The measurement was done by measuring the access time of a variable, And reading or writing to the same variable by other cores.
Situation of 0 cores. The core needs to charge the row, and column.
Situation 1,2,3 cores. The row and column might be pre-charged, allowing for faster access.
Conclusion. 1. The faster access times while having more cores can be explained by the consecutive writing to the same row and column. 2. The measurements should be conducted differently by performing alternating read and write instructions. Additionally the access should be done to different rows and columns to illustrate the effect of contention.
B.3 FPGA architecture

The FPGA is a reconfigurable chip. This means that the hardware can be programmed to contain a certain design. The benefit of an FPGA is that it is very flexible. Not only the software can be changed also the hardware itself can be reprogrammed in field.

To get a better understanding of the FPGA it is useful to make a comparison between three types of chips: ASIC, CPU and FPGA. An ASIC chip is specific for a certain task, very efficient, cheap and fast. But the ASIC can only do the job it is designed to do. The functionality of the chip is hardwired in silicon. There is no overhead or software that is processed instruction by instruction. The input of the chip is a digital signal that goes through the logic and an output is produced. An example could be bitwise adding. A CPU Central Processing Unit and is more general purpose, the chip is not made for a specific task but it can do a variation of things. The chip reads instructions and executes them, the work it needs to perform is described in software. The CPU can also bitwise add bits but is a lot slower than an ASIC and it can only perform few tasks in parallel. The benefit of the CPU over the ASIC is that it can also do word processing or streaming video. The FPGA is some where in the middle of those two. It is possible for an FPGA to put part of its functionality in hardware and part in software. The penalty is that it is slower than an ASIC and has less abstraction than a CPU.

![Figure B.9: Architecture FPGA](image)

In Figure B.9a is shown that the FPGA consists of memory and logic. The logic tiles can be programmed by the Quartus hardware programmer. The software ends up in the embedded RAM (in case of onchip memory usage). In Figure B.9b is shown that the programmable matrix consists of programmable interconnects (yellow in Figure B.9a) and programmable logic blocks (green in Figure B.9a). The chip on the DE0 board contains 15,408 Logic blocks and 504K total RAM bits divided over 56 memory blocks.

![Figure B.10: Architecture FPGA](image)

In Figure B.10a is shown what the actual floor plan is of the cyclone chip. The configurable logic blocks are clustered per 16 logic elements as can be seen in Figure B.10b. The smallest unit of logic is the logic element LE and is shown in Figure B.11a. The LE contains a four-input look-up table (LUT), which can implement any function of four variables.
Both the register and LUT can be programmed and determine the function of the LE. The routing to outside the LE can be done by using the multiplexers shown in Figure B.11a and are used for communication with the local, row, and column interconnect. LEs in a LAB have register chain outputs, which allows registers in the same LAB to cascade together. The memory blocks are dual ported as is shown in Figure B.11b which means that it is possible for 2 entities to read or write at the same time.

The interconnects [46] can be programmed to connect the Logic elements. In Figure B.12 is shown how the LEs are connected by interconnects.
In Figure B.13a is shown that the interconnect is like a crossbar. The connections are made by means of switches. It is possible to isolate a part of the design by not making any interconnections. In Figure B.13b the schematic representation is given of a multi-core design. The interconnect is not by means of a bus.

Crossbar switches \cite{47} provide a many-to-many communications mechanism for chips or systems with multiple masters and multiple slaves. Unlike a bus or network, crossbars and switch fabrics support multiple simultaneous transactions. This offers improvements in bandwidth, except in the case where only one master wants to conduct a transaction at a time. In that case, a conventional bus or network would work just as well. In a multi-core situation multiple masters might initiate transactions simultaneously. The crossbar can perform multiple communications at the same time while a bus might add a delay.

In Figure B.14a is shown that the Nios II softcore needs a part of the total LEs, 1200 for an economical core. The Nios II core is an IP block available to program in the hardware. The NIOS II core is a softcore processor comparable with a RISC. The routing and placement phase of the Quatus program figures out where the hardware is actually configured. Since the IP-block is a standardised block all the device drivers and HAL libraries become available to be used in software. In Figure B.14b can be seen that the "User program" is build on top of the provided layers. In our case this "User program" consists of the ERIKA kernel layer and the API-layer.
Appendix C

User manual

Here we present the user manual for programming the hardware and software onto the embedded platform. The user manual consist of creating a 4 core demo setup. The hardware used is the DE0 board from Terasic. First a hardware design is generated as is shown in Figure C.1. Second the software is created and build using the ERIKA kernel.

!!Do NOT use any spaces in filenames, project names and paths during installation, creation of projects and storing files!!
Order information
Board: DE0 from Terasic
Normal price: $119 + some import taxes
Academic price: $81 + some import taxes
Cable: IDE (Parallel ATA connector)

C.1 Installation

Go to: https://www.altera.com/downloads/download-center.html
Download:
- Quartus II Web Edition Software v8.1
- Nios II Embedded design suite Windows Vista (32-bit), and Windows XP (32-bit)

The required programs are shown in Figure C.2.
You will need to register to start the download, yet it is free to do so. Note that if you have a popup blocker the files might be found in the right hand corner as a blocked popup.

Install the downloaded files.
In your download folder you can find 81_quartus_free.exe and install it. Also install 81_nios2eds_windows.exe.

Download:
- INSTALL_nios2.1.4.2.2 INSTALL_nios2.1.4.2.2
- EMF 2.0.3
- GEF SDK 3.0.1
- 090504_evidence_ee.zip
- rtd_gpl_nios2.142.20090511_1832.zip

Locate the target directories.
These directories are located, on a typical installation in:
- Eclipse: C:\altera\81\nios2eds\bin\eclipse
- Components: C:\altera\81\nios2eds\components

Please note that:
1. You need to have a file system which is not read only to copy the files in the right places.
2. When extracting the files, first extract them in a new folder. Then, copy and paste the files "MANUALLY" into the file paths which are given.

**Step 1:** Installing the eclipse plugins

Now you are ready to install the Eclipse plugins. The Eclipse Plugins are a set of files that have to be copied inside the Eclipse directory in addition to the standard files provided by Altera in the Nios II IDE. You need to copy the contents of the following zip files under the Eclipse directory:

- emf-sdo-xsd-SDK-2.0.3.zip
- GEF-SDK-3.0.1.zip
- rtd GPL nios2_142_20090511_1832.zip

To be able to copy the files, first uncompress the files. The zip files all contain an "eclipse" directory. Inside that directory, there are other directories named "plugins" and "features". Please copy all the files in the plugins folder of the zip files into the plugins folder of eclipse. "C:\altera\81\nios2eds\bin\eclipse\plugins".

Copy all the files in the zip files within the folder features to the folder called features in your Eclipse directory. "C:\altera\81\nios2eds\bin\eclipse\features".

**Step 2:** Installing the software components.

You are now ready to install ERIKA Enterprise inside the components directory. To do that, uncompress the file 090504_evidence_ee.zip. The file contains a directory "evidence_ee", which has to be copied inside the components directory. "C:\altera\81\nios2eds\components".

**Step 3:** Checking if RT-Druid has been installed Nios II EDS

You are now ready to launch the Nios II IDE. If the Eclipse plugins are correctly installed, you will notice the availability of a new project of type "RT-druid Oil and C/C++ Project". Where you can find it in the Nios II IDE (when you start it up from all programs menu). Under the menu File → new → other → RT_druid_oil and c/c++ project. This step is shown in Figure C.3.

![Figure C.3: Check whether the plugins are installed correctly](image-url)
A Installing the USB blaster driver

Connect the DE0 FPGA embedded development board to the mains power supply. The power connector is shown as number 1 in Figure C.4. To be able to communicate with a PC we need to connect the USB cable with connector 2 as is shown in Figure C.4. The board can be turned on by pushing the button shown by number 3 in Figure C.4. The initial program stored in the chip is a testing program, the LED’s on the board should start to emit light.

It is possible to use the board without the connection to the power supply. In which case the board is powered by the USB port of the PC. For our application the board does not require a lot of energy and it will not be any problem.

