MASTER

Flexible spin-lock model
analysis and implementation

Balasubramanian, S.M.N.

Award date:
2017

Link to publication
Flexible Spin-Lock Model - Analysis and Implementation

Author: Sri Muthu Narayanan
Supervisor: Dr. ir. Reinder J. Bril

A thesis submitted in fulfillment of the requirements for the degree of Master of Science in the Systems Architecture and Networking group
Department of Mathematics and Computer Science

Committee members:
Dr. MSc. Nabi Najafabadi
Dr. MSc. Moris Behnam
Lic. MSc. Sara Afshar

July 17, 2017
The Flexible Spin-Lock Model (FSLM) unifies suspension-based and spin-based resource access protocols (RAP) for partitioned fixed-priority preemptive scheduling (P-FPPS) based real-time multiprocessor platforms. P-FPPS and spin-based RAPs are prescribed for multiprocessor platforms by AUTOSAR [1], the standard for the automotive industry. Recent work [2] has been done in defining the protocol for FSLM and providing a schedulability analysis without accounting for the implementation overheads.

In this thesis, an initial implementation of FSLM[3][4] in the OSEK/VDX-compliant Erika Enterprise RTOS on an Altera Nios II platform using 4 soft-core processors is used as the basis, upon which improvements are made to optimize the performance. The improvements include a faster inter-core communication mechanism and memory optimization by using dual shared stacks per core instead of one stack per task. The thesis also includes recommendations and insights for further optimizing the performance by exploring the low-level hardware locks used by the operating system kernel to enable mutual exclusion.

The improved implementation is then used to characterize the implementation overheads related to resource access. The schedulability analysis of FSLM is extended to include the implementation overheads. The thesis also supplements the analysis with measurement results, enabling an analytical comparison of FSLM with the natively provided multiprocessor stack resource policy (MSRP), which may serve as a guideline for the choice of FSLM or MSRP for a specific application.
Acknowledgements

I would sincerely like to thank my graduation thesis supervisor Dr. ir. Reinder J. Bril at Eindhoven University of Technology (TU/e) for his continuous support throughout the graduation thesis phase. Since the first day, Dr. Bril was always available whenever I had any questions, immediately providing me with appropriate guidance with his vast experience while also ensuring that the thesis remained my own work. He was extremely prompt in providing feedback through the weekly discussions without which this work would have been impossible. It was indeed a pleasure to work under his guidance.

I would also like to thank Dr. Moris Benham and Sara Afshar from Mälardalen University, Sweden for their timely guidance and suggestions for shaping the goals of the thesis. Their feedback during critical phases of the thesis was helpful in spotting potential errors and rectifying them in a timely fashion.

I would like to extend my thanks to Dr. Paolo Gai from Evidence S.r.l., Italy, for providing valuable knowledge transfer regarding the Erika Enterprise RTOS that helped in quick-starting the thesis. I am also thankful to Mr. Richard Verhoeven from the Mathematics and Computer Sciences Department at TU/e for his practical support with the FPGA hardware. A special thanks to Dr. Nabi Najafabadi from TU/e for participating as the external examiner in the graduation committee.

I also would like to acknowledge the work performed by Maikel Verwielen in his Masters thesis, providing detailed documentation regarding the internal workings of the Erika Enterprise resource access functions and user manuals for creating multiprocessor systems on Altera based FPGA platforms. His documentation proved to be very useful in getting acquainted with the hardware and software platforms in a short time period.

Finally, I express my gratitude to my parents and friends for their continuous encouragement and support throughout my Masters degree …

Sri Muthu Narayanan BALASUBRAMANIAN
List of Figures

1.1 Methodology .............................................................. 5

3.1 Illustration of a task spinning while waiting on a global resource [4]. ........ 15
3.2 Nios II embedded system design flow [36]. .................................. 17
3.3 Overview of the hardware system created for the study. ......................... 18
3.4 Development view of Erika [4]. ............................................. 19

4.1 Spinning on a global spin-lock. .............................................. 22
4.2 Spinning on a local spin-lock. ................................................ 24
4.3 Spinning on a local spin-lock. ................................................ 24
4.4 Inter core interrupts hardware connection for Remote Notifications [4]. ...... 26
4.5 Calls graph for the function EE_oo_GetResource(). ......................... 28
4.6 Calls graph for the function EE_oo_ReleaseResource(). .................... 29
4.7 The EE_di_send() function. .................................................. 31
4.8 The EE_di_handler() function. .............................................. 32
4.9 Flow chart of EE_di_execute(). .............................................. 33
4.10 Remote Notification component functions with the difference between DI and RN highlighted. .................................................. 34
4.11 Comparison of execution times of component functions of Remote Notification (RN) and Dedicated Interrupts (DI). ......................... 37

5.1 FSLM single stack evolutions under HP and CP approaches. .................. 41
5.2 Partial overview of RT-Druid automatic code-generation architecture ......... 46
5.3 Example Dual shared stack overview. ......................................... 49

6.1 Context of Flex-Tool in the software build process. ................................ 56
6.2 Sequence diagram for Flex-Tool actions. ..................................... 57
6.3 Screen-shot from Flex-Tool execution. ....................................... 58

7.1 Overhead scenarios in the implementation of MSRP and FSLM w.r.t resource access. 60
7.2 Resource request overheads in FSLM (new implementation). .................. 61
7.3 Resource access overheads in FSLM (new implementation). .................... 62
7.4 Example time-line scenario illustrating blocking context switch .......... 63
7.5 Resource release overheads in FSLM (new implementation). .................... 63
7.6 Resource access overheads in MSRP (native implementation). ................. 64
7.7 Resource release overheads in MSRP (native implementation). ................ 65
7.8 Overview of all overhead components for FSLM. ................................ 66
7.9 Example time-line scenario illustrating Remote Holding Time overheads .... 68

8.1 Performance counters in SOPC design ..................................... 73
8.2 Output of JTAG UART port from CPU 0 on the terminal. ..................... 75
8.3 Structure of the example application used for measurements. .................. 77
8.4 Relative sizes of the overheads in MSRP and FSLM (initial and new) implementa-
tions. ........................................................................ 79
8.5 Predictability of the cumulative overheads in MSRP, FSLM (old and new). .... 80
8.6 Analytical evaluation scenario. ................................................ 81
8.7 The response time of $\tau_{hp}$ as a function of the gcs lengths of lower priority and remote tasks under MSRP, FSLM-CP and FSLM-HP with 2 processors. .... 82
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3</td>
<td>Hardware design specifications.</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>Functional verification of DI mechanism - Test cases.</td>
<td>36</td>
</tr>
<tr>
<td>4.2</td>
<td>Comparison of execution times of component functions of Remote Notification (RN) and Dedicated Interrupts (DI) for 75 samples (25 per core).</td>
<td>36</td>
</tr>
<tr>
<td>4.3</td>
<td>Comparison of DI integration options.</td>
<td>39</td>
</tr>
<tr>
<td>8.1</td>
<td>Comparison of measurement methods.</td>
<td>72</td>
</tr>
<tr>
<td>8.2</td>
<td>Resource usage of performance counters.</td>
<td>73</td>
</tr>
<tr>
<td>8.3</td>
<td>Performance counter macros and functions.</td>
<td>74</td>
</tr>
<tr>
<td>8.4</td>
<td>CPI calculation for context switching overheads</td>
<td>78</td>
</tr>
<tr>
<td>8.5</td>
<td>Overheads in MSRP and FSLM (in terms of number of cycles).</td>
<td>79</td>
</tr>
<tr>
<td>9.1</td>
<td>Execution times for $10^5$ iterations of mutex acquisition and release for CPU 1 and CPU 2 under contention and no contention.</td>
<td>91</td>
</tr>
<tr>
<td>9.2</td>
<td>Execution times for $10^5$ iterations of mutex acquisition and release for CPUs 1, 2 and 3 under contention and no contention.</td>
<td>92</td>
</tr>
</tbody>
</table>
Listings

4.1 RN message types in the original Erika Enterprise. ................................................. 27
4.2 Updated code of EE_oo_GetResource()................................................................. 29
4.3 Updated code of EE_oo_ReleaseResource()............................................................ 30
4.4 Pseudo code for EE_di_send() function................................................................. 31
4.5 C-code for EE_di_send() function............................................................................ 31
4.6 C-code for EE_di_handler() function.......................................................................... 32
4.7 C-code for EE_di_execute() function......................................................................... 32
5.1 Example 2-core OIL file snippet - CPU object and OS object. ................................. 43
5.2 Example OIL file snippet - Task objects with shared and private stacks. .................. 45
5.3 Example auto-generated stack configuration - snippet from eecfg.c ......................... 47
5.4 Example modified dual stack configuration.............................................................. 48
5.5 OIL file snippet showing IRQ_STACK configuration................................................ 49
6.1 Snippet of “extern” configuration parameters from ee_internal.h ......................... 53
6.2 Snippet of example initialization of variables in cpu1_main.c .................................. 54
8.1 Snippet from ee_fslm_measure.h showing the measurement data structures. ............ 74
8.2 Snippet from fslm.h showing the measurement data structures................................. 75
8.3 Snippet from cpu0_main.c showing the initialization of MeasureQ............................ 75
8.4 Snippet of EE_hal_spin_out() showing the code instrumentation for measurement of $OH_{nd}$................................................................. 76
8.5 Snippet from ee_fslm_measure.h showing the measurement macros.......................... 77
9.1 Snippet section guarded by mutex in EE_hal_spin_out() ........................................ 86
9.2 Snippet of example alarm initialization in cpu1_main.c............................................ 89
9.3 Mutex interference measurement code setup snippet.............................................. 90
A.1 Contents specifying pin assignments that must be added to the four_core_demo.v file ......................................................................................................................... 105
B.1 Convention for mentioning the TASK STACK attribute in OIL file ......................... 110
B.2 CPU application file variables.................................................................................... 111
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>EDF</td>
<td>Earliest Deadline First</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out</td>
</tr>
<tr>
<td>FMLP</td>
<td>Flexible Multiprocessor Locking Protocol</td>
</tr>
<tr>
<td>FSLM</td>
<td>Flexible Spin-Lock Model</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>IRQ</td>
<td>Interrupt ReQuest</td>
</tr>
<tr>
<td>ISR</td>
<td>Interrupt Service Routine</td>
</tr>
<tr>
<td>LBL</td>
<td>Local Blocking due to Local resources</td>
</tr>
<tr>
<td>LBG</td>
<td>Local Blocking due to Global resources</td>
</tr>
<tr>
<td>LE</td>
<td>Logic Element</td>
</tr>
<tr>
<td>LED</td>
<td>Light Emitting Diode</td>
</tr>
<tr>
<td>LIFO</td>
<td>Last In First Out</td>
</tr>
<tr>
<td>MPCP</td>
<td>Multiprocessor Priority Ceiling Protocol</td>
</tr>
<tr>
<td>MSRP</td>
<td>Multiprocessor Stack Resource Policy</td>
</tr>
<tr>
<td>OIL</td>
<td>OSEK Imlementation Language</td>
</tr>
<tr>
<td>OSEK</td>
<td>Offene Systeme und deren schnittstellen für die Elektronik in Kraftfahrzeugen (Open Systems and their interfaces for the Electronics in motor vehicles)</td>
</tr>
<tr>
<td>P-FPPS</td>
<td>Partitioned Fixed Priority Preemptive Scheduling</td>
</tr>
<tr>
<td>PCP</td>
<td>Priority Ceiling Protocol</td>
</tr>
<tr>
<td>PIP</td>
<td>Priority Inheritance Protocol</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RAP</td>
<td>Resource Access Protocol</td>
</tr>
<tr>
<td>RM</td>
<td>Rate Monotonic</td>
</tr>
<tr>
<td>RN</td>
<td>Remote Notifications</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real-Time Operating System</td>
</tr>
<tr>
<td>SOC</td>
<td>System On Chip</td>
</tr>
<tr>
<td>SOPC</td>
<td>System On a Programmable Chip</td>
</tr>
<tr>
<td>SRP</td>
<td>Stack Resource Policy</td>
</tr>
<tr>
<td>VDX</td>
<td>Vehicle Distributed eXecutive</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The complexity of embedded systems are growing at a rapid rate, especially in the last three decades. As a direct consequence of Moore’s law, the processors are becoming increasingly powerful causing the competing companies in the international market to pack more functionality in their devices to attract consumers. As a result, we now have small devices controlling seemingly simple tasks such as house heating with the capability of processing orders of magnitude more information than the computer systems used on the Apollo missions that helped humanity reach the moon. As the physical size of a transistor used in a processor started to reach its minimum limit in the scale of nanometers, engineers started moving towards systems with multiple processors rather than trying to fit more processing power into a single processor.

The permeation of multiprocessor platforms in embedded systems has resulted in an increased need for providing efficient resource management and scheduling methods. Real-time embedded systems are highly resource constrained and are often used for safety critical applications demanding time critical operations, thereby requiring resource handling techniques that guarantee meeting the application deadlines with predictable response times. In a typical multi-processor real-time embedded system, the tasks residing on multiple processing cores access shared global resources. Efficient resource sharing protocols are required to co-ordinate the resource access among competing cores without inducing overheads.

In traditional lock-based resource access protocols (RAPs)\(^1\) for real-time multi-processor platforms, a task that is blocked on a global resource either performs a non-preemptive busy wait, i.e. spins, or releases the processor, i.e. suspends.

In case of non-preemptive spinning, the task that is blocked on a resource boosts its priority to the highest priority level on the core and thus becomes non-preemptable. The task registers itself on the resource queue and waits for the resource to become available. For the entire duration that a task is blocked on a global resource in a core, it remains as the active (running) task although no useful work is being done. When the resource finally becomes available, the task acquires the resource and lowers the priority to its original level only after the completion of the resource usage (referred to as the global critical section).

On the other hand, in a suspension based protocol, when a task is blocked on a resource, it relinquishes the processor to other tasks by registering itself on the resource queue and suspending itself. Other tasks on the core are now free to execute while the requesting task is suspended. When the resource becomes available, the requesting task preempts any other task that is running on the core irrespective of its priority level and acquires the resource. It raises its priority to a non-preemptive level and executes the global critical section. The Multiprocessor Stack Resource Policy (MSRP) [5] is an example for a spin-based protocol while the Multiprocessor Priority Ceiling Protocol (MPCP) [6] is an example for a suspension-based protocol.

While it may seem that out of the two above approaches, a suspension based protocol offers high CPU utilization, it must be noted that switching between tasks is not instantaneous. The time taken for performing a context switch between tasks can induce significant overheads in the system. Hence, it is important to carefully consider the effects of the resource access protocol on the worst-case response times of the tasks by accounting for the blocking times and overheads induced by the protocol. A solution that provides a fine balance between the two traditional methods might be desirable.

The Flexible Spin-Lock Model (FSLM) [2] unifies both traditional approaches for partitioned fixed-priority preemptive scheduling (P-FPPS) by introducing a spin-lock priority for each processor, allowing the blocked task to spin at an arbitrary priority level. In partitioned systems, the

---

\(^1\)In this report the terms resource sharing protocol and resource access protocol are used interchangeably
tasks are mapped onto the processors statically (off-line). P-FPPS and spin-based global RAPs are also prescribed for multi-processor platforms by AUTOSAR [1], the standard for the automotive industry.

Now, we can treat non-preemptive spin-based protocols (like MSRP) as using the highest priority (HP) as spin-lock priority whereas suspension based protocols (like MPCP) as using the Lowest Priority (LP) as spin-lock priority. Although FSLM allows for the choice of an arbitrary spin-lock priority, a particular range of spin-lock priorities are of primary interest since it has been found to significantly improve the schedulability. One such spin-lock priority is called the ceiling priority (CP) defined as the highest resource ceiling of global resources on a core. In other words, CP is the priority of the task with the highest priority among all tasks on the core which use a global resource. Initial simulation results based on the developed theory [7] confirm the expectations with respect to improved schedulability under CP when compared to HP. The remainder of the thesis considers spin-lock priorities in the range of [CP, HP] for FSLM. The focus of the thesis is to create an efficient implementation of FSLM and study the runtime overheads induced, characterize and incorporate them in the schedulability analysis.