The driver of the USB-blaster is not a standard driver of Windows. We need to manually install the driver. To install the driver we should open the device manager (it is part of the control panel). Go to “start” and type “Device Manager” as is shown in Figure C.5.

A new window will appear called “Device Manager” as shown in Figure C.6. Search for the “Altera USB-Blaster” or “Unknown Device” in the list of devices. Right click on the “Altera USB-Blaster” and select “Update Driver Software” as is shown in Figure C.6.
A new window will appear called "Update Driver Software - Altera USB-Blaster" as shown in C.7. Select "Browse my computer for driver software".

The driver files are located in the installation folder of Quartus. Use the following path "C:\altera\81\quartus\drivers\usb-blaster". Check "Include subfolders" as shown in Figure C.8. Select "Next".
In case you would use browse to select the location make sure to select the parent directory called "usb-blaster". Do NOT select x32 or x64, as is shown in Figure C.9.

![Figure C.9: Update device driver](image)

Probably a new window will appear called "Windows Security" as shown in Figure C.10. Select "Install this driver software anyway". It is very well possible that additional protection software prevents the installation. In which case it might be necessary to make an exception rule for the driver.

![Figure C.10: Update device driver](image)

B In case the driver does not work properly after installation

Uninstall the driver via the device manager. Next run: "regedit.exe". Search for the remaining items in the registry containing the word "blaster". Remove the entry from the registry as shown in Figure C.11.

![Figure C.11: Update device driver](image)
C.2 Creating the hardware design

Start the program.

Open Quartus from "Start → All Programs → Altera → Quartus II web edition" as shown in Figure C.12.

Create a new project by doing as follows:
Click on "Open File → New..."
As shown in Figure C.13.

A new window will appear called "New" as is shown in Figure C.14 Select "New Quartus II Project".
Select "OK".
As shown in Figure C.14.
A new window called "New Project Wizard: Introduction" will appear. As shown in Figure C.15a. Select "Next".

![New project wizard](image1.png) ![Project name](image2.png)

**Figure C.15: Create a new project**

A new window called "New Project Wizard: Directory, Name, Top-Level Entity [page 1 of 5] as shown in Figure C.15b.

Select a working directory by the button "..." Navigate to a place where you want to have your project stored.

"What is the name of this project?"
Here you are allowed to create a name for the project. It is important to use the same name in the SOPC builder later thus be consistent. For example fill in "HelloWorld" as shown in Figure C.15b.

"The name of the top-level design entity for this project?"
For example fill in "HelloWorld" as shown in Figure C.15b.

Select "Next".

It is possible to choose three different path locations. The first path is the location of all the files that will be generated by Quartus (mostly Verilog or VHDL). The second path it the place where the .qpf file will be stored which is the Quartus project file. The third path is the top level design location. The top level design is the file that contains the building blocks. The top level design is created within the SOPC builder. It is required that the file you create with the SOPC builder has the same name as will be provided in this third field. It is possible to provide different file locations for all three paths if the programmer wants to use the same top level design in multiple project files. For instance if the user would use the same design in different FPGA’s.

Default it is convenient to have the same path for all the files that belong to the same project. When the complete path is given in the first field and no path is specified (only the file name) in the second and third than all files end up in the same location.
A window will appear "New Project Wizard: Add Files [page 2 of 5]" as shown in Figure C.16. Select "Next".

A new window will appear "New Project Wizard: Family & Device Settings [page 3 of 5]" as shown in Figure C.17.

In the Device family drop down menu select "Cyclone III"
In the list of "Available devices" select "EP3C16F484C6"
As shown in Figure C.17.
Select "Next."

Note that it is possible to reduce the amount of results given in the list of "Available devices" by selecting a package for example "FGBA (Fine Pitch Ball Grid Array)" The Pin count of the chip we use is 484 pins. And the speed grade is the quality of the chip. Note that the speed 6,7 and 8 are exactly the same chip except that the production of it was less or more successful. After production of a batch the chips are all tested and categorized in grade 6,7,8. Depending on the number off production errors the speed decreases.
A new window will appear "New Project Wizard: EDA Tool Settings [page 4 of 5]" Figure C.18.

![Figure C.18: Select the EP3C16F484C6](image)

Select "Finish".

Now open the SOPC builder. In Figure C.19 is shown how to open the SOPC builder.

![Figure C.19: How to open the SOPC builder within Quartus](image)

A new window appears "Create New System" as shown in Figure C.20.

![Figure C.20: Provide a name for the SOPC design](image)

It is important to provide the same name as was chosen for the top level design in previous steps in the "System Name" field. In this example we provide "HelloWorld". Select "OK".
A Adding 4 Nios core’s to the design

The hardware design we create with this manual is a case example with the specifications as shown in Figure C.1. To make that specific configuration we are adding 4 cores.

Repeat the following steps 4 times

We want to add a Nios II processor to the system as is shown in Figure C.21.

Select "Nios II processor" from the system component list on the left of the screen as is shown in Figure C.22.
Select "Add".

A new window will appear "Nios II Processor cpu_0" as shown in Figure C.22
Select "Nios II/e" from the "Select a Nios II core" bar.
Select "Finish".

Figure C.21: Add a core to the design

Figure C.22: Add a core to the design
B Adding the SDRAM (external memory) to the design

We want to add the SDRAM controller as shown in Figure C.23. To do this select the "Memories and Memory Controllers" in the system components list in the left of the screen as shown in Figure C.23. Select "SDRAM Controller" from the list. Select "Add".

![Figure C.23: Add a SDRAM controller to the design](image)

A new window will appear called "SDRAM Controller - sdram_0" as is shown in Figure C.24. In the dropdown menu called "Presets" select "Custom". In the dropdown menu called "Data width" select "16" bits. Select "Finish".

![Figure C.24: Add a SDRAM controller to the design](image)
The design should now look as is shown in Figure C.25. It is possible to change the names of the components in the design. The names of the components are also used in the .oil configuration file and the software in the API-layer. It is recommended to use the same names for the demo project created by this manual. In case you choose to use different names for components note that the names also need to be changed in the software and configuration files.

To change the name of the component right click on the component to reveal additional options as shown in Figure C.25. Select "Rename" from the list. Change the name to "sdram".

Figure C.25: How to rename components in the design.
C Adding the internal memory to the design

Now we want to add the internal memory to the design. To do this select the "Memories and Memory Controllers" in the system components list in the left of the screen as shown in Figure C.26.
Select "On-Chip Memory" from the list.
Select "On-Chip Memory (RAM or ROM)" from the list.
Select "Add".

Figure C.26: Add a internal RAM to the design

A new window called "On-Chip Memory (RAM or ROM) - onchip_memory2.0" will appear. In order to increase the total memory to 48 kByte as is shown in Figure C.27.
Select "32" in "Data width" and "48" in "Total memory size" fields. Make sure that the metric is select as "kBytes" as is shown in Figure C.27.
Select "Finish".

It is possible to increase the size with some bytes, depending on the amount of components the program will be able to route it or not.

Figure C.27: Add a internal RAM to the design
D Adding the PLL to the design

Now we need to add a PLL to make the clock domain for the external memory.

To do this select the "PLL" in the system components list in the left of the screen as shown Figure C.28. Select the "PLL" from the list. Select "Add".

![Figure C.28: Add a PLL step 1](image)

A new window will appear called "PLL - pll0" as is shown in Figure C.29. To initialise the PLL we make use of the MegaWizard.

Select "Launch Altera’s AltPLL MegaWizard"

![Figure C.29: Add a PLL step 2](image)

A new window will appear called "MegaWizard Plug-In Manager [page 1 of 12]" as is shown in Figure C.30. From the wizard we only need to change the Output clocks.

Select "Output Clocks".
A new window will appear called "MegaWizard Plug-In Manager [page 6 of 12]" as is shown in Figure C.31.
The first output clock needs a frequency of 50 MHz.
Check "Use this clock".
Check "Enter output clock frequency".
Change clock frequency to "50" MHz.
Select "Next".
APPENDIX C. USER MANUAL

Figure C.32: Add a PLL step 5

A new window will appear called "MegaWizard Plug-In Manager [page 7 of 12]" as is shown in Figure C.32. The second output clock needs a frequency of 50 MHz with a phase shift of -60 degrees. Check "Use this clock" Check "Enter output clock frequency". Change clock frequency to "50" MHz. Change clock phase shift to "-60" deg. Select "Finish".

Figure C.33: Add a PLL step 6

A new window will appear called "MegaWizard Plug-In Manager [page 12 of 12] Summary" as is shown in Figure C.33.
It is possible that the waveform is checked when the summary window appears. Any additional files can be used for simulation or timing analysis. We don’t simulate the design and thus do not need any additional files. Note that it is not wrong to select them.

Select "Finish".

We need to close the PLL component Figure C.34.
Select "Finish".

We rename the PLL to "pll" by right click on the PLL component and selecting "Rename" as was shown in Figure C.25.

**E Adding a mutex to the design**

We need a mutex to allow for exclusive access. To do this select "Peripherals" in the system components list in the left of the screen as shown in Figure C.35.
Select "Multiprocessor Coordination" from the list.
Select "Mutex" from the list.
Select "Add".

A new window will appear called "Mutex - mutex_0" as is shown in Figure C.36.
Core 0 needs to start everything up, to do so it will own the mutex initially.
Change initial value to "0x1".
Select "Finish".
F The inter core interrupt communication overview

Each core should be able to send an interrupt to each other core. In Figure C.37 is shown how the core’s are connected. Each core has a physical I/O input pin connected from the FPGA to the Nios core’s. There is an array of 4 I/O output pins. Those output pins are shared between the 4 cores. Thus all cores are able to output on each of the four outputs. Externally we connect the pins by wires.

Note that this means that we have 4 input pins of width 1 that are specific to a core. We have 1 output array or width 4 that is shared between all 4 cores.

G Adding the inter core interrupt communication inputs for the 4 cores

Repeat the next steps 4 times

We need to create the 4 connections for interrupts between cores. To do this select "Peripherals" in the system components list in the left of the screen as shown in Figure C.38.
Select "Multiprocessor Peripherals" from the list.
Select "PIO (Parallel I/O)" from the list.
Select "Add".

Figure C.38: Add an inter core interrupt port step 1
A new window will appear called "PIO (Parallel I/O) - pio_0" as is shown in Figure C.39.
Each input should get a width of 1.
Change Width to "1".
Check "Input ports only".
Select "Next".

![Figure C.39: Add an inter core interrupt port step 2](image)

The triggers should all happen on the same edge and cause an interrupt routine to start as is shown in Figure C.40.
Check "Synchronously capture".
Check "Rising edge".
Check "Generate IRQ".
Check "Edge".
Select "Finish".

![Figure C.40: Add an inter core interrupt port step 3](image)

We rename the just created PIO to "ipic_input_0 ... ipic_input_3" respectively by right click on the PIO component and selecting "Rename" as was shown in Figure C.23.
Adding the inter core interrupt communication output for the 4 cores

We need to create the inter core interrupt outputs for the 4 cores. To do this select "Peripherals" in the system components list in the left of the screen as shown in Figure C.41.

Select "Multiprocessor Peripherals" from the list.
Select "PIO (Parallel I/O)" from the list.
Select "Add".

A new window will appear called "PIO (Parallel I/O) - pio_4" as is shown in Figure C.42.

The inter core output should be one output array of size 4.
Change Width to "4".
Select "Output posts only".
Select "Finish".

We rename the just created PIO to "ipic_output" by right click on the PIO component and selecting "Rename" as was shown in Figure C.25.
Adding the 1 button input connections for each of the 4 cores

Repeat the next steps 4 times

Additionally we want to connect a button as an input to each of the 4 cores. To do this select "Peripherals" in the system components list in the left of the screen as shown in Figure C.38.
Select "Multiprocessor Peripherals" from the list.
Select "PIO (Parallel I/O)" from the list.
Select "Add".

A new window will appear called "PIO (Parallel I/O) - pio_0" as is shown in Figure C.39.
Each input should get a width of 1.
Change Width to "1".
Check "Input ports only".
Select "Next".

The triggers should all happen on the same edge and cause an interrupt routine to start as is shown in Figure C.40.
Check "Synchronously capture".
Check "Rising edge".
Check "Generate IRQ".
Check "Edge".
Select "Finish".

We rename the just created PIO to "led_pio_0 ... button_pio_3" respectively by right click on the PIO component and selecting "Rename" as was shown in Figure C.25.

Adding the 2 LED output connections for each of the 4 cores

Repeat the next steps 4 times

We want to connect 2 LED’s to each core. To do this select "Peripherals" in the system components list in the left of the screen as shown in Figure C.41.
Select "Multiprocessor Peripherals" from the list.
Select "PIO (Parallel I/O)" from the list.
Select "Add".

A new window will appear called "PIO (Parallel I/O) - pio_4" as is shown in Figure C.42.
The inter core output should be one output array of size 4.
Change Width to "4".
Select "Output posts only".
Select "Finish".

We rename the just created PIO to "led_pio_0 ... led_pio_3" respectively by right click on the PIO component and selecting "Rename" as was shown in Figure C.25.
APPENDIX C. USER MANUAL

K Adjusting the properties of the Nios cores

Repeat the next steps 4 times

We need to change the properties of the processors. To do this double click on the Nios II processor component called "core_0...core_3". A new window will appear called "Nios II Processor - cpu_0" as shown in Figure C.43.

Select in the dropdown menu "Memory" either the "sdram" or the "onchip memory". The Reset Vector and Exception Vector determine the place in the memory where the project code is stored. In case we choose the onchip memory we should place all the reset vectors with a distance of 0x3000 which is 12kByte per core and makes 48kByte total. The available memory is 48kByte which makes a perfect distribution of the memory. The reset vector addresses become 0x0, 0x3000, 0x6000 and 0x9000 respectively. The exception vectors should be the reset vector +20, thus 0x20, 0x3020, 0x6020 and 0x9020. In case the SDRAM memory is chosen we have 8MByte to work with instead of 32kByte thus more space can be allocated for each core to work with.

Select Reset Vector : "onchip_memory2_0".
Select Exception Vector : "onchip_memory2_0".
Select Reset Offset: {0x0, 0x3000, 0x6000, 0x9000}.
Select Exception Offset: {0x20, 0x3020, 0x6020, 0x9020}.
Thus core 0 is 0x0 and core 1 is 0x3000.

We recommend to use the SDRAM and select:
Select Reset Vector : "SDRAM".
Select Exception Vector : "SDRAM".
Select Reset Offset: {0x0, 0x40000, 0x80000, 0x120000}.
Select Exception Offset: {0x20, 0x40020, 0x80020, 0x120020}.

Select "Advanced Features" from the top menu in the same window.

Select "Advanced Features" from the top menu in the same window.

We need to change the CPUID control register, Figure C.44.
Check "Assign CPUID control register value manually".
Assign each core its CPUID, core0 =0, core 1=1, core2=2 and core3=3.
Select "finish".
Repeat this for each of the 4 cores.

L Reordering of the components

It is convenient to change the order the components appear in. We need to rename the components as is shown in Figure C.45. The reordering is not mandatory. We choose to put all components that will be connected to all cpu’s on the top. All components related to a specific core are grouped together. Moving a component up or down in the list can be done by selecting the component and clicking on "Move Up" or "Move Down".
M  Optional adding of the uart

The UART is a hardware device that translates data between parallel and serial forms. Adding the UART allows for printing text to a terminal that listens on the receiving port (the pc used to program the board). It is convenient to have a UART available in the design. The FPGA its program is uploaded to the board over the JTAG protocol. This protocol allows for programming the chip, communication like UART, testing and debugging the chip (with correct debugger).

To add the UART to the design select "Interface Protocols" in the system components list in the left of the screen as shown in Figure C.46.
Select "Serial".
Select "JTAG UART".
Select "Add".

A new window will appear called "JTAG UART - jtag_uart_0" as is shown in Figure C.47.
Select "Finish".