1.1 Background

The Flexible Spin-Lock Model [2] (FSLM) unifies spin-based and suspension-based resource sharing protocols for partitioned fixed priority preemptive scheduling (P-FPPS) based real-time multiprocessor systems by explicitly identifying the spin-lock priority as a parameter. Analysis for FSLM has been developed, assuming the implementation overhead could be ignored. An initial implementation of FSLM [3][4] in Erika Enterprise [8] has been realized and instantiated on an Altera Nios II platform [9] using four soft-processing cores. The initial implementation was complemented with measurement results. The initial implementation was hampered by the limitations of the FPGA development board, an Altera DE0, and left various topics for further investigation. To gain a better understanding of the implementation overheads, it is desirable to move towards a better development platform that does not limit the experiments that could be performed and has sufficient resources to allow improvements to the initial implementation.

1.2 Problem statement

During the initial implementation, a strong correlation between the memory configuration (off-chip, on-chip, type of memory) and the implementation overhead was noted. Due to the unavailability of sufficient on-chip memory, shared data structures were moved to off-chip memory devices that contributed to increased overheads. Selecting a hardware platform that provides appropriate memory of the right size may therefore strongly influence the overheads.

Second, the re-use of Remote Notification (RN) mechanism for the initial implementation of FSLM turned out to be extremely inefficient. The RN mechanism uses a system of shared memory based data structures and inter-processor hardware interrupt lines to achieve communication between the processors in the system. Other mechanisms, such as dedicated interrupt lines for FSLM, which do not use shared memory, may be more appropriate.

Third, the existing implementation (the Erika Enterprise RTOS itself) only uses a one (low-level) hardware mutex to regulate mutual exclusive access to various shared data structures between processors in the system, which may therefore become a limited resource and a bottleneck.

Fourth, FSLM allows the usage of just two stacks (instead of one stack per task) [7] for spin-lock priorities in [CP, HP], which may significantly reduce the memory requirements of the system. However, the initial implementation uses one stack per task in the system. It may be beneficial to implement the dual shared stack and explore the memory savings offered by such a method.

Finally, the schedulability analysis for FSLM provided in [2] does not take into account the implementation overheads. It is therefore beneficial to identify, measure and characterize the

---

FSLM allows the usage of just two stacks (instead of one stack per task) [7] for spin-lock priorities in [CP, HP], which may significantly reduce the memory requirements of the system. However, the initial implementation uses one stack per task in the system. It may be beneficial to implement the dual shared stack and explore the memory savings offered by such a method.

Finally, the schedulability analysis for FSLM provided in [2] does not take into account the implementation overheads. It is therefore beneficial to identify, measure and characterize the

---

2Erika Enterprise is a free of charge open-source OSEK/VDX Hard Real Time Operating System
3Nios II is a 32-bit soft processor core IP block designed for Altera FPGAs [9]
4In this report, the terms processor and core are used interchangeably
5Remote Notifications are used by the Erika kernel to enable inter-core communication in multiprocessor SOCs. Explained further in Chapter 4.
induced implementation overheads and accommodate them in the schedulability analysis to provide better scheduling estimates.

1.3 Research questions

This section presents a list of research questions that give rise to the primary objectives of the thesis. The analysis of the research questions presented below is used for the identification of the objectives. The research questions are revisited in later chapters of the report where they are answered based on the findings from the thesis.

Question 1 What are the guidelines for choosing FSLM over MSRP for a given application?

While ignoring the runtime overheads, preemptive spin-based protocols such as FSLM have been shown to outperform non-preemptive protocols such as MSRP. It is desirable to know from an implementation perspective if FSLM outperforms MSRP with runtime overheads included. However, from the initial implementation of FSLM [3] [4], it has been identified that significant runtime overheads are encountered as a part of FSLM which are not incurred in MSRP. To answer the question, an efficient implementation of FSLM is required. The overheads incurred in the implementations of FSLM and MSRP must then be measured. The overheads should be incorporated into the schedulability analysis to enable a comparison between MSRP and FSLM from a practical perspective. By choosing appropriate spin-lock priority levels other than HP for FSLM, considering the response time of high priority tasks, FSLM might outperform MSRP when the remote blocking time in MSRP exceeds the cumulative runtime overheads induced in FSLM. For remote blocking times to be sufficiently large, it might help to consider longer global critical section lengths and systems with more processors.

Question 2 Is the dual shared stack implementation in FSLM for spin-lock priorities in [CP, HP) feasible? What are the benefits of such an implementation?

The default tooling provided by Erika Enterprise RTOS does not explicitly allow for the configuration of dual shared stacks. Hence, a new implementation with support for such shared stacks must be created to examine the feasibility and the ease of creation of such a system. Stack space saving in a shared stack context is usually achieved only when multiple tasks occupy the same preemption level, thereby ensuring that such tasks can never occur in the stack simultaneously. However, while dealing with the worst-case scenario, under FSLM, all tasks are assumed to have unique priority levels (preemption level is considered to be equal to the priority). Hence, even with stack sharing, the total memory requirement would remain unchanged. However, since FSLM is a preemptive spin-based protocol, tasks can be interrupted meaning the interrupt requests (IRQs) require their own stack space. Thus memory space required by the IRQs may be minimized by adopting the dual shared stacks implementation.

Question 3 How does a dedicated interrupt based inter-core communication mechanism for notifying resource release between the cores compare to the existing generic shared memory based communication mechanism for FSLM in the range [CP, HP]?

Choosing a specific range of spin-lock priorities in FSLM, i.e., [CP, HP] enables at most one task to be waiting for a global resource in a core at any given time. Thus a simple interrupt based inter-core communication mechanism may be sufficient to notify cores regarding the release of a resource. The initial implementation of FSLM [3] [4] showed that by using the existing generic shared memory based Remote Notification mechanism, significant runtime overheads are incurred. An implementation using dedicated interrupts must therefore be made to show that it is practically feasible for such a system to achieve communication regarding resource release while also reducing the overheads. In addition, for a full fledged kernel, both mechanisms might have to co-exist to offer different types of inter-core communication. The possibilities must be explored on the new implementation.

Question 4 Can the overheads induced due to the usage of low-level hardware locks by the operating system kernel be minimized?

Using the same low-level hardware lock for guarding unrelated shared data in a multiprocessor system may lead to increased overheads as the low-level lock now may become a bottleneck.
The effects of using the same low-level hardware lock for guarding the shared queues of different resources must be measured on the platform to draw further conclusions. By appropriately utilizing the required number of low-level hardware locks for guarding shared data in a system, the RTOS related overheads may be minimized.

### 1.4 Objectives and goals

The objectives of this assignment derived from the research questions are twofold:

(i) Improve the existing implementation of FSLM by reducing the implementation overheads and memory requirements.

(ii) Identify and characterize the overheads in the implementation and incorporate them in the schedulability analysis of FSLM to make the analysis more suited to real-time applications.

A preparation study was performed prior to the master thesis to identify the individual goals of the project and to perform a feasibility study. During the course of the preparation phase, the following process related activities were carried out:

- Selecting and ordering an FPGA board (Altera DE2-115) with more Logic Elements (LEs) and embedded memory in comparison to the Altera DE0 board that was used for the initial implementation [3][4].

- Porting the existing implementation of FSLM to the new Altera DE2-115 board.

The preparation phase resulted in the identification of the following goals for the thesis, which are classified under four categories:

#### Implementation Goals

I.1 Addition of dedicated interrupt line for inter-core communication to replace the existing methods that uses Remote Notification mechanism.

I.2 Implementation of dual shared stacks for FSLM, replacing the one stack per task in the existing implementation.

I.3 Investigate the effects of utilizing dedicated low-level hardware spin-lock per global resource in the internal data structures of the RTOS kernel.

#### Measurement Goals

M.1 Create a setup for performing measurements on the platform.

M.2 Identify and measure the sources of overhead within the context of the resource sharing protocol (FSLM and MSRP).

#### Analysis Goal

A.1 Extend the schedulability analysis of FSLM and MSRP by including the identified implementation overheads.

#### Documentation Goals

D.1 Final thesis report capturing all analysis and activities undertaken as a part of the assignment.

D.2 Provide comprehensive documentation as an internal document detailing all the steps involved in creating the hardware and software design.

D.3 A paper to be submitted at an appropriate outlet.
1.5 Contributions

All the goals of the thesis listed in Section 1.4 were completed. Apart from the identified goals, the following additional contributions were made:

- A utility tool called “Flex-Tool” for automatically configuring the RTOS (Chapter 6) to accommodate per processor spin-lock priorities and dual shared stacks.

- Recommendations for implementing the dedicated interrupts mechanism in the full-featured version of Erika Enterprise RTOS (Chapter 4.6).


A complete list of the outcomes of the thesis is presented in Chapter 10 along with the mapping of the goals with the outcomes.

1.6 Methodology

The work performed in this thesis consists of an implementation part and a research part. The method followed to accomplish each part is detailed below:

Implementation methodology

Since the thesis involves relatively small feature additions to the existing Erika Enterprise software, the V-model for software development, which is an extension of the traditional waterfall model, is employed. The steps recommended by the V-model tailored to the needs of this assignment are listed:

1. The shortcomings of existing implementation were studied to determine the requirements for the improved implementation. The requirements were then formulated.

2. A high level design for the implementation was performed by creating flowcharts and specifying existing invariants and predicates.
3. The features were then implemented adhering to standard software development recommendations.

4. A sanity-check after the completion of implementation was performed to ensure the integrity of the software system.

5. Detailed testing of the implemented feature was then performed by empirically specifying test cases and running tests on the system and also by formally verifying the invariants and predicates.

For parts of the thesis related to measurement of overheads in the implementation, a qualitative approach was followed where the components of the overheads were methodically defined and measured with precision of CPU clock cycles.

**Research methodology**

The research methods employed in the thesis are inspired from the steps undertaken in [12] that were originally recommended in [13].

1. Identify the research questions.

2. Derive the objectives from the research questions.

3. Formulate the goals based on the analysis of the objectives.

4. Propose a solution for each goal.

5. Analyze and evaluate the results.

Repeat steps 3, 4 and 5 until the goals are achieved. Figures 1.1a and 1.1b illustrate the implementation and research methodology used in this thesis work respectively.

**1.7 Reading guide**

This thesis report contains a significant amount of content and hence might cater to different types of audience. A reading guide for specific audience based on each of the two objectives specified in Section 1.4 is presented below:

**Implementation**

Chapter 1: Introduction p. 1  
Chapter 3.3: System Overview p. 13  
Chapter 4: Inter-core Communication p. 21  
Chapter 5: Dual Shared Stacks p. 41  
Chapter 7: Overheads p. 59  
Chapter 8: Measurements and Analysis p. 71  
Chapter 9: Low-level hardware locks p. 85

**FSLM overheads analysis**

Chapter 1: Introduction p. 1  
Chapter 3.2: FSLM Overview p. 14  
Chapter 7: Overheads p. 59  
Chapter 8: Measurements and Analysis p. 71
1.8 Thesis outline

Chapter 1: Introduction
The problem is introduced followed by the research questions, identification of objectives and definition of the goals of the thesis.

Chapter 2: Related Work
An overview of existing work related to the objectives of the thesis is provided in this chapter. The context in which this thesis is complementary to the existing work is also briefly described.

Chapter 3: Project Overview
The real-time scheduling model and associated terms that are used throughout the rest of the thesis are introduced in this chapter. A brief overview of the Flexible Spin-Lock Model and the existing schedulability analysis for FSLM that does not consider the implementation overheads is also presented in this chapter. An overview of the hardware and software systems used for the implementation of FSLM in this thesis is also included.

Chapter 4: Inter-core Communication
Goal I.1 and Question 3
The motivation, design, implementation and verification of the dedicated interrupt based inter-core communication mechanism suitable for FSLM spin-lock approaches in [CP, HP] are presented in this chapter. Measurement results comparing the native RN mechanism in Erika Enterprise with the newly implemented dedicated interrupt based method are also presented.

Chapter 5: Dual Shared Stacks
Goal I.2 and Question 2
The implementation of dual shared stacks for FSLM spin-lock approaches in [CP, HP] for Erika Enterprise is presented in this chapter. The chapter provides a calculation for the savings in stack memory offered by adopting the dual shared stacks over per task private stack.

Chapter 6: FSLM Tool
A brief overview of “Flex-Tool”, a utility tool that was developed as a part of the thesis to automatically configure the RTOS files to accommodate per processor spin-lock priorities and dual shared stacks per core is presented in this chapter.

Chapter 7: Overheads
Goal A.1
The runtime overheads in the implementations of FSLM and MSRP are identified and characterized in this chapter. The chapter presents a generic method for the classification of overheads related to RAPs that can be extended to implementations of other RAPs with relatively small modifications. The identified overheads are then incorporated into the existing schedulability analysis of FSLM and MSRP.

Chapter 8: Measurements and Analysis
Goals M.1, M.2 and Question 1
The measurement setup and results of measuring the overheads in the implementations of FSLM and MSRP are presented. An analytical evaluation based on a scenario is also presented, giving a guideline for the choice of FSLM over MSRP based on certain parameters.

Chapter 9: Low-level Hardware Locks
Goal I.3 and Question 4
The effects of low-level hardware locks in Erika Enterprise on the performance is investigated by performing measurements on the selected platform. Recommendations for minimizing the overheads induced due to the low-level hardware locks are also provided.

Chapter 10: Conclusions
A summary of the outcomes of the thesis and avenues for future work are presented in this chapter.
Chapter 2

Related work

In this chapter, a summary of existing literature relevant to one or more goals of the thesis are presented. The existing work that was studied as a part of literature survey for the thesis have been classified according to their categories and presented below. The topics covered under related work include multiprocessor resource access protocols, previous implementations of resource access protocols, memory utilization in resource constrained systems and topics associated with runtime overheads.

2.1 Multiprocessor resource access protocols

A typical multiprocessor system consists of two types of resources: local and global resources. The resources that are accessed only by tasks on the same processor are called as local resources while global resources are accessed by tasks on two or more processors. Uniprocessor resource access protocols are used for handling local resources. The Priority Ceiling Protocol (PCP) [14] and Stack Resource Policy (SRP) [15] are examples of resource access protocols used for the handling of local resources in a system. PCP uses a static value called priority ceiling for each semaphore guarding a resource. The priority ceiling of the resource (or alternatively, the semaphore associated with the resource) is equal to the priority of the highest priority task that can lock it. Priority ceiling ensures that when a job of a task is allowed to enter the first critical section, it cannot be blocked by lower priority tasks until the completion of all its critical sections. SRP extends PCP by allowing [16] support for dynamic priority scheduling, the use of multi-unit resources and also sharing of runtime stack for all tasks in the processor. In SRP, no task is permitted to start until its priority is greater than the system ceiling and also, no task cannot be blocked once it starts.

A set of terms associated with resource access in a multiprocessor system are introduced before exploring RAPs further. Direct blocking refers to the delay incurred by a task due to lower priority tasks on the same processor. Lower priority tasks becoming non-preemptable during resource access of either local or global resources can give rise to blocking. Local Blocking due to Local resources (LBL) refers to blocking due to local resources and Local Blocking due to Global resources (LBE) refers to blocking due to global resources. Besides direct blocking, since a task on a processor can potentially be blocked while waiting for a resource that is held by a task on a different processor, remote blocking must also be accounted for in the response time analysis of tasks.

2.1.1 Spin-based protocols

The Multiprocessor Stack Resource Policy (MSRP) was introduced by Gai et al. in [5] is the multiprocessor extension of the Stack Resource Policy (SRP) [15] which is a non-preemptive spin-based protocol. In spin-based protocols, the task waiting for a global resource actively spins with its priority raised in an atomic operation typically to the highest priority on the core, making the task non-preemptable. In [5], the authors present an efficient resource access protocol in terms of memory for a multiprocessor system on chip while trading off on the processor utilization. This is done by introducing the concept of creating a mutually non-preemptive set of tasks by using an imaginary pseudo resource that is shared between them. This concept of spinning non-preemptively results in the reduction of stack memory usage.