The uart should be connected to only one of the cores. It is very possible to connect to all 4 cores and create a ssh connection to all of those terminals. (This is tested, https://www.youtube.com/watch?v=O54sIjSjo60 gives a nice instruction video on how to achieve this). The experience is that the UART buffer can give runtime errors (something with buffer size). Or even the compiling and programming can give errors. Therefore we advice to use only 1 UART over JTAG connected to only one core.
N Optional adding of performance counter

The performance counter allows for measuring the time parts of the code take. The precision of the counter is a single clock tick and is therefore more precise than any timer available. This component may take a lot of the available logic block (not memory blocks) and interconnects of the FPGA. The FPGA we use has sufficient logic blocks to allow the use of the "Performance Counter unit".

To add the counter to the design select "Peripherals" in the system components list in the left of the screen as shown in Figure C.48.

Select "Debug and Performance".
Select "Performance Counter Unit".
Select "Add".

![Figure C.48: Adding the precision counter (used for measurements)](image)

A new window will appear called "Performance Counter Unit - performance_counter_0" as is shown in Figure C.49.

![Figure C.49: Adding the precision counter (used for measurements)](image)

The code to be used within the NIOS II IDE, the software designer not the Quartus hardware designer. We already provide it here in Listing C.1 in case you would want to add this optional component.

```c
#include "altera_avalon_performance_counter.h"
PERF_RESET(PERFORMANCE_COUNTER_0_BASE);
PERF_START_MEASURING(PERFORMANCE_COUNTER_0_BASE);
PERF_BEGIN(PERFORMANCE_COUNTER_0_BASE,1);
//here code we want to measure
PERF_END(PERFORMANCE_COUNTER_0_BASE,1);
PERF_STOP_MEASURING(PERFORMANCE_COUNTER_0_BASE);
int time = perf_get_section_time((void *)PERFORMANCE_COUNTER_0_BASE, 1);
alt_printf("time=%x!
",time);
```

Listing C.1: The code to be used in software (NOT in Quartus hardware)
O Connecting of components

On the right hand side of Figure C.51 (on the next page) are all the connections shown (this is how the end result should look like). Make sure to get the same connections by clicking on the interconnects. In Table C.2 a more detailed overview is given.

<table>
<thead>
<tr>
<th>Instruction 0</th>
<th>Data 0</th>
<th>Instruction 1</th>
<th>Data 1</th>
<th>Instruction 2</th>
<th>Data 2</th>
<th>Instruction 3</th>
<th>Data 3</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>SDRAM</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>Mutex</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>tpc_output</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>Onchip_memory</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Cpu_0</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Ledpio_0</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Buttonpio_0</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Ipic_input_0</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>Cpu_1</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Ledpio_1</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Buttonpio_1</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Ipic_input_1</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>Cpu_2</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Ledpio_2</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Buttonpio_2</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Ipic_input_2</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>Cpu_3</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Ledpio_3</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Buttonpio_3</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Ipic_input_3</td>
</tr>
</tbody>
</table>

Table C.2: More clear view on how the components are connected inside the SOPC builder

Figure C.50: To connect the cores reduce the number of details visible
To reduce the number of connections visible minimize all components except core0 by clicking on the - sign as shown in Figure C.50. Then continue and make the connections for core 2, 3, and 4. The connections should be as follows. Each core’s instructions master should go to the SDRAM, onchip_memory and the jtag_debug_module, on this bus the instructions are communicated. The datamaster should be connected to all components with a connection to that specific core. Connect datamaster to all the common components (memory, pll, mutex and ipic_output). Connect datamaster to all core specific components (led_pio, button_pio and ipic_pio).

P Overview of the hardware design created with the SOPC builder

![Figure C.51: Final overview](image)

Performance of resource access protocols 151 Thursday 30th June, 2016 18:15
Q Summary of the properties of building blocks

• Nios
  – Version: Economical
  – Reset vector: Sdram
  – Exception vector: Sdram
  – Offset: dependant on core*
  – CPUID register: dependant on core**
  – Jtag module: (optionally enabled on one of the cores)

• PIO Button
  – Width 1
  – Input ports only
  – edge capture register: Rising edge
  – Interrupt: Generate IRQ, Edge

• PIO Led
  – Width 2
  – Output ports only

• PIO IPIC input
  – Width 1
  – Input ports only
  – Edge capture register: Rising edge
  – Interrupt: Generate IRQ, Edge

• PIO IPIC output
  – Width 4
  – Output ports only

• Mutex
  – Initial value 0x1
  – Initial owner 0x0

• SdramController
  – Presets Custom
  – Data width 16 bits
  – Chip select 1
  – Banks 4
  – Row 12
  – Column 8

• PLL
  – Input freq 50MHz
  – Clk 0 out 50MHz 0 phase shift
  – Clk 1 out 50MHz -60 degrees phase shift

* The reset vector needs to be placed somewhere in the memory. To do this we need to specify an actual address where the program will start reading instructions when the reset takes place. This address needs to be different for each core. We choose the starting addresses as follows: Take the whole available memory, or a larger enough section to at least fit OS code 4 times (without printfs +/- 6 kByte per core). Divide that memory space such that each restart vector is equally separated. In our case we took 0x0, 0x40.000, 0x80.000 and 0x120.000. The exception vector is the number found by thus 0x20, 0x40.020, 0x80.020, 0x120.020.

** The CPUID is the same number as the number of the core, thus core_0 has CPUID 0 and core_3 CPUID 3.
R Connect the components to the clock signal

We added the PLL to the design. The PLL was correctly configured by use of the Altera’s ALTPLL megawizard. We created 2 additional clock signals. These clock signals are required to make the SDRAM work on the same clock domain as the clock in the chip. The SDRAM is off chip and the propagation of the clock through the wires and transistors takes some time. We therefore apply a phase shift of -60 degrees.

After the additional clock signals are created, they are also shown in the clock settings window of the SOPC builder. Figure C.52 shows the clock signals.

![Figure C.52: The clock signals](image1)

![Figure C.53: Change the clock signal for a specific component](image2)

First we rename the "Clk0" to "pll_cpu" and the "Clk1" to "pll_sdram". We need to connect all the components to the correct clock. Connecting a component to a clock can be done by left clicking on the clock name in the clock column as is shown in Figure C.53. The "CLK" clock goes to the "PLL" such that it can create the additional clock signals (pll_cpu and pll_sdram). The "pll_sdram" will be connected to the SDRAM controller. All the other components are connected to the pll_cpu. Figure C.51 shows how the updated clock should look like.

S Auto configure the design.

![Figure C.54: Auto assign](image3)

Under system we need to auto assign the base address and auto assign the IRQs as is shown in Figure C.54.

![Figure C.55: Save the design](image4)

To save press "ctrl+s" or select "File" and then "save" as is shown in Figure C.55.

![Figure C.56: Generate the design](image5)

Finally we need to Generate the design as is shown in Figure C.56. Off course, the design should be error free before continuing.
At this point we completed the system design in the SOPC builder and return to the Quartus platform. First we need to check whether all the files are added: Go to the project navigator and right click on Files, "add/ remove files in project". In Figure C.57 is shown how to add files to the design.

![Figure C.57: Adding files to the design](image)

A new window appear called "Settings - projectname" as is shown in Figure C.58.

![Figure C.58: Adding files to the design](image)

Select "Add all" to add all the files generated with the SOPC builder to the project. Select "Ok" as is shown in Figure C.58. All the added files should now appear in the project navigator.
U Analyze and synthesis

The design we have created up till now contains components that are internally connected. The connections between the components and the board need to be made. First analyze and synthesis the project. On the left side of the Quartus II main screen (not the SOPC builder) is a window called “Task” as is shown in Figure C.59. Double click on : “Analysis & Synthesis”.

![Figure C.59: To compile and synthesise the design](image)

Open the assignment editor as is shown in Figure C.60. In the top of the screen there is a bar, select "Assignments" from that bar. Select "Assignment Editor" form the drop down menu.

![Figure C.60: How to open the assignment editor](image)

A new window appears called ”Assignment Editor” as is shown in Figure C.61. Here we can specify the locations of the pins. For all the components we use in this demo manual the pin locations are given in Figure C.61.

To connect additional components we need to find out which pins are connected to which parts of the board. The wiring of the board is described in the User manual of the FPGA board. The manual can be found at: [http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=165&No=364&PartNo=4](http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=165&No=364&PartNo=4) (search for “DE0 Terasic manual” in Google).
<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Assignment Name</th>
<th>Value</th>
<th>Enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2U_E2</td>
<td>Location</td>
<td>P2U_E2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_C4</td>
<td>Location</td>
<td>P2U_C4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_A5</td>
<td>Location</td>
<td>P2U_A5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_B6</td>
<td>Location</td>
<td>P2U_B6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_A6</td>
<td>Location</td>
<td>P2U_A6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_C7</td>
<td>Location</td>
<td>P2U_C7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_B7</td>
<td>Location</td>
<td>P2U_B7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_B4</td>
<td>Location</td>
<td>P2U_B4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_A7</td>
<td>Location</td>
<td>P2U_A7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_B5</td>
<td>Location</td>
<td>P2U_B5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_A4</td>
<td>Location</td>
<td>P2U_A4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_G8</td>
<td>Location</td>
<td>P2U_G8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_E5</td>
<td>Location</td>
<td>P2U_E5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_E7</td>
<td>Location</td>
<td>P2U_E7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_F9</td>
<td>Location</td>
<td>P2U_F9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_G9</td>
<td>Location</td>
<td>P2U_G9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_M9</td>
<td>Location</td>
<td>P2U_M9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_A9</td>
<td>Location</td>
<td>P2U_A9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_C10</td>
<td>Location</td>
<td>P2U_C10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_B10</td>
<td>Location</td>
<td>P2U_B10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_A10</td>
<td>Location</td>
<td>P2U_A10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_E10</td>
<td>Location</td>
<td>P2U_E10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_F10</td>
<td>Location</td>
<td>P2U_F10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_E7</td>
<td>Location</td>
<td>P2U_E7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_B8</td>
<td>Location</td>
<td>P2U_B8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_E7</td>
<td>Location</td>
<td>P2U_E7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_D6</td>
<td>Location</td>
<td>P2U_D6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_H2</td>
<td>Location</td>
<td>P2U_H2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_C3</td>
<td>Location</td>
<td>P2U_C3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_J1</td>
<td>Location</td>
<td>P2U_J1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_J2</td>
<td>Location</td>
<td>P2U_J2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_J3</td>
<td>Location</td>
<td>P2U_J3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_M1</td>
<td>Location</td>
<td>P2U_M1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_F2</td>
<td>Location</td>
<td>P2U_F2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_E1</td>
<td>Location</td>
<td>P2U_E1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_C1</td>
<td>Location</td>
<td>P2U_C1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_C2</td>
<td>Location</td>
<td>P2U_C2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_M1</td>
<td>Location</td>
<td>P2U_M1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_U1</td>
<td>Location</td>
<td>P2U_U1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_R0</td>
<td>Location</td>
<td>P2U_R0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_V1</td>
<td>Location</td>
<td>P2U_V1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_W0</td>
<td>Location</td>
<td>P2U_W0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_Y0</td>
<td>Location</td>
<td>P2U_Y0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_F3</td>
<td>Location</td>
<td>P2U_F3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2U_M6</td>
<td>Location</td>
<td>P2U_M6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure C.61: The pin editor
V Compiling the design

Now we want to compile the design. To compile the design double click on "Compile Design" as is shown in Figure C.62.

![Figure C.62: To compile the design](image)

W Program the hardware

At this point there should be no errors. It does not matter that there are warnings. Select the symbol in the right corner of the screen as shown in Figure C.62. The button is called "Program" and is next to the SOPC builder button.

A new window will appear called "Quatus II" as shown in Figure C.63. Select "OK".

![Figure C.63: Select OK](image)
A new window will appear called "Quartus II Programmer - Chain1.cdf" as shown in Figure C.64. Often the right file is already selected by the Quartus program. In case this has not happened by default we can add the file by selecting "Add File".

![Figure C.64: The Quartus programmer](image)

A new window will appear called "Select Programming File" as is shown in Figure C.65. Select "Open".

![Figure C.65: Add the .sof file to be programmed](image)

It is possible to start programming, by selecting "Start" as is shown in Figure C.64. The hardware description is now programmed in the FPGA, a new window will appear called "OpenCore PLus Status" as is shown in Figure C.66.

![Figure C.66: Add the .sof file to be programmed](image)
C.3 Creation of software

Start the "Nios II IDE" software as shown in Figure C.67.

![Figure C.67: Start the Nios software as an administrator](image)

Create a new project by selecting:
"File → New... → other",
as is shown in Figure C.68.

A new window will appear called "New" as is shown in Figure C.68.
Select "Evidence → RT-Druid Oil and C/C++ Project".
Select "Next".

![Figure C.68: Create a new RT-Druid Oil and C/C++ Project](image)
Create a project using one of these templates: Take "nios2 → OO examples → Task demo" As shown in Figure C.69. Select "Next".

To specify the name for the software project put a name in the field "project name" as shown in Figure: C.70. Select "Next".

A Optionally let already the project files appear

{To let the project files appear we need to build and clean the project, as is shown in Figure C.71. The build will give some errors, we will solve those later.

}
B Create the system library files

In the Quartus hardware designer we created the hardware description for the FPGA. Now in the software platform we create the program that runs on the hardware description. To be able to use the hardware we require a library. In the hardware designer each component did get an address. It is possible to use the components in the design by writing instructions to those addresses. By creating a library the writing to address instruction becomes available in the form of an easier to use function. For instance printf(); is actually writing to the addresses of the UART buffer of the JTAG component.

Repeat the following steps for the number of cores in the design (in our case 4 times)

Create a new system library by selecting "File → New → Nios II System Library" as shown in Figure C.72.

![Create a new system library](image1)

Figure C.72: Create a new system library

A new window appears called "New Project" as is shown in Figure C.72. Provide a name for the system library. For instance Name = "Hello_World_syslib_core0" It is convenient to provide a name with the core number in it to be able to distinguish between cores. Select the.ptf file you created with the SOPC builder for instance "C:\HelloWorld.ptf". Select a different "CPU" for each core in the design. Select "Finish". Build the created system library by right clicking on the system library and select "Build Project" from the drop down menu as shown in Figure C.73b.

![Create a new system library](image2)

(a) Create a new system library

![Build the system library](image3)

(b) Build the system library

Figure C.73: Create and build the system library
C Change the OIL file

We have a hardware description for 4 cores, the demo project we opened as a basis contains only 2 cores. We suggest you first make the 2 core version running on the board. If that works extend the design to 4 cores.

The ERIKA OS kernel layer uses a number of predefined C files as a part of the kernel. To make the OS transparent from the hardware some of the hardware dependant files are generated by a tool called RT-druid. The RT-druid is built in the Nios II IDE by the plugins we installed earlier. The RT-druid reads the OIL file and generates the required kernel C files. To be able to generate those files within the oil file the hardware description and system libraries need to be linked. We need to make changes to the oil file. On the left side of the Nios II IDE is the project explorer as is shown in Figure C.74.

Open the "config.oil" file by double clicking on it.

![Figure C.74: Change to conf.oil file to point to the system library and the hardware project file](image)

We need to specify where the hardware description can be found and where the system libraries are located. On line 79 in Figure C.74 the NIOS_PTF_FILE is requested. The default location is the actual location of the .ptf file. We need to change this to the place where the .ptf file is stored. For instance "C:/Users/s138226/Desktop/HelloWorld/HelloWorld.ptf".

**Note that the slashes are forward slashes and NOT backslashes which is different from the windows file system.**

We also need to point to the system libraries we created in a previous step (bottom previous page). Change the SYSTEM_LIBRARY_NAME line 88 and 99 in Figure C.74 to the name of the created system libraries. For instance "Hello_World_syslib_core0" and "Hello_World_syslib_core1".