Other examples for spin-based protocols include the Multiprocessor Bandwidth Inheritance (M-BWI) protocol [17] and the Multiprocessor resource sharing Protocol (MrsP) [18]. M-BWI
Chapter 2. Related work

allows for the both hard and soft real-time tasks to exist simultaneously in both global and partitioned systems. MrsP is a spin-based protocol and a multiprocessor variant of PCP. The tasks that are spinning under MrsP can use their spinning time for executing the critical section of the task on another processor holding the same resource but has been preempted. The preempted task migrates to the processor with the spinning task and executes its interrupted critical section before migrating back to the original processor.

2.1.2 Suspension-based protocols

The Multiprocessor Priority Ceiling Protocol (MPCP) was introduced by Rajkumar et al. in [6] and is a suspension-based protocol. In suspension-based protocols, a task waiting for a global resource suspends and releases the processor. When the resource becomes available and the task is at the head of the resource queue, it is granted access to the resource and locks the resource. The MPCP extends the Priority Ceiling Protocol (PCP) [14] to a multiprocessor system while retaining the desirable feature of PCP that bounds the remote blocking duration of a job as a function of the duration of critical sections of the other jobs. This prevents a situation in which a job is blocked while a non-critical section of a lower priority job is being executed. In other words, the task that is blocked on a global resource is suspended and as a result, other higher priority tasks that can preempt the blocked task are free to utilize the processor. Upon release of the resource, the preempted task is notified, which then acquires the priority of the resource ceiling, similar to conventional PCP.

Another example for suspension-based protocols is the Distributed Priority Ceiling Protocol (D-PCP) [19] which is designed for distributed systems. It allows for the execution of only the global critical sections of tasks on a different processor. All global critical sections of a specific global resource are bound to a so called synchronization processor. D-PCP also allows for multiple such synchronization processors to exist. D-PCP has also been proposed for fixed-priority partitioned scheduling based systems.

The Flexible Multiprocessor Locking Protocol (FMLP) introduced by Block et al. in [20] is very similar to FSLM since it also utilizes both spin and suspension based resource access protocols. In this paper, the authors claim that FMLP is flexible since it caters to the optimization of two types of resource accesses: long critical sections where suspension-based resource access is preferred and shorter critical sections where spin-based resource access is preferred due to the reduced context switching overheads. The authors also claim that FMLP supports nested resource access. It is to be noted that highest priority is used in FMLP to achieve spinning on short resource accesses while lowest priority is used for suspension on long resource accesses.

The Flexible Spin-Lock Model (FSLM) which is the focus of this master assignment is introduced by Afshar et al. in [2]. In this work, the authors generalize the phenomena of a task getting blocked on a locked resource as spinning with an arbitrary priority level. By adopting this view, it can be seen that the traditional semaphore locking approaches, spin and suspension based protocols can be viewed as two scenarios of spinning on an arbitrary priority level. A complete overview of FSLM is provided in Chapter 3.2. The original schedulability analysis without implementation overheads performed for FSLM in [2] is taken as the basis for this thesis. In this thesis, implementation overheads are characterized and incorporated into the analysis.

2.2 Memory utilization

The authors of [5] apart from introducing MSRP also investigate methods of minimizing the RAM usage, particularly with respect to the stack memory utilization of the tasks while using MSRP. The number of task frames on the stack while assuming a non-interleaved execution of tasks under a preemptive scheduling algorithm is equal to the number of priority levels. To selectively disable preemptions in order to minimize the stack memory requirement, the concept of preemption thresholds [21] is used. According to this concept, every task is assigned a priority level and a preemption threshold that is greater than or equal to the priority level. The task is inserted into the ready queue using the priority level whereas during execution, the priority is raised to the preemption level. By property of MSRP, the authors then establish that it is sufficient to use a single stack memory space per core for all the tasks. By introducing an algorithm for assigning preemption thresholds to the tasks in a processor, the authors of [5] restrict the number of
2.3 RAP implementation

Tasks that could occur simultaneously in the stack since tasks occupying the same preemption thresholds cannot occur on the stack simultaneously.

A method similar to MSRP for utilizing shared stack in FSLM has been proposed in [7]. Since FSLM allows for spinning on priorities other than the highest priority, it is no longer feasible to use a single stack for all tasks on the core. Spinning on priority levels less than the highest priority means that now, a task can be preempted while spinning and might be granted access to the resource while preempted, causing interleaved task executions if the same stack space is used for all tasks in the core. The authors of [7] establish by means of a theorem that it is sufficient to use two stacks per core for FSLM. This is achieved by moving all the tasks with priorities higher than the spin-lock priority of the core to a separate shared stack while using another shared stack for all the tasks with priorities at most the spin-lock priority. This subject constitutes one of the goals of the master thesis and it is explored in further detail in Chapter 5 by providing an implementation for dual shared stacks and investigating the memory savings offered by employing such shared stacks.

2.3 RAP implementation

A comparison between implementations of MPCP and MSRP was performed in [22] on a Janus dual-processor system on chip. The resources were modeled by using various peripheral devices such as serial ports and A/D converter. Shared memory resources were also used in the modeling of resources. The critical sections lengths were modeled as 5\( \mu \)s and 50\( \mu \)s. The critical sections were modeled as short when compared to the overall execution times of the tasks similar to typical automotive applications. The tasks spent close to 20% of the execution time while accessing critical resources. The comparison suggested that for a typical automotive application, MSRP performed better than MPCP in terms of ease of implementation, lesser overheads and optimization of the memory used.

A first implementation of the PCP [14], SRP [15], MPCP (an extension of PCP for multiprocessors), D-PCP [19] (a variant of MPCP used for distributed systems) and FMLP [20] synchronization protocols has been discussed in [23]. A LITMUS\( ^{RT} \) [24] platform has been selected for implementation, which is a real-time extension of the Linux operating system. This paper presents an integrated implementation of the above mentioned protocols in a single unified framework on a general-purpose operating system. The authors of [23] prefer algorithms that mostly correct results with realistic practical performance rather than algorithms with superior theoretical performance with unrealistic assumptions that fail when implemented. The authors also remark that it might be difficult to foresee all possible interactions in an operating system scenario from a theoretical perspective therefore requiring realistic implementations to make performance comparisons that support the theoretical claims.

In [25] a schedulability comparison has been made among MPCP, D-PCP and FMLP considering runtime overheads on LITMUS\( ^{RT} \). The experiments showed that the spin-based FMLP variant provided better performance. In other words, the suspension-based protocols incurred greater overheads when compared to the FMLP variant. However, when the experiments were repeated when ignoring the overheads, it was found that the suspension-based protocols provided high utilization and high schedulability. The results confirmed their earlier results in [26] regarding a preference for a spin-based approach over a suspension-based approach under Earliest Deadline First (EDF) scheduling. This establishes that the implementation overheads play a crucial role in determining the performance of any given protocol.

An initial implementation of FSLM on Erika Enterprise [8] for the Altera Nios II platform [27] was developed in [4] and [3]. The implementation was performed by modifying the native implementation of MSRP on Erika Enterprise to accommodate FSLM. However, it was found that the overheads in the initial implementation were significant and thus resulted in a much inferior performance in comparison to the native implementation of MSRP. A few opportunities for improving the implementation to reduce the overheads were identified in [4]. The works presented in [3] and [4] form the basis for this master thesis as the initial implementation is improved further for better performance in terms of memory and reduced overheads.
2.4 Overhead analysis

An analysis about the effects of context switching overheads in real-time systems has been presented in [28]. Context switch overheads in Linux on ARM based platform were measured in [29] and was found to be only 0.17\% to 0.25\%. [30] analyzes the data from a real Olympus satellite control system and shows that the worst-case context switching overheads can be much higher.

Multiple methods for incorporating context switching overheads in the worst case schedulability analysis have been explored in past literature. The most common method of increasing the computation time of each job by the overhead of context switching in and out of the task has been proposed in [31] [32]. Simplified versions of the previous case where both the context switch in and out of a task are assumed to be equal are proposed in [33] [34] and [30]. This method presents a pessimistic model since when a task is released with a priority less than that of the running task, the context switch into the task need not be accounted for as it has been accounted for in the context switch out of the higher priority task itself [35]. A method of charging the preempted task rather than the preempting task with context switching overheads has been presented in [16].

It can be seen that there are different ways in which overheads can be incorporated into the scheduling analysis. These works provide directions to be used when characterizing the runtime overheads into existing analysis. In this thesis, the implementation overheads are identified in a generic way and formulated as specific terms which are then incorporated into the schedulability analysis based on the conditions in which they occur.
Chapter 3

Preliminaries

This chapter provides an overview of the preliminaries required for the rest of the thesis. A description of the real-time scheduling model, a brief overview of FSLM and a description of the hardware and software systems used for the thesis are included in this chapter.

3.1 Real-time scheduling model

In this section, a description of the real-time scheduling model is presented. The real-time scheduling model (system model) provides a detailed list of the terms used in the system which are later used to describe the behavior of the system and to provide a schedulability analysis for the Flexible Spin-Lock Model. Similar terms are introduced in a later chapter of the report (Chapter 7) to accommodate the terms associated with runtime overheads into the existing analysis. The description of the system model is divided into the basic model and resource sharing related terms.

3.1.1 Basic model

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathcal{P} )</td>
<td>Set of all processors in the system. ( (m ) identical processors ( P_0, P_1, \ldots, P_{m-1} ) )</td>
</tr>
<tr>
<td>( P_k )</td>
<td>Processor (core) with an integer identifier/index ( k ) where ( P_k \in \mathcal{P} )</td>
</tr>
<tr>
<td>( \mathcal{T} )</td>
<td>Set of all the tasks in the system. ( (n ) sporadic tasks ( \tau_0, \tau_1, \ldots, \tau_{n-1} ) )</td>
</tr>
<tr>
<td>( \tau_i )</td>
<td>Task with an integer identifier/index ( i ). ( () Every task ( \tau_i ) consists of an infinite sequence of jobs ( ) )</td>
</tr>
<tr>
<td>( T_{P_k} )</td>
<td>The set of tasks allocated to a processor ( P_k ).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C_i )</td>
<td>Worst-case computation time of task ( \tau_i ).</td>
</tr>
<tr>
<td>( T_i )</td>
<td>Minimum inter-arrival time.</td>
</tr>
<tr>
<td>( D_i )</td>
<td>Relative deadline of task ( \tau_i ). ( () Deadlines are assumed to be constrained, meaning ( D_i \leq T_i ) )</td>
</tr>
<tr>
<td>( \pi_i )</td>
<td>Priority of the task ( \tau_i ). ( \tau_i ) has a higher priority than ( \tau_j ) if ( \pi_i &gt; \pi_j ) where ( i &gt; j ) No two tasks in ( T_{P_k} ) can have the same priority.</td>
</tr>
<tr>
<td>( WR_i )</td>
<td>Worst-case response time of task ( \tau_i ). The set ( T ) is schedulable if and only if ( \forall 0 \leq i \leq n-1 ; WR_i \leq D_i ).</td>
</tr>
</tbody>
</table>

3.1.2 Resource sharing

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathcal{R} )</td>
<td>Set of all local and global resources in the system.</td>
</tr>
<tr>
<td>( \mathcal{R}_{P_k} )</td>
<td>Set of local resources accessed by the tasks in core ( P_k ).</td>
</tr>
</tbody>
</table>
### Chapter 3. Preliminaries

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{RS}^G_{P_k}$</td>
<td>Set of global resources accessed by the tasks in core $P_k$.</td>
</tr>
<tr>
<td>$\mathcal{T}_{P_k,q}$</td>
<td>Set of tasks on a processor $P_k$ using a resource $R_q$.</td>
</tr>
<tr>
<td>$Cs_{i,q}$</td>
<td>Worst-case computation time of the critical section among all the requests by any job of $\tau_i$ for any resource $R_q$.</td>
</tr>
<tr>
<td>$n^G_{i,q}$</td>
<td>Maximum number of requests of a job $\tau_i$ for a global resource $R_q$.</td>
</tr>
<tr>
<td>$n^L_{i,l}$</td>
<td>Maximum number of requests of a job $\tau_i$ for a local resource $R_l$.</td>
</tr>
<tr>
<td>$\text{ceil}_{P_k}(R)$</td>
<td>Ceiling of any resource $R$ on processor $P_k$.</td>
</tr>
<tr>
<td>$\pi^\text{spin}_{P_k}$</td>
<td>Spin-lock priority on a processor $P_k$.</td>
</tr>
<tr>
<td>$\pi^\text{max}_{P_k}$</td>
<td>The highest priority (HP) on a processor $P_k$.</td>
</tr>
<tr>
<td>$\pi^G_{P_k}$</td>
<td>The highest resource ceiling of the global resources (or ceiling priority, CP) used on a processor $P_k$.</td>
</tr>
<tr>
<td>$\pi^\text{LG}_{P_k}$</td>
<td>The highest resource ceiling of any resource (global or local) ($\hat{\text{CP}}$) used on a processor $P_k$.</td>
</tr>
<tr>
<td>$B_i$</td>
<td>Total local blocking time experienced by a task $\tau_i$.</td>
</tr>
<tr>
<td>$B^L_i$</td>
<td>Local blocking due to local resources (LBL) experienced by a task $\tau_i$.</td>
</tr>
<tr>
<td>$B^G_i$</td>
<td>Local blocking due to global resources (LBG) experienced by a task $\tau_i$.</td>
</tr>
<tr>
<td>$\text{spin}_{P_k,q}$</td>
<td>Maximum remote blocking spin-lock time for a task on $P_k$ to acquire $R_q$.</td>
</tr>
<tr>
<td>$\text{spin}_i$</td>
<td>Maximum total remote blocking time for a task $\tau_i$ to acquire all its resources.</td>
</tr>
</tbody>
</table>

The delays caused due to resource sharing for any task in a multiprocessor system can be attributed to **local blocking** and **remote blocking**. Local blocking is also known as priority inversion blocking (pi-blocking) [19]. The total local blocking time experienced by a task $\tau_i$ is denoted by $B_i$. Local blocking due to local resources (LBL) experienced by a task $\tau_i$ is denoted by $B^L_i$ while the local blocking due to global resources (LBG) is denoted by $B^G_i$. The maximum remote blocking spin-lock time for a task on $P_k$ to acquire $R_q$ is denoted by $\text{spin}_{P_k,q}$. The maximum total remote blocking time for a task $\tau_i$ to acquire all its resources is denoted by $\text{spin}_i$.

### 3.2 FSLM overview

The Flexible Spin-Lock Model [2] (FSLM) is a resource sharing protocol for P-FPPS based multiprocessor real-time embedded systems that unifies spin-based and suspension-based approaches by explicitly identifying the spin-lock priority as a parameter. Such protocols are recommended by AUTOSAR [1], the standard for the automotive domain. A brief overview of the Flexible Spin-Lock Model introduced in [2] is presented in this section. The contents of this section have been extracted and summarized from [2], [4] and [7].

#### 3.2.1 Resource sharing rules in FSLM

The resource sharing rules based on a general spinning model for FSLM are presented in this section. The focus of FSLM is for partitioned platforms, meaning that the tasks are mapped onto processors off-line and the allocation remains static throughout the runtime of the system. Partitioned scheduling is attractive due to easier implementation and lower runtime overheads. The general idea of FSLM is that a task $\tau_i$ waiting for a global resource will spin (busy wait) whenever the resource is unavailable. In this thesis, only the spin-lock priorities in the range [CP, HP] are considered although FSLM supports spin-lock priorities in the range [LP, HP]. Spin-lock priorities...
3.2. FSLM overview

in [CP, HP] are particularly interesting because of the improved schedulability offered by spin-lock priorities CP and higher [7]. Figure 3.1 illustrates a task spinning on a core while waiting for a global resource to become available. The rules for resource sharing under FSLM are as follows:

![Figure 3.1: Illustration of a task spinning while waiting on a global resource [4].](image)

**Rule 1.** Local resources are handled by means of SRP [15] uniprocessor synchronization protocol.

**Rule 2.** For each global resource a FIFO-based queue is used to enqueue the tasks waiting for the related resource.

**Rule 3.** Whenever a task $\tau_i$ on a core $P_k$ requests a global resource that is used by tasks on other processors, it places its request in the related resource queue and performs a busy wait (also called spin lock). The task will spin with a priority level $\pi_{\text{spin}}^i$ where $0 \leq \pi_{\text{spin}}^i \leq \pi_{\text{max}}^{P_k}$.