We need to specify the SYSTEM_LIBRARY_PATH on line 89 and 100 in Figure C.74. For instance "C:/Users/s138226/Desktop/new_workspace/Hello_World_syslib_core0".
We can check whether the file paths are correctly specified in previous step. To do so clean the project twice as is shown in Figure C.75. The Console should show "Good, its all ok.". If this is not the case check whether the file names are correct, there are no spaces in the paths and check whether the libraries are build. To check again after changes: build the project and clean it twice again.

Now it is time to build the project (not the system libraries). By right clicking on the project and selecting "Build project" from the drop down menu, as is shown in Figure C.73b. The program should give an error as is shown in Figure C.76. In case a different error is shown resolve that first before continuing.

Performance of resource access protocols 163 Thursday 30th June, 2016 18:15
D  Explanation of a common error

In Figure C.76 on the left in the project explorer are the system library of core 0 "testmanual_C0_syslib" shown and the main project "testmanual_C10". The error exists in the main file for core 0 called cpu0_main.c. The main file uses a define "LED_PIO_BASE" shown with the green squares in Figure C.76. When we builded the system library in this case testmanual_C0_syslib we created amongst others the file system.h. The file system.h contains a lot of the definitions of the hardware. For instance system.h contains LED_PIO_0_BASE 0x00011030. In the SOPC builder we created a LED output called LED_PIO_0 which had a address of 0x11030. By writing to this address we can turn that output pin on or off. Note that the name is "LED_PIO_0" and not "LED_PIO". Thus the main program uses a definition which is not defined. We can solve this by using "LED_PIO_0" instead of "LED_PIO".

Showing this error is illustrative of how the components in the SOPC builder are connected to the system library and can be used in the main files. Instead of changing all the errors manually we provide the code for the first application. First we provide the code for a typical design running ERIKA OS with a task per core, activation of interrupt routines, sharing of data protected my get and release resource primitives.

E  If using SDRAM memory use option A, if using onchip memory use option B

Note that the memory size of the on chip memory is too small to run 4 times the ERIKA OS including tasks for each core. We therefore provide code for a minimal design running ERIKA OS without any tasks on 2 cores (to be extended to 4) which can fit inside the on-chip memory. The reason for providing the minimal design is to be able to run on faster memory which has more predictable timing characteristics. The reason for providing the typical design is to show a run including tasks and resource access. The code of the 2 cores including tasks and resources can also run on a 2 core design in on-chip memory where core0 would have a reset vector of 0x0 and core1 of 0x6000 and none other cores exist.

F  Option A implementation of ERIKA with tasks and resource access

(Fits in 4-core design with SDRAM or 2-core design with on-chip memory)

Here we provide the code of cpu0_main.c, cpu1_main.c, task0.c and task2.c that can be directly copied to the project. Select the code of "cpu0_main.c" in Listing C.2 press "ctrl+c" for copying, select the code of "cpu0_main.c" in the Nios II IDE and press "ctrl+v" (for pasting) followed by "ctrl+s" (for saving). Do the same for cpu1_main.c, task0.c and task2.c.

```c
#include <stdio.h>
#include <system.h>
#include <altera_avalon_pio_regs.h>
#include "ee.h"

int a_local_counter;

static void handle_button_interrupts_cpu0(void* context, alt_u32 id){
  static int led_counter=0;
  ActivateTask(task0);
  ActivateTask(task2);
  led_counter = (led_counter+1)%4;
  IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_0_BASE, led_counter);
  IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_0_BASE, 0);
}

static void init_button_pio_0(){
  IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTON_PIO_0_BASE, 0x3);
  IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_0_BASE, 0x0);
  alt_irq_register( BUTTON_PIO_0_IRQ, NULL, handle_button_interrupts_cpu0 );
}

int main(){
  init_button_pio_0();
  StartOS(OSDEFAULTAPPMODE);  
  while(1){; ;}
  return 0;
}
```

Listing C.2: The code of cpu0_main.c
#include <stdio.h>
#include <system.h>
#include "ee.h"

int main()
{
    StartOS(OSDEFAULTAPPMODE);
    while(1){;;}
    return 0;
}

Listing C.3: The code of cpu1_main.c

#include <stdio.h>
#include "ee.h"
#include "shareddata.h"
#include "resourcedata.h"

TASK(task0){
    GetResource(mutex);
    mutex_mydata++;
    ReleaseResource(mutex);
    TerminateTask();
}

Listing C.4: The code of task0.c

#include <stdio.h>
#include "ee.h"
#include "resourcedata.h"
#include <altera_avalon_pio_regs.h>

TASK(task2){
    int localvariable;
    GetResource(mutex);
    localvariable=mutex_mydata;
    ReleaseResource(mutex);
    IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_1_BASE, localvariable);
    TerminateTask();
}

Listing C.5: The code of task2.c

G Working of the code

The way this program works is that both core 0 and 1 start the ERIKA OS. After startup core 1 will end in a while(1) loop. Core 0 will also end in a while(1) loop. Except on startup Core 0 registered an interrupt routine that starts when a button is pushed. The interrupt routine will activate a task on core0 and core1. Task0 is mapped to core 0 in the config.oil file and task2 to core1. Both task 0 and task 1 as shown in Listing C.4 and C.5 make use of the get and release resource function. Thus we test whether the resource access protocol works. After the resource is released by task0, task2 can use the resource and turns on the LED's. We should see the LED's change anytime we push the button.
APPENDIX C. USER MANUAL

H Option B implementation of ERIKA without tasks and NO resource access
(Fits in 4-core design with on-chip memory)

Here we provide the code of cpu0 main.c, cpu1 main.c that can be directly copied to the project. Select the code of “cpu0 main.c” in Listing C.6 press “crtl+c” (for copying), select the code of “cpu0 main.c” in the Nios II IDE and press “crtl+v” (for pasting) followed by “crtl+s” (for saving). Do the same for cpu1 main.c.

### Listing C.6: The code of cpu0 main.c

```c
#include <stdio.h>
#include <system.h>
#include <altera_avalon_pio_regs.h>
#include "ee.h"

int led_counter=0;
static void toggle(void){
    led_counter = (led_counter+1)%4;
    IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_0_BASE , led_counter);
}

int main(){
    StartOS(OSDEFAULTAPPMODE);
    int i=0;
    while(1){
        toggle();
        for(i=0; i<500000; i++){;;}
    }
    return 0;
}
```

### Listing C.7: The code of cpu1 main.c

```c
#include <stdio.h>
#include <system.h>
#include <altera_avalon_pio_regs.h>
#include "ee.h"

int led_counter=0;
static void toggle(void){
    led_counter = (led_counter+1)%4;
    IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_1_BASE , led_counter);
}

int main(){
    StartOS(OSDEFAULTAPPMODE);
    int i=0;
    while(1){
        toggle();
        for(i=0; i<500000; i++){;;}
    }
    return 0;
}
```

I Working of the code

After starting the OS on core0 and core1 the execution of the code ends up in the while(1) loop as is shown in Listing C.6 and C.7. The whole loop consist of waiting and a toggle function that turns the LED’s on and off. What will be visible by running this code is that each core controls 2 LED’s (thus 4 in total) and turns those LED’s on and off repeatedly.
J  Checking the properties of the system library and the main project

Before we build the project we should check if all the properties are set correctly. In Figure C.77a is shown how to open the properties window of a project. First we check the properties of the system library. Right click on the project in the project explorer on the left side of the screen. Select "Properties" from the dropdown menu.

(a) Open the properties of the system library
(b) The system library should be build as debug

Figure C.77: Check the properties of the system library

A new window will appear called "Properties of [system library name]" as is shown in Figure C.77b. Select the C/C++ Build tab. Select as "Configuration" "Debug". Select "Apply".

Select the "System Library" tab as is shown in Figure C.78.

Select the "JTAG uart" tab to the design its possible to select the jtag_uart from the dropdown menu of "stdout", "stderr" and "stdin". This is required to use the printf instruction. In this demo we do not use the printf and thus we can leave the setting to NULL. In case the uart is used, it should only be set in the system library of core 0. Unless the uart is connect to multiple core’s which we do not recommend.
On the right hand side of Figure C.78 it is possible to specify the place where the different parts of the software are stored in memory. Make sure that you use the same setting here as was done in the SOPC builder. Either put the "Program memory (.text)" in onchip_memory as shown in Figure C.78 and the "Reset Vector" in the SOPC builder in onchip_memory as shown in Figure C.43. Or put both "Program memory (.text)" and "Reset Vector" in SDRAM. The properties as shown in the left bottom of Figure C.79b are set as we preferred. Select "OK".