**Rule 4.** When a task $\tau_i$ is granted access to its requested global resource on a processor $P_k$, its priority is boosted in an atomic operation to $\pi_{\text{max}}^{P_k} + \pi_i$.

**Rule 5.** When a task gets access to its requested resource, it becomes non-preemptable until it releases the resource.

**Rule 6.** The priority of the task is changed to its original priority as soon as it finishes the global critical section and it becomes preemptable again.

**Rule 7.** When the resource becomes available (i.e. it is released), the task at the head of the resource global queue (if any) is granted the resource.

3.2.2 Special spin-lock priorities

The FSLM model used in this thesis focuses on spin-lock priority levels in the range [CP, HP]. It is useful to define these two priority levels.

**Def. 1.** HP - The highest priority level on a processor $P_k$ is denoted as $\pi_{\text{max}}^{P_k}$.

$$\pi_{\text{max}}^{P_k} = \max\{\pi_i | \tau_i \in T_{P_k}\} \quad (3.1)$$

Under the HP spin-lock approach, $\pi_{\text{spin}}^i = \pi_{\text{max}}^{P_k}$.

**Def. 2.** CP - The highest local ceiling of any global resource on a processor $P_k$ is denoted as $\pi_{\text{G}}^{P_k}$.

$$\pi_{\text{G}}^{P_k} = \max\{\pi_i | \tau_i \in T_{P_k} \land \mathcal{R}_{i}^{G} \neq \emptyset\} \quad (3.2)$$

Under the CP spin-lock approach $\pi_{\text{spin}}^i = \pi_{\text{G}}^{P_k}$.

3.2.3 Recap of existing analysis without overheads

This section presents a brief recap of the existing blocking analysis for FSLM under HP ($\pi_{\text{spin}}^i = \pi_{\text{max}}^{P_k}$) and CP ($\pi_{\text{spin}}^i = \pi_{\text{G}}^{P_k}$) spin-lock approaches [2]. The set $T$ is schedulable if and only if $\forall 0 < i < n - 1 \ WR_i \leq D_i$. Without overheads, the analysis for FSLM-HP is identical to the analysis for MSRP.
Chapter 3. Preliminaries

HP spin-lock approach

Under HP, spinning is non-preemptable. The upper bounds $B^L_i$ for LBL and $B^G_i$ for LBG experienced by a task $\tau_i \in T_{pk}$ are given by:

$$B^L_i = \max_{\forall j: \pi_j < \pi_i, \tau_j \in T_{pk}} \{Cs_{j,i}\}$$  \hspace{1cm} (3.3)

and

$$B^G_i = \max_{\forall j: \pi_j < \pi_i, \tau_j \in T_{pk}} \{Cs_{j,i} + \text{spin}_{pk,j}\},$$  \hspace{1cm} (3.4)

respectively, where $\text{spin}_{pk,j}$ is given by

$$\text{spin}_{pk,j} = \sum_{\forall q: R_q \in RS^G_j} \max_{\tau_q \in T_{pk}} Cs_{j,q} \cdot$$  \hspace{1cm} (3.5)

The upper bound on the total pi-blocking ($B_i$) experienced by a task $\tau_i \in T_{pk}$ is now given by

$$B_i = \max\{B^L_i, B^G_i\}.$$  \hspace{1cm} (3.6)

The inflated computation time $\hat{C}_i$ of task $\tau_i$ is defined as:

$$\hat{C}_i = C_i + \text{spin}_i,$$  \hspace{1cm} (3.7)

where the term of $\text{spin}_i$ is upper bounded by

$$\text{spin}_i = \sum_{\forall q: R_q \in RS^G_j, \tau_q \in T_{pk}} \left( \pi_i - \pi_{G_{pk}} \right) \times \text{spin}_{pk,j}.$$  \hspace{1cm} (3.8)

Finally, the worst-case response time $WR_i$ of task $\tau_i$ is given by the smallest positive solution of the following recursive equation

$$WR_i = C_i + B_i + \sum_{\forall h: \pi_h > \pi_i, \tau_h \in T_{pk}} \left\lceil \frac{WR_h}{T_h} \right\rceil \hat{C}_h.$$  \hspace{1cm} (3.9)

CP spin-lock approach

Under CP, the task blocked on a global resource spins at the highest local ceiling of global resources on the core, hence making it non-preemptable by other tasks using global resources on the same core.

For the upper bound for LBL experienced by a task $\tau_i \in T_{pk}$, we can reuse (3.3). The upper bound on LBG experienced by a task $\tau_i \in T_{pk}$ for the CP spin-lock approach is defined as

$$B^G_i = \max_{\forall j: \pi_j < \pi_i, \tau_j \in T_{pk}} \left\{ \begin{array}{ll} Cs_{j,i} & \text{if } \pi_i > \pi_{G_{pk}}; \\ Cs_{j,i} + \text{spin}_{pk,j} & \text{if } \pi_i \leq \pi_{G_{pk}}. \end{array} \right.$$  \hspace{1cm} (3.10)

The total pi-blocking $B_i$ experienced by a task $\tau_i \in T_{pk}$ has the following upper bound:

$$B_i = \max\{B^L_i + B^G_i, \max\{B^L_i + B^G_i\} + 1\}$$  \hspace{1cm} (3.11)

The upper bounds for $\text{spin}_{pk,j}$ and $\text{spin}_i$, the inflated computation time $\hat{C}_i$ and the worst-case response time $WR_i$ are calculated as per (3.5), (3.8), (3.7) and (3.9), respectively.
The schedulability analysis presented in Section 3.2.3 will be extended in order to include the effects of induced implementation overheads in Chapter 7. This concludes the recap of the Flexible Spin-Lock Model.

3.3 System overview

In this section, a brief overview of the FPGA hardware design process and the software system used for the thesis are presented.

3.3.1 Hardware system

A brief overview of the general FPGA design process for Nios II based multiprocessor systems and the hardware system used for this thesis are presented below.

During the preparation phase of the master thesis, to satisfy goal [I.1] of the thesis, a new hardware platform had to be selected. Since the Altera DE0 FPGA platform used for the initial implementation [4] did not have sufficient embedded memory for instantiating 4 soft processing cores, the Altera DE2-115 FPGA board was decided as the alternative. The detailed selection process for the new hardware platform was presented at the end of the preparation phase. Altera FPGA boards allow for the use of Nios II soft processing cores in the hardware design.

The common design process to be followed for a Nios II based multiprocessor system on an Altera FPGA board is detailed in [36]. Figure 3.2 shows the design steps recommended by Altera.

Using the recommended design process, a four processor hardware system was created to be used for implementations and measurements throughout the thesis. Figure 3.3 shows a simplified overall schematic of the four core hardware system designed for the study. A brief summary of the salient features of the hardware design is presented in Table 3.3.
Chapter 3. Preliminaries

<table>
<thead>
<tr>
<th>Processing core</th>
<th>Nios II/e (32 bit RISC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#cores</td>
<td>4</td>
</tr>
<tr>
<td>On-chip memory per core</td>
<td>48 kb</td>
</tr>
<tr>
<td>Clock frequency</td>
<td>50 MHz</td>
</tr>
<tr>
<td>Peripherals</td>
<td></td>
</tr>
<tr>
<td></td>
<td>hardware mutex (1x)</td>
</tr>
<tr>
<td></td>
<td>performance counter (2x)</td>
</tr>
<tr>
<td></td>
<td>JTAG UART (1x)</td>
</tr>
</tbody>
</table>

**Table 3.3:** Hardware design specifications.

**Figure 3.3:** Overview of the hardware system created for the study.

3.3.2 Software system

Due to the support available for MSRP and being OSEK compliant, Erika Enterprise RTOS was chosen as the operating system of choice for the implementation of FSLM in [4]. Since this assignment is a continuation of the previous work done in [3] and [4], the operating system choice remains unchanged.

Figure 3.4 shows the development view of the software system used for the study. The figure presents an overview of the components of the Erika Enterprise RTOS.

The following are a list of primary features of the software system:

- The Erika Enterprise RTOS API enables the application developer (user) to access some of the operating system kernel primitives. The primitives used for interacting directly with the underlying hardware are not exposed to the user through the API and are exclusively handled by the kernel.

- The kernel layer in turn uses the Hardware Abstraction Layer (HAL) provided by Altera to gain access to the underlying hardware.

- In order to provide the user with the options for configuring the kernel, such as specifying the conformance class, mapping the tasks on to specific processors (especially for P-FPPS systems), setting the task priorities and so on, an OIL (OSEK Implementation Language) file is used. OIL is an OSEK standard text based language that allows the user to provide a textual definition of the objects such as cpu, tasks and resources. A more detailed overview of the OIL file is presented in Chapter 5.3.
3.3. System overview

- RT-Druid [37] is a code generator plug-in for the Eclipse [38] development framework. RT-Druid is used to automatically generate the operating system files based on the user specified configuration options in the OIL file during compile time.

- An executable file with the extension .elf is created at the completion of the software design process. The executable file is then programmed onto the hardware platform.

- The multi-core extension [27] of Erika Enterprise with the “multistack” configuration of the “BCC2” conformance class of the OO (OSEK OS) kernel [39] of Erika Enterprise was used to perform the experiments in this study.

For the sake of conciseness of the report, the detailed instructions for creating the hardware and software design are presented in Appendix A.
Chapter 4

Inter-core communication

A significant difference between the implementation of MSRP and FSLM is the need for an inter-core communication mechanism in FSLM. In this chapter, a brief justification for the need of such an inter-core communication mechanism is explained. The initial implementation of FSLM [3] [4] used an inter-core communication mechanism called Remote Notifications (RN) provided by Erika Enterprise, which was found to have significant runtime overheads. This chapter provides an alternative, efficient method for inter-core communication named Dedicated Interrupts (DI), details the design specifications of the new mechanism and supplements it with measurements results comparing the new mechanism with the existing RN based implementation.

4.1 The spin-lock variable

The primary goal of a multiprocessor Resource Access Protocol (RAP) is to provide mutual exclusive access to shared resources among competing tasks. A resource that is shared among tasks in various cores is called as a global shared resource. Every task that uses a global shared resource executes a piece of code called global critical section under mutual exclusion. A spin-lock variable is used by the RAP to moderate the access to the same resource by competing tasks on different cores. This presents a choice of using either a global spin-lock variable or a local spin-lock variable based on the type of RAP.

4.1.1 Spinning on a global spin-lock

The original implementation of the multiprocessor adaptation of Erika Enterprise RTOS [27] includes support for MSRP. Since MSRP is a spin-based protocol where a task blocked on a global resource request spins at the highest priority ($\pi_{max}$) until the requested resource becomes available, the task requesting the resource will continue to remain active from the point at which the resource request is made until the completion of the global critical section, even during spinning. This allows MSRP to utilize low-level global spin-lock variables that are shared between all the cores in the system. Any task that is using a resource acquires the spin-lock variable until the completion of its global critical section. Any other task residing on a different core trying to access the same resource finds that the resource is unavailable and continues to spin at the highest priority until the spin-lock variable becomes available. Since the spinning task cannot be preempted under the MSRP approach, this allows for the continuous polling of the global low-level spin-lock variable by the spinning task until it eventually becomes available, thereby negating the need for any inter-core communication mechanism to notify the release of resources. Example 1 illustrates the implementation of such a global spin-lock based mechanism.

Example 1. Consider a spin based RAP such as MSRP in which the task blocked on a resource spins at the highest priority level (HP). The system model for the example consists of two processors $P_0$ and $P_1$ with one task allocated to each of them $\tau_0$ and $\tau_1$ respectively. Both the tasks $\tau_0$ and $\tau_1$ require access to a shared global resource $R_0$. The state of the data structures initially is shown in Figure 4.1a. The shared global memory contains the data structure Spin_lock which is an array consisting of one spin-lock variable per global resource as shown in the figure. Task $\tau_0$ requests access to the resource $R_0$ by checking the status of the Spin_lock variable of resource $R_0$. Since it is available (un-locked), $\tau_0$ gains access to the resource by locking the spin-lock variable of $R_0$ and proceeds to use the resource. In the meanwhile, task $\tau_1$ tries to access the resource $R_0$ while it is still being used by $\tau_0$. The task $\tau_1$ knows that the resource $R_0$ is unavailable by checking the spin-lock variable of $R_0$, since it is now locked by $\tau_0$. Hence, $\tau_1$ starts spinning...
Chapter 4. Inter-core communication

4.1 Inter-core communication

4.1.1 Spinning on a global spin-lock

Figure 4.1: Spinning on a global spin-lock.

at the highest priority on $P_1$ until the spin-lock variable of $R_0$ is unlocked by $\tau_0$. This scenario where the task $\tau_0$ is actively using the resource $R_0$ while the task $\tau_1$ spins on the global spin-lock variable is shown in Figure 4.1b. Once the task $\tau_0$ completes the execution of its global critical section, it unlocks the spin-lock variable of $R_0$ in one atomic transaction thereby allowing $\tau_1$ to stop spinning and access the resource $R_0$.

To facilitate FIFO ordered global resource access, the native MSRP implementation in Erika Enterprise maintains a distributed polling-bit queue for each global resource (G-T algorithm [40]), i.e. a (non-empty) FIFO queue of polling bits used by cores that want to access that global resource. As per the original implementation, there is a “global polling bit” per core per global resource which is exclusively toggled by the owner core when a global resource is accessed or released. This data is read (polled) by the other cores that require access to the same resource. To enable this, another global data structure per global resource, called “spin-status” is used. The “spin-status” contains the tail of the non-empty FIFO queue. In other words, it contains the address of the global polling-bit that the next core requesting the same resource must inspect. This mechanism creates a FIFO ordered queue for global resource request and is given in detail in Section 5.6 of [4].

It can be seen from Example 1 that this approach of spinning on a global spin-lock is feasible only when the spinning task is allowed to spin at the highest priority (HP), allowing the task to continuously poll the global spin-lock variable of the resource until it becomes available. Hence, within the context of FSLM, this approach is feasible only when $\pi_{P_k}^{spin} = \pi_{P_k}^{max}$, i.e only for FSLM-HP spin-lock approach. For any other arbitrary spin-lock priority level in the range $[CP, HP)$, this method is not feasible.

4.1.2 Spinning on a local spin-lock

FSLM, by definition, allows for the task blocked on a global resource request to spin at any arbitrary priority level while this thesis considers spin-lock priorities in $[\pi_{P_k}^{min}, \pi_{P_k}^{max}]$, i.e $[CP, HP]$. As a result, if the spin lock priority for a core $\pi_{P_k}^{spin} < \pi_{P_k}^{max}$ it is now possible for a task that is spinning to be preempted by another task with priority $\pi_i > \pi_{P_k}^{spin}$. Since the spinning task is no longer guaranteed to remain as the active task until the resource is granted, the continuous polling of a global low-level spin-lock variable is not feasible. Instead, an interrupt based method has to be used where the core with the task blocked on a resource gets notified by the core with the task releasing the resource.
4.1. The spin-lock variable

The initial implementation of FSLM performed in [3] [4], utilized the Erika Enterprise original implementation of MSRP as the basis on which changes were made to include support for FSLM. The detailed list of changes performed to the original implementation can be found in Chapter 8 of [4]. However, for the sake of completeness of the report and to aid the reader in comprehending the motivation for the addition of an inter-core communication mechanism, the primary design choices made in [4] are detailed in this part of the report. In order to add support for spinning at arbitrary priority levels, the following major changes were made to the original Erika Enterprise in [4]:

1. Spinning on a local spin-lock variable instead of global spin-lock variable.
2. Notification of resource release by the releasing task to a blocked task in a remote core when applicable.

Implementation of a local-spin lock method in FSLM

Although it is termed as spinning on a local spin-lock variable, in order to co-ordinate the resource access among tasks residing in multiple cores, it is essential to use global data structures. The following global data structures were used along with a local Spin_lock variable (one per core) in the initial implementation of FSLM [3] [4]:

- ResourceQ - A 2-dimensional array with \( n \times m \) entries where \( n \) denotes the number of global resources in the system and \( m \) denotes the number of processors in the system.
- TailQ - A single dimensional array consisting of \( n \) entries where \( n \) denotes the number of global resources in the system. Denotes the current spin status of the resources by pointing to the tail of the ResourceQ data structure.

The resource requests and accesses under the FSLM implementation on Erika Enterprise are handled as follows:

1. Contents of ResourceQ[\( R_q \)][P_k] are initialized to 0xa0 at startup, denoting that \( P_k \) is not actively using or has pending requests for resource \( R_q \).
2. Contents of TailQ[\( R_q \)] are initialized pointing to the address of ResourceQ[\( R_q \)][P_0].
3. When a task on a core \( P_k \) tries to access a resource \( R_q \), it checks for the contents of the address pointed to by TailQ[\( R_q \)]. The resource is available if the contents of the address pointed to by TailQ[\( R_q \)] is 0xa0 and unavailable otherwise.
4. If the resource is available, the task stores its TaskID in ResourceQ[\( R_q \)][P_k] denoting that it is currently using the resource. It also updates TailQ[\( R_q \)] to point to ResourceQ[\( R_q \)][P_k]. Any task from a different core \( P_l \) trying to access the same resource now will know that the resource is unavailable because the contents of the address pointed to by TailQ[\( R_q \)] will now be the TaskID of the task using the resource instead of 0xa0.
5. If the resource is unavailable, the task from core \( P_l \) locks its local Spin_lock variable. It also updates the value at the address pointed to by TailQ[\( R_q \)], i.e ResourceQ[\( R_q \)][P_k] with its own TaskID. This is the location the core releasing the resource will check to know which core to notify upon release. The task also updates the contents of ResourceQ[\( R_q \)][P_l] with its TaskID and then updates TailQ[\( R_q \)] to point to ResourceQ[\( R_q \)][P_l].
6. During resource release, the core \( P_k \) releasing the resource checks ResourceQ[\( R_q \)][P_k]. If the contents are equal to its TaskID, then the core does not have to notify any other core. Otherwise, the TaskID is used to identify the core to which a notification has to be sent.
7. Upon receipt of a notification (Remote Notification)\(^1\) in core \( P_l \), the Interrupt Service Routine (ISR) resets the value of the Spin_lock variable and the task on core \( P_l \) resumes resource access. In case the spinning task has been preempted by higher priority task, the RN contains the TaskID of the requesting task so that context switch can be facilitated.

\(^1\)Remote Notifications are a shared memory and interrupt based inter-core communication mechanism provided by the Erika Enterprise kernel, explained further in Section 4.2.
Chapter 4. Inter-core communication

Example 2. The example illustrates the functioning of the initial implementation of FSLM as described in Section 4.1.2. The system model for the example consists of two processors $P_0$ and $P_1$ with one task allocated to each of them $\tau_0$ and $\tau_1$ respectively. Both the tasks $\tau_0$ and $\tau_1$ require access to a shared global resource $R_0$. Figure 4.2a shows the initial data structures during startup. Figure 4.2b shows the status of the data structures when the task $\tau_1$ on processor $P_1$ tries to access resource $R_0$. The task initially checks the content of the address pointed to by $\text{TailQ}[R_0]$, i.e $\text{ResourceQ}[R_0][P_0]$ in this case. Since initially, the contents of $\text{ResourceQ}[R_0][P_0]$ is 0xa0, it signifies that the resource $R_0$ is available and the task $\tau_1$ acquires the resource. The task $\tau_1$ now updates the value of $\text{ResourceQ}[R_0][P_1]$ with its TaskID and also changes the value of $\text{TailQ}[R_0]$ to point to $\text{ResourceQ}[R_0][P_1]$. 

![Figure 4.2: Spinning on a local spin-lock.](image)

![Figure 4.3: Spinning on a local spin-lock.](image)
Figure 4.3a shows the status of the data structures when the task $\tau_0$ in core $P_0$ tries to access the resource $R_0$ while it is still being used by $\tau_1$. Task $\tau_0$ checks the content of the address pointed to by TailQ[$R_0$]. Since TailQ[$R_0$] points to ResourceQ[$R_0$][$P_1$] which contains the TaskID of $\tau_1$ instead of 0x0, it means that the resource is currently unavailable. Task $\tau_0$ sets the Spin_lock variable on $P_0$ to locked.

Task $\tau_0$ now updates the value of ResourceQ[$R_0$][$P_1$] with its TaskID ($\tau_0$) since this is the location where $\tau_1$ will check upon completion of resource access to check which core has to be notified. After updating ResourceQ[$R_0$][$P_1$], the task $\tau_0$ updates the value of ResourceQ[$R_0$][$P_0$] with its TaskID and sets TailQ[$R_0$] to point to ResourceQ[$R_0$][$P_0$]. The task $\tau_0$ now starts to spin.

Figure 4.3b illustrates the status of the data structures when the task $\tau_1$ on $P_1$ has completed the resource access. Upon completion of resource access, the task $\tau_1$ checks the content of ResourceQ[$R_0$][$P_1$]. If the contents are anything apart from the TaskID of $\tau_1$ itself, it signifies that another task from a remote core has made a request in the meanwhile and it has to be notified regarding the release of the resource. In this example, upon release of the resource $R_0$ by $\tau_1$, the content of ResourceQ[$R_0$][$P_1$] is the TaskID of $\tau_0$ denoting that a notification has to be sent to core $P_0$. The task $\tau_1$ updates the value of ResourceQ[$R_0$][$P_1$] to 0x0a0 and then sends a Remote Notification to the core $P_0$ as shown in the figure. Upon receipt of the interrupt in $P_0$, the ISR resets the local Spin_lock variable. The RN message buffer contains the TaskID of the task $\tau_0$ to facilitate the context switch. The task $\tau_0$ now resumes and accesses the resource $R_0$.

## 4.2 Remote Notification mechanism

The working of the Remote Notification mechanism provided by the Erika Enterprise operating system which is used in the initial implementation of FSLM [3] [4] is summarized in this section. A complete description of the working of RN mechanism can be found in Chapter 5.7 of [4].

Remote Notification is an inter-core communication mechanism provided by Erika Enterprise that uses shared memory between the cores to exchange messages while using a system of inter-core interrupts to notify the cores regarding an impending action.

In the Erika Enterprise manual for the multi-core implementation of the RTOS on FPGAs [27], the central idea behind remote notifications is explained. In order to hide the multiprocessor structure to retain the same RTOS API calls for both single and multiprocessor implementations, remote notifications are used. The remote notification mechanism is based on multi-core interrupts and it is used to provide an implementation to the kernel primitives acting on tasks allocated to remote processors. Erika claims that the remote notification mechanism is hidden inside the implementation of the Erika Enterprise primitives in order to maintain the same interface to the developer for both single and multi-core systems.

The manual [27] also states that there is in fact an additional overhead caused by the remote execution of calls due to sending of remote notifications from one side and inter-processor interrupt being raised on the receiving side. It is stated that the majority of the overhead results from the usage of Altera Avalon Mutex calls internally. The Avalon bus connects the processors to the peripherals on the FPGA platform. Figure 4.4 shows the hardware connections in between cores that enables the remote notification mechanism. It can be observed that each core has one input to receive interrupts while there are 4 outputs to send interrupts to all the cores. The software implementation prevents the possibility of a core interrupting itself, i.e a core cannot send a RN to itself.

A brief description of the working of the remote notification mechanism is presented below. It is essential to know that the communication happens through interrupts and shared memory between the cores. The shared memory contains the RN data structures. All the cores have access to this RN data structure, thus enabling inter core communications. Both the sending and receiving actions are performed without interruptions in order to maintain consistency. Erika Enterprise utilizes a set of two RN message buffers on each core. This is to ensure that while one message buffer is being used by the receiving core, another sending core can still add messages on the other message buffer. A new type of RN message was introduced in [4] that utilizes the same underlying RN mechanism for the purpose of communicating the release of resource between cores.

The following non-exhaustive list of steps capture the working of RN mechanism briefly:

- The sending core acquires the mutex on the shared memory, the memory where the RN data structure of the receiving core is located.
- The sending core modifies the content of the receiving cores RN data structure.
• The sending core releases the lock.
• The sending core raises an interrupt via the interconnect on the receiving core.
• The receiving core, upon receiving the interrupt, acquires the lock on its RN data structure.
• The core reads the message and releases the lock.
• The receiving core executes the RN message.

The above operations are accomplished using the functions `EE_rn_send()`, `EE_rn_handler()` and `EE_rn_execute()` implemented in the Erika kernel and declared in the file `ee_rn_internal.h`.

4.3 Dedicated Interrupts

One of the primary goals of this thesis is to implement an inter-core communication mechanism that reduces the runtime overheads incurred due to the existing Remote Notification mechanism. This section details the motivation for such a system, the design considerations and the details of the implementation. This new inter-core communication mechanism that is designed as a replacement for the RN mechanism is termed as Dedicated Interrupts (DI) since it is no longer a general purpose inter-core communication system as intended by the original implementation of Erika Enterprise, but merely a system of interrupts dedicated to indicate the release of resources in the multiprocessor RAP context.

4.3.1 Motivation for Dedicated Interrupts

The RN mechanism as explained in Section 4.2 utilizes both a system of interrupts and shared memory to facilitate inter-core communication. The use of shared memory for communication requires mutual exclusive access to the shared RN data structures giving rise to a significant runtime overhead each time such a notification is sent or received. In this section, the decision to replace the RN mechanism with a purely interrupt based notification mechanism is motivated.

Remote Notifications offer a generic interface for communicating between the cores. The nature of these communicated messages can vary depending upon the application that uses the Erika Enterprise kernel. In the original implementation of the Erika Enterprise, a total of 6 RN
types are defined and supported. Listing 4.1 shows the types of RN supported by the original Erika Enterprise. For example, the RN message type `EE_RN_TASK` is used for the remote activation of a task on a core from a different core.

```
/* Notification types */
#ifdef __RN_COUNTER__
define EE_RN_COUNTER 1
#endif

#ifdef __RN_EVENT__
define EE_RN_EVENT 2
#endif

#ifdef __RN_TASK__
define EE_RN_TASK 4
#endif

#ifdef __RN_FUNC__
define EE_RN_FUNC 8
#endif

#ifdef __RN_BIND__
define EE_RN_BIND 16
#endif

#ifdef __RN_UNBIND__
define EE_RN_UNBIND 32
#endif
```

**Listing 4.1:** RN message types in the original Erika Enterprise.

The support for each type of RN is included or excluded from the kernel based on the kernel type and conformance class requirements of the real-time application. In the initial implementation of FSLM [4] it was found that for an application based on P-FPPS, while using the multicore extension [27] of the “multistack” configuration of the “BCC2” conformance class of the OO (OSEK OS) kernel [39] of Erika Enterprise, none of the six original RN types are required to be enabled in the implementation. Hence, the initial implementation of FSLM in Erika Enterprise [3] [4] introduces a new type of RN message, `RN_ReleaseResource` used exclusively for notifying the release of a resource to another remote core.

The RN data structures consist of a control part and a message part. To leverage the existing framework of RN in [4], the new type of RN created (`RN_ReleaseResource`) uses the control and message parts of the message according to the rules of RN mechanism, detailed in Section 5.7 of [4]. The only content of the message part is the TaskID of the task on the remote core that has to be notified. This information is retrieved from the `ResourceQ[R_q][P_k]` as explained in Section 4.1.2. The content is then used by the receiving core to perform the context-switch as required if the required task has been preempted while spinning by a higher priority task with a priority `π_{i} \in T_{P_k}`.

*Lemma 12* in [2] proves that there can only be *at most one task on a processor P_k with a pending request to a global resource at any time if* \( π_{P_k}^{spin} \geq π_{G_{P_k}} \). This indicates that upon receipt of a RN, the ISR of the receiving core inherently is made aware of a resource release for one of its spinning tasks that has either been preempted by a higher priority task or is still actively spinning at `π_{P_k}^{spin}`. This motivates the removal of the usage of the RN data structures altogether and to merely use a simple inter-core hardware interrupt to signal the release of a resource for spin-lock priorities in [CP, HP]. The removal of shared memory based RN buffers is expected to significantly reduce the runtime overheads associated with each resource access and release in the implementation of FSLM.

The rest of this section focuses on the design, implementation and verification of a dedicated interrupt mechanism replacing the RN mechanism in the implementation of FSLM.

### 4.3.2 Design

The high level functional requirements for the Dedicated Interrupts mechanism are presented in this section. The invariant and predicates that must hold for the DI mechanism are also defined in this section.

**Requirements**

1. Sending the interrupt
Chapter 4. Inter-core communication

a) CPU ID of the receiving core must be determined by the sending core from the TaskID of the target task.
b) An inter-core interrupt is raised on the corresponding target core by the sending core.
c) The target core should never be the same as the sending core.
d) The sending core must execute the interrupt sender function uninterruptedly.

2. Receiving the interrupt

a) Upon receiving an inter-core interrupt, the interrupt handler on the receiving core must identify the requesting task and the resource on which the task was blocked.
b) System ceiling on the receiving core is set to non-preemptable level.
c) The local spin lock variable must be reset to unlocked on the receiving core.
d) Context switch must happen if the requesting task was preempted while spinning.

Invariant and Predicates

Invariant 1. During startup, on each core in the system, the spin-lock is unlocked, which implies that the number of received DIs is equal to the number of processed DIs.

Predicate 1. A core will receive a DI only if the local spin-lock variable is locked.

Predicate 2. The number of received DIs on a core can at most be only one greater than the number of processed DIs on the core, for spin-lock priorities in [CP, HP].

4.3.3 Implementation

The function names used for sending and handling inter-core communications in Remote Notifications (RN) are changed for the implementation of the Dedicated Interrupts mechanism although parts of the implementation are re-used. The following three functions were defined to facilitate the Dedicated Interrupts mechanism:

1. EE_di_send() - Used to send a dedicated interrupt signifying the release of a global resource to a remote core.
2. EE_di_handler() - Used as the Interrupt Service Routine on the receiving core to handle the incoming dedicated interrupt.
3. EE_di_execute() - Used to perform the required action when a dedicated interrupt is received.

Changes to EE_oo_GetResource() and EE_oo_ReleaseResource()

The implementation of dedicated interrupt mechanism for signaling resource release will no longer involve shared memory based messages as in case of RN mechanism. Hence, the TaskID of the task spinning on a remote core blocked on a resource cannot be communicated using the RN shared data structures. Additional changes have to be made upon each resource access and release to locally maintain a record of the TaskID of the task requesting and releasing the global resource.

The functions EE_oo_GetResource() and EE_oo_ReleaseResource() are used to access and release a resource respectively.

---

![Calls graph for the function EE_oo_GetResource()](image_url)
The calls graph of the functions `EE_oo_GetResource()` and `EE_oo_ReleaseResource()` are shown in Figures 4.5 and 4.6 respectively. A “Calls graph” of a function visualizes the tree of function calls used by the function. The functions `EE_hal_spin_in()` and `EE_hal_spin_out()` are used respectively to spin-in and spin-out of the global resource during resource access and release.