In case a project is copied, renamed or moved it is required to check whether the build directory of the main project is still correct. To do so open the properties of the project by right clicking on the project and selecting "Properties" from the dropdown menu as is shown in Figure C.79a. Open the "C/C++ Make Project" tab as shown in Figure C.79b. Check whether the Build directory is set to \[project name\]\Debug.

If any changes were applied we need to rebuild the system libraries and the main project. Right click on the project and select "Build" from the dropdown menu as was shown in Figure C.73b. Rebuild first the system libraries of core 0 and 1, after that rebuild the main project. In case you used the on-chip memory the design will not fit, the error shown in Figure C.80 will occur. To solve this open the config.oil file, by double clicking on the file in the project explorer on the left side of the screen. Select the code that is related task 0, 1 and 2 and delete it. Save the file by pressing "ctrl+s". Rebuild the project again. The program should now be build without errors.

(a) Open the properties of the main project
(b) The build directory should be correct

Figure C.79: Check the properties of the system library

Figure C.80: The memory is too small to contain a design with more than one task per core

Thursday 30th June, 2016 18:15 168 Performance of resource access protocols
APPENDIX C. USER MANUAL

K Creation of the launch configuration

Note that the board should still contain the hardware description that was put on the board with the Quartus program. If this is not the case reprogram the hardware as was described earlier as "Program the hardware" Subsection W.

Now the project should have been build without any errors. The next step is to create the run configuration for each of the cores. We need to repeat the next step for the number of cores we want to run at the same time, in our case two.

Select the "Run" button a triangle symbol as shown in Figure C.81a. A new window will appear called "Run" as is shown in Figure C.81b. Select "Nios II Hardware". Select "New launch configuration" the square with a plus sign as is shown in Figure C.81b.

In the Figure C.82 is shown what properties we need to specify for each core. We need to provide a name, preferable with the core number in it for instance "Hello_World_C0" To select a project you can push the "Browse" button. All open projects are visible in here. When the project was build the compiler created a binary file for each of the cores. The binary file has an .elf extension. Select a "Nios II ELF Executable" by use of the "Search" button. Be sure to select the right core.

![Figure C.81: Create the Run configuration](image)

(a) ![Create, manage, and run configurations](image)

(b)

In the Figure C.82 is shown what properties we need to specify for each core. We need to provide a name, preferable with the core number in it for instance "Hello_World_C0" To select a project you can push the "Browse" button. All open projects are visible in here. When the project was build the compiler created a binary file for each of the cores. The binary file has an .elf extension. Select a "Nios II ELF Executable" by use of the "Search" button. Be sure to select the right core.
The "SOPC Builder System PTF File" should contain the file that was created by the SOPC builder. Use the "Browse" button to navigate to the correct .ptf file. Select the correct "CPU" for instance "cpu_0". The "Additional nios2-download arguments" are often correct by default.

In a 2 core design the arguments are (core_0,1), (core_1,0).
In a 4 core design the arguments are (core_0,1), (core_1,2), (core_2,3), (core_3,0).

Select "apply".

L Configuring the preferences for multiple active run

Note that this step needs to be done only once for each pc the software gets installed. After the initial setting of the parameters it will be saved and this step can be skipped. A new window will appear called "Preferences" as is shown in Figure C.84. Select "Nios II" from the list. Check the box called "Allow multiple active run/debug sessions" Select "OK".

Now we need to create the launch configuration for a multi-core setup. Select the "Run" button, the triangle symbol at the top of the screen as was shown in Figure C.81a. The "Run" window will appear as is shown in Figure C.85a.
APPENDIX C. USER MANUAL

Figure C.84: Change the preferences to allow for multiple active runs

M  Create a multi-core launch configuration
To allow the environment to launch multiple cores at the same time we need to adjust the settings. In the top of the screen of the Nios II IDE select "Window" as is shown in Figure C.83. From the dropdown menu select "Preferences".

Figure C.85: Create a multi-core Run configuration

N  The program runs
Assuming the hardware is successfully programmed and the multi-core software is actively running it is possible to test the program. Push the button on the board that is depicted in Figure C.86. The program without tasks and resource sharing should automatically start toggling the LED's.

Figure C.86: Press the button to test the program in case of design with tasks and resource access
Appendix D

Measurement of read and write

We measured the time it takes to read or write to shared memory. Each measurement loops 1000 times and we derived the maximum, average and minimal time. The purpose is to understand how much influence the contention on the bus has on read and write times. The code we measure is "this_is_shared =1;" for writing and "if( this_is_shared ==1) {;;}" for reading. Listing D.1 provides the code used to measure the time a read and write instruction take as a function of cores reading and writing on the bus.

```c
alt_putstr("Measurement 1\n");
for(i=0; i!=1000; i++)
  PERF_RESET(PERFORMANCE_COUNTER_0_BASE);
  PERF_START_MEASURING(PERFORMANCE_COUNTER_0_BASE);
  PERF_BEGIN(PERFORMANCE_COUNTER_0_BASE,1);
  this_is_shared=1;
  PERF_END(PERFORMANCE_COUNTER_0_BASE,1);
  PERF_STOP_MEASURING(PERFORMANCE_COUNTER_0_BASE);
  time2 = perf_get_section_time((void *)PERFORMANCE_COUNTER_0_BASE, 1);
  alt_printf("%x\n", time2);
}

alt_putstr("Measurement 2\n");
for(i=0; i!=1000; i++)
  PERF_RESET(PERFORMANCE_COUNTER_0_BASE);
  PERF_START_MEASURING(PERFORMANCE_COUNTER_0_BASE);
  PERF_BEGIN(PERFORMANCE_COUNTER_0_BASE,1);
  if(this_is_shared==1){;;}
  PERF_END(PERFORMANCE_COUNTER_0_BASE,1);
  PERF_STOP_MEASURING(PERFORMANCE_COUNTER_0_BASE);
  time2 = perf_get_section_time((void *)PERFORMANCE_COUNTER_0_BASE, 1);
  alt_printf("%x\n", time2);
}
```

Listing D.1: The code of core 0, to measure the write and read times

For the initial measurements we only utilized core 0, the other cores where running in a while(1){;} loop. We found that the writing takes less time than the reading. Some inaccuracy which occurs in the measurement is that "if(this_is_shared==1){;;}" not only reads. The function also performs a comparison and a conditional if operation (with maybe branch prediction). Each measurement runs through the for loop 1000 times and gives thus 1000 measurements. The first time the measurement is done the reading or writing time is significantly worse than all additional executions. The reason could be that the instruction fetch takes additional time, while it is stored in an instruction register it does need to be loaded. We want to measure what the influence of contention on the bus is. We therefore disregard the first time and keep the remaining 999 measurements. In D.1 and D.2 the results from the first measurement are shown.

<table>
<thead>
<tr>
<th></th>
<th>Maximum</th>
<th>Minimum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write cycles</td>
<td>42</td>
<td>35</td>
</tr>
<tr>
<td>Read cycles</td>
<td>110</td>
<td>103</td>
</tr>
</tbody>
</table>

Table D.1: Measurement write and read with no contention
First we measure what the effect is of reading the same shared data, Listing D.2.

```c
//core 1..3 read shared memory
while(1){
    if(this_is_shared==1){;;}
}
```

Listing D.2: The code of core $P_1..P_3$, to measure the write and read times

![Figure D.1: The influence of reading while writing](image1)

Second we measure what the effect is of writing the same shared data, Listing D.3. The results are given in Figure D.3 and D.4.

```c
//core 1..3 write shared memory
while(1){
    this_is_shared=2;
}
```

Listing D.3: The code of core $1..3$, to measure the write and read times

![Figure D.2: The influence of reading while reading](image2)

![Figure D.3: The influence of writing while writing](image3)
At last we measure what the effect is of writing data to a large matrix, Listing D.4. The results are given in Figure D.5 and D.6.

```c
//core 1..3 write a matrix into memory
register int a;
register int b;
int c[100][100];

while(1){
    for(b=0; b!=100; b++){
        for(a=0; a!=100; a++){
            c[a][b]=1;
        }
    }
}
```

Listing D.4: The code of core 1..3, to measure the write and read times

We notice that reading the same data has no significant influence. The writing of data has influence on the bus. The maximum time changes by 20% in some measurement. Strange about the behaviour is that more cores using the bus does not always mean more contention.
In Figure D.7, D.8, D.9 and D.10 the measurements of reading and writing while the program code is stored in on-chip memory.