To keep track of the task requesting and waiting for a global resource in a core, a new local variable `EE_resource_task[2]` is updated upon each resource access and release.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data structure</td>
<td>1-d Array</td>
</tr>
<tr>
<td>Index 0</td>
<td>Resource ID</td>
</tr>
<tr>
<td>Index 1</td>
<td>Task ID</td>
</tr>
<tr>
<td>Number of elements</td>
<td>2</td>
</tr>
</tbody>
</table>

During startup, both entries of the array `EE_resource_task[]` are initialized to -1 denoting that no task has currently requested or is using any resource. Every time a resource access is made (local or global) irrespective of whether or not the resource is available, the variable `EE_resource_task[]` is updated with the Resource ID of the resource at index 0 and the TaskID of the requesting task at index 1. This mechanism is suitable only for spin-lock priorities in the range [CP, HP]. For other spin-lock priorities, there can be more than one pending global resource request and the mechanism will not work. The values are reset to -1 upon the release of the resource. Hence, when a core receives a dedicated interrupt, the information about the task that requested a resource is available locally without having to be communicated. Listings 4.2 and 4.3 show the updated code of `EE_oo_GetResource()` and `EE_oo_ReleaseResource()` respectively.
LISTING 4.2: Updated code of EE_oo_GetResource().

```c
StatusType EE_oo_ReleaseResource(ResourceType ResID) {
    EE_TID rq, current;
    EE_UREG isGlobal;
    register EE_FREG flag;
    extern int Preemption_took_place;

    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();
    // Indicate that the task has released the resource
    EE_resource_locked[ResID] = 0;
    EE_resource_task[0] = -1;
    EE_resource_task[1] = -1;

    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 (Preemption_took_place==1){
        Preemption_took_place=0;
        if (EE_th_ready_prio[EE_stkfirst] >= EE_th_ready_prio[rq] || 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] = RUNNING;
            EE_sys_ceiling |= EE_th_dispatch_prio[rq];
        }
        EE_hal_ready2stacked(EE_rq2stk_exchange());
    }
    EE_hal_end_nested_primitive(flag);
    return E_OK;
}
```

LISTING 4.3: Updated code of EE_oo_ReleaseResource().

Sending core

The implementation of EE_di_send() function is detailed here. The EE_di_send() function is used for sending an interrupt to a remote core \( P_y \) from a core \( P_x \) \((x \neq y)\) indicating the release of a global resource \( R_q \). The function prototype is as follows:

\[
\text{int } \text{EE_di_send(EE_TID rtid)}
\]

<table>
<thead>
<tr>
<th>Argument 1</th>
<th>rtid</th>
<th>TaskID of the remote task that needs to be notified regarding the resource release.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns</td>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Otherwise</td>
</tr>
</tbody>
</table>

Figure 4.7a illustrates the execution flow of the function EE_di_send(). The functions called by EE_di_send(), i.e. the calls graph is shown in Figure 4.7b. It can be observed that there are no shared data structures and mutual exclusion involved in sending an interrupt. The function, as can be seen from Figure 4.7a, is executed with incoming interrupts disabled.

The function EE_di_send() is called during resource release from EE_hal_spin_out() function as shown in Figure 4.6. The pseudo code and the C-code of the function EE_di_send() are given in Listings 4.4 and 4.5 respectively.
4.3. Dedicated Interrupts

LISTING 4.4: Pseudo code for EE_di_send() function.

```c
int EE_di_send(EE_TYPERN rtid)
{
    register EE_UINT8 cpu;
    register EE_FREG flag;

    flag = EE_hal_begin_nested_primitive();
    //Find the core to which the remote task belongs
    cpu = EE_rn_cpu[rtid];
    if (cpu == EE_CURRENTCPU) {
        // THIS SHOULD NEVER HAPPEN Local notification not allowed
        EE_hal_end_nested_primitive(flag);
        return 1;
    } else {
        EE_hal_IRQ_interprocessor(cpu);
    }
    EE_hal_end_nested_primitive(flag);
    return 0;
}
```

LISTING 4.5: C-code for EE_di_send() function.

Receiving core

The function EE_di_handler() is registered on each core as the interrupt handler function. The input interrupt line for every core is specified in the CONF.OIL file and it is assigned to the function EE_nios2_IRQ_handler() during startup. Hence it serves as the ISR associated with all the dedicated inter-core interrupt lines shown in Figure 4.4.

```
void EE_di_handler()
{
    Arguments | None
    Returns   | void
}
```
The flow chart and the calls graph for the function `EE_di_handler()` are shown in Figures 4.8a and 4.8b respectively. The processed interrupt is marked as served with interrupts disabled as shown in the figure. The task that requested the resource which has been recently released is retrieved directly from the variable `EE_resource_task[]` as seen in Line 8 of Listing 4.6. The function `EE_di_execute()` is called with the interrupts enabled. This may be done to ensure that no new interrupts are missed. Since there can only be one global resource request pending on a core under FSLM spin-lock priorities [CP, HP], no new interrupts will arrive until the completion of the current global critical section. Thus, having the interrupts enabled while calling `EE_di_execute()` will have no negative effects. However, as can be seen from Listing 4.7, the requesting task is executed with non-preemptive system ceiling and hence cannot be preempted during its global critical section. The C-code for the function `EE_di_handler()` is presented in Listing 4.6.

Listing 4.6: C-code for `EE_di_handler()` function.

```c
void EE_di_handler(void)
{
    EE_hal_IRQ_begin_primitive();
    EE_hal_IRQ_interprocessor_served(EE_CURRENTCPU);
    EE_hal_IRQ_end_primitive();
    EE_di_execute(EE_resource_task[0], EE_resource_task[1]);
}
```

Figure 4.8: The EE_di_handler() function.

The `EE_di_execute()` function raises the system ceiling to non-preemptive level and resets the `Spin_lock` variable to 0. The function also performs the context switch into the task requesting the global resource if the task had been previously preempted while spinning. The function prototype for the `EE_di_execute()` function is presented below:

```c
void EE_di_execute(ResourceType ResID, EE_TID requesting_task)
{
    //Check if the resource ID and task ID are valid
    if(ResID != 0xff && requesting_task != -1){
        //Check if the resource ID and task ID are valid
        if(ResID != 0xff && requesting_task != -1){
```
4.3. Dedicated Interrupts

The component functions of the Dedicated Interrupts mechanism were constructed based on the existing Remote Notification method offered by Erika Enterprise. The set of changes in DI when compared to RN are investigated, and the validity of the changes are established by means of simple textual proofs and functional verification.

Figures 4.10a and 4.10b show the flow charts of EE_rn_send() and EE_rn_handler() functions respectively while also highlighting using extended boxes, the parts that are not included in the DI method. This figure is for illustration purposes only. The function call to EE_di_execute() in DI handler occurs at the end (Figure 4.8a) unlike in RN. The significant differences between the two methods in terms of implementation can be summarized as follows:

1. RN uses the shared memory based RN message buffers for communicating the additional information about the nature of the interrupt. DI does not use shared memory since it is not a generic communication method like RN.

2. RN supports multiple pending requests while processing in order to allow for different types of actions. Since DI is dedicated only to signal resource release, the implementation is only required to handle at most one incoming interrupt at a time (See proof for Predicate 2).

Check Invariant and Predicates

The design is validated by checking the invariant and predicates defined in Sections 4.3.2.
Invariant 1. During startup, on each core in the system, the spin-lock is unlocked, which implies that the number of received DIs is equal to the number of processed DIs.

By design, DIs can be sent only from within the EE_oo_ReleaseResource() function. The function EE_oo_StartOS that is executed during the system startup does not contain calls to the EE_oo_ReleaseResource() function and hence, during startup, the number of received and processed DIs on all cores in the system is equal to zero.

Predicate 1. A core will receive a DI only if the local spin-lock variable is locked.

Upon every global resource request, the only place where the variable spin_lock on the core is set to 1 if the requested resource is found to be unavailable is within the EE_hal_spin_in() function. Following the setting of the spin-lock, the TailQ[] is updated. This mechanism as described in Figure 4.3a is responsible for indicating the target core to the core holding the resource for sending a DI. In the function EE_hal_spin_in(), the two steps occur in sequence, thereby denoting a bi-implication relation between the setting of the spin-lock and expecting a DI.

Predicate 2. The number of received DIs on a core can at most be only one greater than the number of processed DIs on the core, for spin-lock priorities in [CP, HP].

A proof by contradiction can be provided to establish this predicate. Assume that the number of received (unprocessed) Dedicated Interrupts (DI) is 2 or more greater than the number of processed DIs on a core with spin-lock priority in [CP, HP]. This signifies that either one remote core has sent 2 or more DIs to the receiving core (Case (i)) or two or more remote
4.3. Dedicated Interrupts

cores have sent DIs to the receiving core (Case (ii)). The two cases are examined further:

Case (i) Assume that 2 or more DIs are sent from the same core. For a given global resource \( R_q \), this can only happen if the sending core releases \( R_q \), accesses it again and subsequently releases it for the second time. However, since the resource requests are FIFO ordered, when the sending core releases \( R_q \) for the first time and sends a DI to the receiving core, it implies that before the sending core can access \( R_q \) again, it can only occur in the \( ResourceQ \) after the receiving core. For 2 or more different global resources released consecutively by the sending core, the receiving core can at most occur in only one of the \( ResourceQ \) since Lemma 12 in [2] proves that there can only be at most one task on a processor with a pending request to a global resource at any time for spin-lock priorities CP and above. Thus, in combination with predicate 1, it can be observed that a new DI cannot be expected until the previous DI has been processed and the spin-lock variable is locked again.

Case (ii) Assume 2 or more cores send DIs to the same core simultaneously. From Lemma 12 of [2], it is clear that all the sending cores must send DIs for the same global resource. This is impossible since by definition, at any given instant, at most only one task in the system can access a resource (mutual-exclusion). Also, resource access is carried out non-preemptively, thereby making the scenario where multiple cores send DIs regarding the same resource to the same core impossible.

This predicate establishes that at any given instant, there can never be 2 or more pending interrupts notifying the release of resources. In other words, under the condition that the spin-lock priority is in \([CP, HP]\), a DI can never occur while a core is already processing a previously received DI. This serves to justify the removal of the check present in RN mechanism for a new interrupt while a core is handling a previously received interrupt as illustrated in Figure 4.10b.

Functional verification

A functional verification of the newly implemented Dedicated Interrupts mechanism was also performed. The test environment used for the functional verification is detailed below:

<table>
<thead>
<tr>
<th>Test environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cores</td>
</tr>
<tr>
<td>Number of tasks</td>
</tr>
<tr>
<td>Number of global resources</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task ID</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>Task 0 (( \tau_0 ))</td>
</tr>
<tr>
<td>Task 1 (( \tau_1 ))</td>
</tr>
<tr>
<td>Task 2 (( \tau_2 ))</td>
</tr>
<tr>
<td>Task 3 (( \tau_3 ))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Spin-lock priorities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 0 (( P_0 ))</td>
</tr>
<tr>
<td>Core 2 (( P_2 ))</td>
</tr>
</tbody>
</table>

All the tasks are manually triggered using peripheral I/O buttons. The task \( \tau_0 \) with resource access consists of a long global critical section (gcs) preceded and succeeded by a non-critical section (ncs) whereas task \( \tau_1 \) only consists of a long gcs. Blinking of respective LEDs indicates the execution of tasks with a faster blinking blinking rates denoting the execution of gcs. All ncs and gcs lengths are sufficiently increased to enable manual monitoring of the execution status of the tasks. The test cases that were executed to functionally verify the DI mechanism are listed in Table 4.1.
### 4.4 Performance comparison

The implementation of the Dedicated Interrupts method and Remote Notification method are compared in this section. A performance comparison between the component functions of Remote Notification and Dedicated Interrupts methods in terms of number of execution cycles is made. Altera performance counter cores were used since they involve minimum intrusion in the software for measurements. The performance counter cores [41] support high-resolution measurements in terms of clock cycles thereby providing a resolution of 20ns for a clock frequency of 50 MHz. The detailed experimental setup and software intrusions used in general for measurement process in this project are explained in Chapter 8 under Sections 8.1 and 8.2. The execution times of the Remote Notification functions `EE_rn_send()`, `EE_rn_handler()` and `EE_rn_execute()` are measured while using the initial implementation of FSLM [3] [4]. In the new implementation of FSLM, the execution times of functions `EE_di_send()`, `EE_di_handler()` and `EE_di_execute()` are measured.

The implementation of `EE_di_execute()` (originally that of `EE_rn_execute()` in [4]) has been performed for spin-lock priorities in [CP, HP]. The context switch into the task requesting the resource for the execution of the global critical section is performed by the interrupt handler. Thus, the measurement of execution time of `EE_di_execute()` also includes the execution time of the context switch. Since the completion of context switch will result in the beginning of execution of the global critical section, the context switch execution time is calculated separately (Chapter 8.3.1) and added to the rest of the execution time of `EE_di_execute()`.

Table 4.2 lists average execution time, minimum and maximum values along with the standard deviation of the measured values. The experiments for the measurements were performed on 3 cores with 25 samples from each core. For consistency and ease of comparison, experiments were performed such that the execution time of `EE_rn_execute()` and `EE_di_execute()` always account for the context switching block shown in Figure 4.9. Figure 4.11 shows a plot of the measured values on 3 different cores obtained for all the DI and RN component functions.

<table>
<thead>
<tr>
<th>Test case description</th>
<th>Expected result</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Establish that all tasks work under no contention, 1. Press button 0 - wait for the task to complete (τ₀ on core 1), 2. Press button 1 - wait for the task to complete (τ₁ on core 1), 3. Press button 3 - wait for the task to complete (τ₃ on core 2)</td>
<td>All tasks are executed according to the design: 1. LED 0 blinks 30x 2. LED 1 blinks 10x 3. LED 3 blinks 10x</td>
<td>PASS</td>
</tr>
<tr>
<td>Task without resource access on one core runs in parallel with task with resource access on another core. 1. Press button 3 - τ₃ on core 2 begins running 2. Before τ₂ on core 2 completes, press button 1</td>
<td>Both tasks run in parallel.</td>
<td>PASS</td>
</tr>
<tr>
<td>Task with resource access on one core gets blocked when other core has the resource. 1. Press button 3 - τ₃ on core 2 begins running 2. Before τ₁ on core 2 completes, press button 0</td>
<td>τ₀ on core 1 is blocked after ncs executes its gcs after the completion of gcs of τ₁ on core 2.</td>
<td>PASS</td>
</tr>
<tr>
<td>Test for full feature (activation of remote blocked task that has been preempted when the resource becomes available): 1. Press button 0 - the ncs of τ₀ on core 1 starts execution 2. Before completion of ncs, press button 3 - the critical section of τ₃ on core 2 starts execution and acquires the resource. 3. When the gcs of τ₃ is almost complete (Last 3 blinks), press button 2 - τ₁ on core 1 preempts the spinning task and starts execution</td>
<td>When the gcs of τ₁ completes and the resource becomes available, τ₀ on core 1 is preempted and τ₀ begins the execution of its gcs. On completion of the gcs, τ₀ resumes and runs to completion, followed by the remaining ncs of τ₀.</td>
<td>PASS</td>
</tr>
</tbody>
</table>

#### Table 4.1: Functional verification of DI mechanism - Test cases.

<table>
<thead>
<tr>
<th>Send</th>
<th>Handle</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EE_rn_send()</code></td>
<td><code>EE_di_send()</code></td>
<td><code>EE_rn_handler()</code></td>
</tr>
<tr>
<td>Average Execution time (cycles)</td>
<td>662</td>
<td>98</td>
</tr>
<tr>
<td>Max</td>
<td>701</td>
<td>104</td>
</tr>
<tr>
<td>Min</td>
<td>623</td>
<td>90</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>24.94</td>
<td>5.79</td>
</tr>
</tbody>
</table>

#### Table 4.2: Comparison of execution times of component functions of Remote Notification (RN) and Dedicated Interrupts (DI) for 75 samples (25 per core).
4.4. Performance comparison

As can be seen from Table 4.2 and Figure 4.11, there is a significant improvement in terms of performance between the RN and DI methods with DI outperforming RN. The average execution time of the interrupt handler has reduced by almost 97% since there is no shared memory based messages to decode unlike RN since all information required to process the incoming interrupt is now stored and retrieved locally from the variable EE_resource_task[]. The variations in the execution times of RN functions can mainly be attributed to the shared memory RN message buffers. The DI execution times are also found to fluctuate to a small extent for each pair of origin and destination cores. However, for a given pair of origin and destination cores, the execution times are constant.

The advantages and disadvantage of employing the DI method over the existing RN method is presented:

Advantages

1. The component functions of DI present a significant reduction in execution time over the corresponding component functions of RN mechanism. The send, handle and execute functions of DI result in 85.19%, 97.03% and 39.01% decrease in the average execution times respectively over their corresponding RN counterparts.

2. The execution times of the DI component functions are found to be less dispersed when compared to RN as evidenced by the standard deviation values in Table 4.2. Both the reduction in execution time and the tight grouping of the measured values in DI can be partly attributed to the lack of shared memory related components as highlighted in Figures 4.10a and 4.10b.

3. In summary, for spin-lock priorities in [CP, HP], the DI method outperforms RN.

Disadvantages

1. For spin-lock priorities less than CP, the DI method fails to work because Predicate 2 will no longer hold. This is because more than one pending global resource request can exist simultaneously on a core.

---

The graph shows the execution times measured on 3 cores for 25 repetitions per core.
4.5 Compatibility for spin-lock priorities in [LP,CP)

The initial [3][4] and new implementations of FSLM have been created as a proof of concept on the Erika Enterprise RTOS while only considering spin-lock priorities in the range [CP, HP]. As a result, few design choices have been made with the idea of optimizing the performance of the software for only for [CP, HP] on the specific platform of choice (Altera Nios II). This section explains a list of such design choices which might not be suitable when spin-lock priorities in [LP,CP) or other platforms are also considered.

1. For enabling non-preemptive execution of global critical sections, the system ceiling must be raised to a non-preemptive level. In the initial implementation [4], it is assumed that there can be at most 8 separate priorities on the core. Hence, an 8-bit value of 0x80 (1000 0000 in binary) is chosen as the non-preemptive system ceiling. Thus, the implementation will likely not work for platforms with 16 or 32 bit priorities and hence, the value of non-preemptive system ceiling should be increased to 0x0800 or 0x8000 respectively.

2. In both the initial and new implementations, the interrupt handlers EE_rn_handler() and EE_di_handler() respectively are designed for spin-lock priorities in [CP, HP] under the assumption that Predicate 2 always holds. Thus, the handler no longer keeps track of new incoming interrupts while executing an already received interrupt unlike the native RN handler implementation in Erika. However, when spin-lock priorities in [LP,CP) are also considered, Predicate 2 no longer holds and the handler will likely fail to work as expected.

3. The EE_rn_execute() and EE_di_execute() functions in the initial and new implementations respectively are also specifically designed for spin-lock priorities in [CP, HP]. As a result, the context switch into the task requesting the global resource for the execution of its global critical section is made by the interrupt handler. This also assumes that Predicate 2 holds and no other interrupt will be received until the completion of execution of the global critical section. Since nested access of global resources are not allowed in FSLM this design choice was made in [4]. However, when other spin-lock priorities in [LP,CP) are considered, this system would likely fail to work as expected.

4. Since RN message buffers are no longer used, a new local data structure EE_resource_task[] has been introduced in DI method. Although this could also be achieved by using a single unprotected global data structure, since on different platforms, the access times for global memory might be significantly longer than for local memory, the choice was made to use local memory for storing the data structure EE_resource_task[].

4.6 Recommendations for integration with Erika Enterprise

The presented DI implementation in Erika Enterprise has been performed as a proof of concept. However, it might be desirable to extend Erika Enterprise in the future with full featured support for FSLM and consequently the DI mechanism. The possible options for integrating the DI mechanism with the full featured version of Erika Enterprise are investigated in this section.

In the implementation created for this project, the Erika Enterprise kernel has been significantly reduced to merely support the implementation of FSLM. As a result, since none of the original RN message types supported by the native Erika Enterprise in Listing 4.1 are used for FSLM, the RN feature was completely replaced by the DI mechanism which uses the same inter-core interrupt lines originally used by RN. However, for a full featured version of Erika Enterprise, there might be a requirement for the features of FSLM including DI to co-exist with other default RN message types, potentially used for non-RAP related parts of the application. Also, when FSLM spin-lock priorities in [LP,CP) are considered in the future, the co-existence of DI and RN might be desired as DI does not support spin-lock priorities less than CP. Thus, an investigation is performed to determine the possible options and presented as recommendations.

Based on the implementation of DI and the similarities between the component functions of RN and DI methods, the following options are most likely to be considered for the integration of the DI mechanism into Erika Enterprise. The features of both the options are briefly explored.

- Option 1: Implement DI as a special case of RN mechanism.
• Option 2: Implement DI as a stand alone inter-core communication mechanism in addition to RN.

<table>
<thead>
<tr>
<th></th>
<th>Option 1</th>
<th>Option 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of inter-core interrupt networks</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Modifications to OIL and RT-Druid</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Use of shared memory</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Execution time</td>
<td>RN &gt; Option 1 &gt; DI</td>
<td>RN &gt; Option 1 &gt; (Option 2 = DI)</td>
</tr>
<tr>
<td>Predictability of execution time</td>
<td>Low</td>
<td>High</td>
</tr>
</tbody>
</table>

Table 4.3: Comparison of DI integration options.

**Option 1: DI as a special case of RN**

From the implementation of DI, it can be observed that the DI mechanism can be viewed possibly as a special case of RN mechanism in which the shared memory based messages are no longer used. Thus, for the full featured implementation of FSLM on Erika Enterprise, Dedicated Interrupt may be declared as a new RN type under which in both the sending and receiving cores, a majority of the shared memory related messages are skipped. The features of this option are listed below:

1. The existing inter-core interrupt lines are sufficient since DI is now a special message type of RN.
2. No additional configuration options are required in the OIL file and hence no changes are required in the RT-Druid code-generation plug-ins.
3. However, defining DI as a special message type of RN still requires the use of shared memory, although to a smaller extent. Remote Notification mechanism uses unique identifiers (Listing 4.1) to classify the type of notification. This implies that DI message type would carry a unique identifier as well. Hence, the sending core needs to write the notification type in the RN data buffer of the receiving core.
4. The execution times for sending and handling a received notification would significantly increase since the RN shared message buffers would still be used, requiring mutual exclusion. This also would result in the increase in the dispersion of the execution times.

**Option 2: Implementing DI as a new inter-core communication mechanism**

This option suggests the implementation of DI as a new mechanism while RN remains for the original notifications listed in Listing 4.1. The features of this option are as follows:

1. A new inter-core interrupt network needs to be created in the hardware design. Thus, the system would contain two inter-core interrupt networks to be used by DI and RN independently.
2. The interrupt network of DI needs to be defined with a priority higher than that of the interrupt network used by RN. This is to ensure that there are no additional blocking overheads contributing to the schedulability analysis of the RAP, i.e FSLM. Although the chosen hardware platform allows for the existence of two sets of inter-core interrupts, support for using the two networks must still be added in the software.
3. The CPU_DATA section for each core on the OIL file would require a new attribute to specify the base address of the new DI hardware interrupt line. RT-Druid code-generation plug-ins have to be updated to incorporate the new system of interrupts and assign it to the DI mechanism in the operating system kernel.
4. Since the stand alone DI mechanism no longer uses the shared memory unlike Option 1, the execution times of the sending and handling functions will be smaller and less dispersed, equal to the measurements presented in Table 4.2 for DI.
Table 4.3 summarizes the comparison between the two options for integration of DI in Erika Enterprise. The execution times of RN and DI presented in Table 4.2 are used to provide an estimate for the execution times of the two explored options. It can be observed that although Option 2 requires an additional set of inter-core interrupt lines and minor modification to the code-generation plug-ins, the performance is expected to be significantly better than Option 1.
Chapter 5

Dual shared stacks

In this chapter, the runtime stack memory utilization for the selected range of spin-lock priorities [CP, HP] in FSLM is explored. A method of using two stacks per core for restricted spin-lock priorities is proposed and a proof of concept is created in the Erika Enterprise implementation of FSLM.

5.1 Runtime stack space

Stack space of a task in a system with support for multiple tasks can be defined as the reserved memory space for the storage of the task context, i.e., the contents of the CPU registers [16]. In conventional operating systems, to enable seamless support for multiple tasks with interleaved task executions and preemptions, each task in the system typically has its own private predefined stack space. A potential drawback of this method is the fact that with an increase in the number of tasks, significant amount of memory is required for the stacks of all the tasks. This could be problematic in the case of real-time embedded systems since such systems are usually resource constrained and memory resources might be at a premium.

Sharing of a single runtime stack by multiple tasks has been proposed previously in works such as SRP [15] and MSRP [5]. A stack incorporates two functions namely push and pop. A push operation adds an element to the top of the stack. An element in this case refers to the context of a task. A pop operation removes the most recently added element from the top of the stack. Thus, a stack is Last In First Out (LIFO) ordered. An element that is popped from a stack must be immediately used by the active task to prevent the loss of the popped element. Since the pop operation removes the most recently added element, it is only possible to resume a task that is at the top of the stack. As a consequence of the above properties of a stack, two or more tasks with interleaving executions cannot inhabit the same shared stack.

Figure 5.1: FSLM single stack evolutions under HP and CP approaches.
Figure 5.1a shows a sample evolution of tasks on a core using a shared stack space under the FSLM-HP approach. The concept of interleaved executions among tasks sharing a common stack space leading to stack inter-penetration is illustrated in Figure 5.1b. At time $t = 5$, the spinning task $\tau_0$ is preempted by $\tau_1$. The top of the stack now contains the context and CPU registers of task $\tau_1$. At time $t = 8$, $\tau_1$ is preempted by $\tau_2$ thereby placing the context of $\tau_2$ at the top of the stack. At time $t = 11$, the resource access is granted to $\tau_0$ which must immediately begin the execution of its global critical section. However, the context of the task $\tau_0$ is not at the top of the stack as shown in Figure 5.1b. This scenario results in an interleaved execution of tasks and hence showing that for a spin-lock approach less than HP, a single shared stack space per core is insufficient.

5.2 Stack requirement - FSLM

This section presents an argument regarding the maximum number of stacks required per core for FSLM. Ignoring the implementation overheads, the HP spin-lock approach of FSLM is similar to MSRP. It has been shown in [5] that using spin-lock priority equal to $\pi^\text{max}_P$, i.e HP spin-lock priority allows using a single stack for all tasks in the core. The authors of [7] present a theorem using a series of lemmas to show that it is sufficient to use two stacks on each core for all spin-lock priorities in the range $[\text{CP}, \text{HP})$.

A brief recap of the theorem and the set of lemmas that support the proof of the hypothesis presented in [7] is as follows:

**Lemma 1.** The local resource sharing protocol SRP ensures that once a job is started, it cannot be blocked due to a local resource until completion; it can only be preempted by higher priority jobs [15].

**Lemma 2.** For any task $\tau_i \in T_P$ where $\pi_i \leq \pi^\text{spin}_P$ a job of lower priority task which is preempted by a job of $\tau_i$ cannot execute until the job of $\tau_i$ is finished.

**Lemma 3.** For any two tasks $\tau_i \land \tau_j \in T_P$ where $\pi^\text{spin}_P < \pi_i, \pi_j < \pi^\text{max}_P$ a job of $\tau_j$ which is preempted by a job of $\tau_i$ cannot execute until the job of $\tau_i$ is finished.

The above lemmas are used in [7] to establish the following theorem:

**Theorem 1.** For a spin-lock priority $\pi^\text{spin}_P$ in the range $[\pi^\text{CP}_P, \pi^\text{max}_P)$, FSLM allows the set of tasks on core $P_k$ to require only two stacks, i.e. tasks with a priority at most $\pi^\text{spin}_P$ share one stack and the other tasks share another stack.

A complete textual proof of the above theorem can be found in Section 4 of [7]. The theorem proves that while following the suggested stack allocation, there will be no interleaved execution of tasks sharing the same stack. This serves as a motivation for the implementation of dual shared stack for the implementation of FSLM in Erika Enterprise. The initial implementation of FSLM on Erika is presented in [4] and uses an individual stack for every task in a core.

5.3 Stack configuration options in Erika Enterprise

Erika Enterprise allows for the specification of RTOS objects before compilation. OSEK Implementation Language (OIL) [42] Language, a text-based description language, is used for the specification of the objects that exist in the application. The OIL file then serves as the input for RT-Druid [37], a dedicated tool-suite providing a system modeler, code-generator plug-ins for the open-source Eclipse framework [38] and schedulability analysis plug-ins. The OIL file is parsed by RT-Druid code-generator resulting in the generation and compilation of the application. This section presents a brief overview of the available options for configuring the runtime stack of the tasks in the multi-core extension of the “BCC2” conformance class of the OO (OSEK OS) kernel [39] of Erika Enterprise.

Static RTOS objects such as tasks and resources can be predefined at application compile time in the OIL file. To specify the stack related configuration parameters for the application, the following objects in the OIL file are used:

i) CPU object
ii) OS object

iii) Task object

The objects are used as containers for specifying the relevant attributes which can be numbers, strings or enumerations based on the nature of the attribute. The attributes can also be contained within so called “sections” within the objects. An exhaustive list of all the attributes and corresponding configuration parameters for all objects in the OIL file can be found in [37] and [42].

5.3.1CPU object

The CPU object in an OIL file is used as a container for all the other objects and it does not have any attributes. The OS object and Task objects are contained within the CPU object. It is to be noted that there is only one CPU object in an OIL file.

5.3.2OS object

The OS object contains the compilation parameters and the global configuration for the Erika Enterprise kernel. A non-exhaustive list of attributes contained within the OS object are listed below:

- Compilation attributes
- OSEK attributes
- Multi-core attributes
- Nios-II target attributes (specific to Altera Nios-II [27] target)
- CPU_DATA sections
- Kernel conformance class

Listing 5.1 shows a sample snippet from an OIL file for a Nios II based two-core system. The listing shows the CPU object and the OS object along with a few attributes specified under the OS object.

```c
CPU test_application {
    /* --- Begin OS object --- */
    OS EE {
        /* --- Begin Compilation attributes --- */
        CFLAGS = "-DALT_DEBUG -O0 -g";
        CFLAGS = "-Wall";
        LDFLAGS = "-Wl,-Map -Wl,project.map";
        ASFLAGS = "-g";
        LODEFS = "\";
        LIBS = "-lm";
        /* --- End Compilation attributes --- */
        /* --- Begin Nios II target attributes --- */
        NIOS2_SYS_CONFIG = "Debug";
        NIOS2_APP_CONFIG = "Debug";
        NIOS2_DO_MAKE_OBJDUMP = TRUE;
        MASTER_CPU = "cpu0";
        IPIC_GLOBAL_NAME = "IPIC_OUTPUT";
        NIOS2_PTF_FILE = "C:/altera/Projects/four_core_mar23/four_core_SOPC.ptf";
        NIOS2_MUTEX_BASE = "MUTEX_0_BASE";
        /* --- End Nios II target attributes --- */
        /* --- Begin CPU_DATA section - CPU 0 --- */
        CPU_DATA = NIOSII {
            ID = "cpu0";
            MULTI_STACK = TRUE;
            APP_SRC = "cpu0_main.c";
            APP_SRC = "shareddata.c";
            STACK_TOP = "__alt_stack_pointer";
            SYS_SIZE = 0x1000;
        }
    }
    /* --- End OS object --- */
}
```
With respect to the stack configuration, within the OS object, the CPU_DATA section is of interest and is explained further.

5.3.3 CPU_DATA section

In a typical multi-core system, the OS object contains a CPU_DATA section for every individual core in the system.

- The MULTI_STACK attribute in that section is used to define whether or not the system supports multiple stacks for the application tasks in that core. The attribute can be set to TRUE or FALSE. In the example Listing 5.1, for CPUs 0 and 1, the definition of the MULTI_STACK attribute can be found in lines 33 and 49 respectively.

- For Nios II based systems [27], the architecture allows for a default stack on each core named DUMMY_TASK. Erika Enterprise schedules the main() function on each core as a background task, named as “dummy” task on this DUMMY_STACK. As mentioned, the DUMMY_STACK is feature based on the architecture and hence is not explicitly defined as an attribute in the CPU_DATA section.