Figure D.7: The time reading takes while 0-3 other core’s read to that shared data

Figure D.8: The time reading takes while 0-3 other core’s write to that shared data

Figure D.9: The time writing takes while 0-3 other core’s reading to that shared data

Figure D.10: The time writing takes while 0-3 other core’s write to that shared data
Appendix E

Code

In Listing E.1 is shown how function names are declared in the ERIKA-kernel. The code often has a general name for a function that can be called to execute the function in this case GetResource(). Elsewhere in the code where the function of GetResource() is described are often multiple versions of the function. Depending on for instance the kernel-type (and often parameters of the OIL file) the function is mapped to an implementation of the function. In Listing E.1 the OO version of GetResource() is the defined implementation. In this document we decided to name some functions (all functions of the kernel layer we use) with the reduced name excluding the letters in front.

```c
//Use in API layer:
GetResource(mutex);

//Code in Kernel layer
#define GetResource EE_oo_GetResource

void EE_oo_GetResource(ResourceType ResID){
    ...
    ...
}
```

Listing E.1: The function names are defined with extended names depending on the kernel-type

<table>
<thead>
<tr>
<th>Extended name</th>
<th>Reduced name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EE_hal_spin_in()</td>
<td>spin_in()</td>
</tr>
<tr>
<td>EE_hal_spin_out()</td>
<td>spin_out()</td>
</tr>
<tr>
<td>EE_rn_execute()</td>
<td>rn_execute()</td>
</tr>
<tr>
<td>EE_rn_handler()</td>
<td>rn_handler()</td>
</tr>
<tr>
<td>EE_rn_send()</td>
<td>rn_send()</td>
</tr>
<tr>
<td>EE_rq_queryfirst()</td>
<td>rq_queryfirst()</td>
</tr>
<tr>
<td>EE_rq2stk_exchange()</td>
<td>rq2stk_exchange()</td>
</tr>
<tr>
<td>EE_rq_insert()</td>
<td>rq_insert()</td>
</tr>
<tr>
<td>EE_OO_ReleaseResource()</td>
<td>ReleaseResource()</td>
</tr>
<tr>
<td>EE_OO_GetResource()</td>
<td>GetResource()</td>
</tr>
<tr>
<td>EE_OO_ActivateTask()</td>
<td>ActivateTask()</td>
</tr>
<tr>
<td>EE_OO_StartOS()</td>
<td>StartOS()</td>
</tr>
</tbody>
</table>

Table E.1: Definition of functions
### APPENDIX E. CODE

#### Listing E.2: The c-code of spin_in(), flexible spin-lock kernel code

```c
EE_hal_spin_in(EE_TYPESPIN m) {
    spin_lock=0;
    EE_altera_mutex_spin_in();
    if(*(EE_UINT32 *)TailQ[m] != 0xa0) {
        *(EE_UINT32 *)TailQ[m] = GlobalTaskID[EE_stkfirst];
        ResourceQ[m][EE_CURRENTCPU] = GlobalTaskID[EE_stkfirst];
        TailQ[m] = (EE_UINT32 *)&ResourceQ[m][EE_CURRENTCPU];
        EE_altera_mutex_spin_out();
    }
    if(spin_lock==1) {
        EE_sys_ceiling |= EE_th_spin_prio[EE_stkfirst];
        EE_sys_ceiling &= ~0x80;
    }
    while(spin_lock != 0);;
}
```

#### Listing E.3: The c-code of spin_out(), flexible spin-lock kernel code

```c
#include "ee_internal.h"
StatusType EE_oo_ReleaseResource(ResourceType ResID) {
    EE_TID rq, current;
    EE_UREG isGlobal;
    register EE_FREG flag;
    External int Task_exe;
    isGlobal = ResID & EE_GLOBAL_MUTEX;
    ResID = ResID & ~EE_GLOBAL_MUTEX;
    if (ResID >= EE_MAX_RESOURCE) { return E_OS_ID;}
    current = EE_stkfirst;
    if (EE_th_ready_prio[current] > EE_resource_ceiling[ResID]) { return E_OS_ACCESS;}
    flag = EE_hal_begin_nested_primitive();
    EE_th_resource_last[current] = EE_resource_stack[EE_th_resource_last[current]];
    EE_resource_locked[ResID] = 0;
    if(isGlobal) EE_hal_spin_out(ResID);
    rq = EE_rq_queryfirst();
    EE_sys_ceiling &= ~0x80;
    EE_sys_ceiling &= ~EE_th_spin_prio[EE_stkfirst];
    EE_sys_ceiling |= EE_th_dispatch_prio[EE_stkfirst];
    if(Task_exe != -1) {
        Task_exe = -1;
        if(EE_th_ready_prio[rq] >= EE_th_ready_prio[EE_stkfirst] || rq == EE_NIL) {
            EE_hal_stkchange(EE_stkfirst);
        }
    }
    if(rq != EE_NIL) {
        if(EE_sys_ceiling < EE_th_ready_prio[rq]) {
            EE_th_status[current] = READY;
            EE_th_status[rq] = READY;
            EE_th_dispatch_prio[rq] = EE_th_dispatch_prio[rq] & ~EE_th_ready2stacked((EE_UREG)rq);
        }
    }
    EE_hal_end_nested_primitive(flag);
    return E_OK;
}
```

#### Listing E.4: The c-code of ReleaseResource(), flexible spin-lock kernel code

Performance of resource access protocols 177 Thursday 30th June, 2016 18:15
APPENDIX E. CODE

Listing E.5: The c-code of rn_execute(), the MSRP/HP implementation on a local spin-lock

```c
void rn_execute(EE_TYPERN rn, EE_UINT8 sw){
    extern int spin_lock;
    if (EE_rn_type[rn][sw] & EE_RN_RESOURCE){
        spin_lock=0;
    }
    return;
}
```

Listing E.6: The c-code of rn_execute(), flexible spin-lock kernel code

```c
void EE_rn_execute(EE_TYPERN rn, EE_UINT8 sw){
    if (EE_rn_type[rn][sw] & 0x40){
        extern int spin_lock;
        extern int Preemption_took_place;
        EE_sys_ceiling|=0x80;
        spin_lock=0;
        if(EE_rn_task[rn]!=EE_stkfirst){
            Preemption_took_place=1;
            EE_hal_stkchange(rn);
        }
        EE_rn_type[rn][sw] &= ~0x40;
    }
}
```

Listing E.7: The c-code of GetResource() functionally unchained in the original and flexible spin-lock protocol

```c
StatusType EE_oo_GetResource(ResourceType ResID){
    register EE_UREG isGlobal;
    register EE_FREG flag;
    isGlobal = ResID & EE_GLOBAL_MUTEX;
    ResID = ResID & ~EE_GLOBAL_MUTEX;
    if (ResID >= EE_MAX_RESOURCE) { return E_OS_ID; }
    if (EE_resource_locked[ResID] || EE_th_ready_prio[EE_stkfirst] > EE_resource_ceiling[ResID]) {return E_OS_ACCESS;}
    flag = EE_hal_begin_nested_primitive();
    EE_resource_stack[ResID] = EE_th_resource_last[EE_stkfirst];
    EE_th_resource_last[EE_stkfirst] = ResID;
    EE_resource_locked[ResID] = 1;
    EE_sys_ceiling |= 0x80;
    if (isGlobal){ EE_hal_spin_in(ResID);}
    EE_hal_end_nested_primitive(flag);
    return E_OK;
}
```