- The SYS_SIZE attribute is used to specify the total stack memory allocated to all the tasks in that core (total stack memory irrespective of shared or private stacks).
5.3.4 TASK object

The TASK objects are specified within the CPU object to configure the properties of individual tasks in the application. Similar to the OS object, it is desirable to primarily investigate the “stack” related attributes of the TASK object in this section while an exhaustive list of all attributes can be found in [37].

- TASK keyword marks the definition of a TASK object.
- CPU_ID attribute is used to specify the CPU (core) to which the task is mapped.
- APP_SRC attribute is used to specify the path the application source file containing the implementation of the task.
- PRIORITY specifies the priority of the task. Similar to the definition of task priority in the system model (Chapter 3.1), the priority of a task $\tau_i$ is greater than that of $\tau_j$ if $i > j$.
- RESOURCE is used to specify the resources (local and global) utilized by the task. A TASK object can contain multiple RESOURCE attributes.
- STACK attribute in the task object is used to specify if the task uses a dedicated private stack or a shared stack. The attribute can be set to PRIVATE or SHARED. Setting a private stack also allows for the specification of an individual stack size using the SYS_SIZE attribute. When tasks in a core are configured with SHARED stack, RT-Druid code-generator assigns the tasks to the default DUMMY_STACK on which the main() function is scheduled as a “dummy” task as explained in Section 5.3.3.

```c
TASK task2 {
    CPU_ID = "cpu0";
    APP_SRC = "task2.c";
    PRIORITY = 3;
    ACTIVATION = 1;
    SCHEDULE = FULL;
    AUTOSTART = FALSE;
    STACK = PRIVATE{
        SYS_SIZE = 0x100;
    };
};

TASK task3 {
    CPU_ID = "cpu1";
    APP_SRC = "task3.c";
    PRIORITY = 1;
    ACTIVATION = 1;
    SCHEDULE = FULL;
    AUTOSTART = FALSE;
    STACK = SHARED;
    RESOURCE = mutex1;
};
```

Listing 5.2: Example OIL file snippet - Task objects with shared and private stacks.

Listing 5.2 shows an example declaration of two tasks “task2” and “task3” allocated to cores “cpu0” and “cpu1” respectively. It can be seen in line 8 that “task2” is configured with a PRIVATE stack of size 256 bytes. Whereas, “task3”, as seen in line 18 is configured with a SHARED stack.

5.4 Dual shared stack for FSLM in Erika

The options available for configuring the application task stacks in Erika Enterprise were briefly described in the previous section (Section 5.3). This section presents a method for utilizing the available stack configuration options to create two stacks per core for the implementation of FSLM in Erika Enterprise. In order to leverage the existing configuration options, the section also briefly describes the code-generation flow of RT-Druid with a focus on the stack configuration options.

5.4.1 RT-Druid auto code-generation for stack configuration

The RT-Druid [37] code-generation plug-ins use the OIL file as the input to automatically generate the operating system configuration files specific to the application defined in the OIL file.
Figure 5.2 shows a partial overview of the automatic code-generation architecture of RT-Druid. It can be seen from the figure that the RT-Druid tool requires the conf.OIL file to generate core specific configuration files. RT-Druid generates 2 files namely eecfg.h and eecfg.c for each core in the system. A file named common.c is generated only for the MASTER_CPU in the system. According to [43], the role of the Master CPU is to initialize the shared data.

- **common.c** contains the definitions for macros related to the synchronization during startup and spin-locks in the HAL layer. This file is generated only for the Master CPU.
- **eecfg.h** contains the common defines for each CPU including the defines related to the task IDs, mutex definitions and other options pertaining to the attributes specified in the OIL file.
- **eecfg.c** contains the definitions and initialization of data structures related to the kernel, mutexes, alarms, counters and interrupts. This file also contains the stack definition for application.

**Data structures in the stack definition**

The following data structures are used in the stack definition of every core in the file eecfg.c.

- **__alt_stack_pointer** - The symbol used inside Altera Nios II system libraries as the initial stack pointer.
- **EE_hal_thread_tos[]** - An array containing the indices of the top of stack (tos) corresponding to each task in the core and the main() function. Index 0 always corresponds to the main() function that is scheduled as a background task.
- **EE_nios2_system_tos[]** - An array with the actual addresses of the tos pointers. Index 0 always refers to the tos of the DUMMY_STACK.
- **EE_hal_active_tos** - Denotes the currently active stack. Initialized to 0 always during startup.

The data structures are explained further by considering the following example scenario.
Example 3. Consider a core with four tasks $\tau_0$, $\tau_1$, $\tau_2$ and $\tau_3$. The task names in plain-text are “task0”, “task1”, “task2” and “task3” respectively. Tasks $\tau_0$ and $\tau_1$ are configured with the TASK attribute set to SHARED while $\tau_2$ and $\tau_3$ are configured with PRIVATE stacks.

Example auto-generated stack configuration - snippet from eecfg.c.

Listing 5.3 shows the auto-generated stack definition by RT-Druid tool for the system described in Example 3. It can be seen from the listing that the array $\text{EE}_nios2\_system\_tos[]$ consists of 3 addresses, denoting the starting address for each of the three stacks on the core. The total number of tasks in a core is one greater than the number of user-defined tasks to accommodate for the main() function as a background task. Hence, in the presented example, the number of tasks in the core is 5. The array $\text{EE}_hal\_thread\_tos[]$ in the listing has 5 elements, with each index denoting a task. The main() function is always denoted by index 0, followed by the user-defined tasks as indicated by the comments in the listing. The value at each index in the array $\text{EE}_hal\_thread\_tos[]$ denotes the corresponding stack (defined in $\text{EE}_nios2\_system\_tos[]$) to which the task belongs. Hence, in this example, it is can be seen that the 5 tasks (including the main() function) are allocated to 3 stacks, where tasks $\tau_0$ and $\tau_1$ share the default shared stack along with the main() function while tasks $\tau_2$ and $\tau_3$ are allocated private stacks.

5.4.2 Configuration for dual shared stack

It can be observed from the available stack configuration options in Erika Enterprise that all tasks with TASK attribute set to SHARED are allocated to the default stack (DUMMY_STACK). To create another shared task, one of the following options exist:

1. Modify the RT-Druid tool by adding additional keywords for the TASK attribute to auto-assign tasks to two shared stacks.

2. Leverage the existing tooling and modify the auto-generated eecfg.c files to create two shared stacks on each core.

Since the FSLM implementation on Erika Enterprise is currently intended as a proof of concept, it is desirable to select an approach which retains the native tooling and code-generation plug-ins. Considering that any modifications to the RT-Druid tool chain would require extensive assistance from Evidence S.r.l., the latter option of modifying the auto-generated code with the existing RT-Druid code-generation plug-ins was chosen.

Implementing dual shared stack

In order to leverage the existing single shared stack per core support in Erika Enterprise in the process of creating dual shared stack, the following method is employed:

- Since the tasks with priorities at most $\pi_{\text{spin}}^P$ for spin-lock priorities in [CP, HP) can share one stack, their STACK attributes are set to SHARED. This ensures that these tasks are automatically assigned to the default common shared stack (DUMMY_STACK).
• Tasks with priorities higher than $\pi_{PK}$ are defined with STACK attribute set to PRIVATE, resulting in the auto-generation of individual stacks for this group of tasks.

• After the completion of automatic code-generation by RT-Druid, the stack properties of the group of private tasks in the eecfg.c file are modified such that a second shared stack is created.

```c
extern int __alt_stack_pointer;
#define __RTD_SYS_STACK_ADDRESS (int)(__alt_stack_pointer)

EE_UREG EE_hal_thread_tos[EE_MAX_TASK+1] = {
    0, /* dummy*/
    0, /* task0*/
    0, /* task1*/
    1, /* task2*/
    1 /* task3*/
};

struct EE_TOS EE_nios2_system_tos[2] = {
    {(EE_ADDR)(__RTD_SYS_STACK_ADDRESS )},
    {(EE_ADDR)(__RTD_SYS_STACK_ADDRESS- 3584 -36 )}
};

EE_UREG EE_hal_active_tos = 0; /* dummy */
```

Listing 5.4: Example modified dual stack configuration.

Listing 5.4 shows the modifications made to the auto-generated stack configuration of Example 3. The set of generic changes that have to be made to the stack definition part of eecfg.c are listed below:

- **EE_nios2_system_tos[]** - Only the stacks at indices 0 and 1 are retained in this array. This ensures that the resulting system contains one additional stack apart from the default DUMMY_STACK.

- **EE_hal_thread_tos[]** - The tos indices for all tasks with values 2 and above are changed to 1. This indicates that all tasks with priority higher than $\pi_{PK}$ now use the second stack while the rest of the tasks along with the main() function have already been auto-assigned to the default shared stack (DUMMY_STACK).

By using this method, it is trivial to observe that for spin-lock priority $\pi_{PK}$ equal to HP, all the tasks will be allocated the default DUMMY_STACK.

### 5.4.3 Memory size calculation for dual shared stacks

The modification of stack configuration to accommodate all the tasks into two stacks does not yet take into account the size of memory allocated to the two shared stacks. It is desirable to observe the method in which stack sizes are allocated in the native implementation of Erika Enterprise.

Figure 5.3a shows the stack allocation for the tasks in “cpu0” whose stack configuration is as described in Example 3 and Listing 5.3. The following observations can be made:

i) The complete size for all stacks in the core is as specified by the SYS_SIZE attribute under the CPU_DATA section of the OS object in the OIL file.

ii) Each task with a PRIVATE stack has a value for SYS_SIZE defined under the TASK object. A private stack of the specified size is allocated as a part of the overall stack memory space towards the end as shown in Figure 5.3a.

iii) The space left after allocation of all private stacks as specified in step (ii) is used as the default DUMMY_STACK to which all tasks with SHARED stack attribute and the main() function are allocated.

Modifying the auto-generated configuration files as described in Section 5.4.2 results in the stack allocation illustrated in Figure 5.3b. It can be observed that the second shared stack is created by removing all the top of stack pointers of the private stacks except for that of the first private stack. This results in the existence of only two stacks.
5.4. Dual shared stack for FSLM in Erika

**IRQ stack space**

Similar to the stack space requirement for tasks in a system, the Interrupt Requests (IRQs) also require runtime memory. In Erika Enterprise, setting the `MULTI_STACK` attribute to `TRUE` also allows for the setting of another attribute named `IRQ_STACK`. Setting the value of the attribute `IRQ_STACK` to `TRUE` specifies that the IRQs are executed using their own dedicated stack space. The size of the dedicated IRQ stack can then be specified using the `SYS_SIZE` attribute.

```c
MULTI_STACK = TRUE {
    IRQ_STACK = TRUE {
        SYS_SIZE = 0x200;
    };
};
```

**Listing 5.5**: OIL file snippet showing `IRQ_STACK` configuration.

Listing 5.5 shows an example declaration of a dedicated IRQ stack in the OIL file. However, unless specified, the default value of `IRQ_STACK` in Erika Enterprise is automatically set to `FALSE`. This signifies that the IRQs also utilize a part of the stack space allocated to the tasks. In a system with multiple tasks with each task having its own private stack and `IRQ_STACK` set to `FALSE`, every private stack consists of an additional IRQ stack space. The implementation of FSLM on Erika is configured without dedicated IRQ stacks, i.e., `IRQ_STACK` is set to `FALSE`.

**Calculating the stack memory requirement**

A method for calculating the appropriate stack sizes for the two stacks in a dual shared stack is presented. This calculation is presented to serve as a guide for setting the values of the `SYS_SIZE` attributes in the `CPU_DATA` section of the core as well as in the `TASK` objects such that the resulting shared stack sizes after modifying the `eecfg.c` file will be appropriate. The following set of terms are defined to aid in the memory requirement calculation:
Chapter 5. Dual shared stacks

- $\lambda_T^{P_k}$: Overall memory size for all stacks in a core $P_k$.
- $\lambda_{m}^{P_k}$: Stack space requirement for the main function of core $P_k$.
- $\lambda_{IRQ}^{P_k}$: Stack space requirement associated with IRQ.
- $\lambda_i$: Stack space requirement for task $\tau_i$ (without IRQ stack space).
- $\lambda_{S_0}^{P_k}$: Size of the first shared stack $S_0$ including IRQ space (tasks with $\pi_i \leq \pi_{spin}^{P_k}$).
- $\lambda_{S_1}^{P_k}$: Size of the second shared stack $S_1$ including IRQ space (tasks with $\pi_i > \pi_{spin}^{P_k}$).
- $n_{S_0}^{P_k}$: Number of tasks in stack $S_0$ of core $P_k$.
- $n_{S_1}^{P_k}$: Number of tasks in stack $S_1$ of core $P_k$.

The value of the memory size required for the first shared stack $S_0$ also includes the stack space required for handling IRQs and the default background task (main function). It is defined as:

$$\lambda_{S_0}^{P_k} = \lambda_{m}^{P_k} + \lambda_{IRQ}^{P_k} + \sum_{\forall i: \tau_i \in T_{P_k} \land \pi_i \leq \pi_{spin}^{P_k}} \lambda_i.$$  \hspace{1cm} (5.1)

Similarly, the value of the memory size required for the second shared stack $S_1$ is defined as:

$$\lambda_{S_1}^{P_k} = \lambda_{IRQ}^{P_k} + \sum_{\forall i: \tau_i \in T_{P_k} \land \pi_i > \pi_{spin}^{P_k}} \lambda_i.$$  \hspace{1cm} (5.2)

The overall memory size required on a core $P_k$ to accommodate all the tasks in the core including the default background task (main function) and IRQs is calculated as follows:

$$\lambda_T^{P_k} = \lambda_{S_0}^{P_k} + \lambda_{S_1}^{P_k},$$  \hspace{1cm} (5.3)

**OIL file configuration values**

By using the above equations, it is possible to determine the value of SYS_SIZE attribute under both the CPU_DATA section and the TASK object in the OIL file for every core.

The SYS_SIZE attribute under the CPU_DATA section of the OS object which specifies the total memory requirement for all stacks on the core $P_k$ is given by:

$$\forall P_k \in P \text{ SYS SIZE}_P_k = \lambda_T^{P_k},$$  \hspace{1cm} (5.4)

where $\text{SYS SIZE}_P_k$ denotes the SYS_SIZE attribute under the CPU_DATA section of core $P_k$ in the OIL file.

The cumulative size of all the private stacks of the tasks with priority greater than the spinlock priority on the core must be equal to $\lambda_{S_1}^{P_k}$. If $\text{SYS SIZE}_i$ denotes the SYS_SIZE attribute under the TASK object in the OIL file, for every task with priority greater than $\pi_{spin}^{P_k}$ this can be represented as:

$$\sum_{\tau_i \in T_{P_k}, \pi_i > \pi_{spin}^{P_k}} \text{SYS SIZE}_i = \lambda_{S_1}^{P_k}.$$  \hspace{1cm} (5.5)

By using (5.5), it is possible to calculate SYS_SIZE$_i$ for each individual task in the core with priority greater than $\pi_{spin}^{P_k}$. One possible definition for SYS_SIZE$_i$ can be:

$$\forall \tau_i \in T_{P_k}, \pi_i > \pi_{spin}^{P_k} \text{SYS SIZE}_i = \left\lceil \frac{\lambda_{S_1}^{P_k}}{n_{S_1}^{P_k}} \right\rceil.$$  \hspace{1cm} (5.6)
Alternatively, it is also possible to define \( \text{SYS}_{i} \) in any suitable way as long as the sum of all the private stacks of the tasks with priority greater than the spin-lock priority on the core is equal to \( \lambda_{P_{k}} \) as illustrated in (5.5). This ensures that when the second shared stack is created by merging all the privates stacks, the newly created shared stack will have sufficient memory to accommodate all the tasks and also the space required for an IRQ stack.

### 5.5 Memory savings associated with dual shared stack

This section investigates the savings in terms of memory space achieved while utilizing the dual shared stacks per core instead of the conventional method of individual stack per task.

In uni-processor context, SRP allows for the sharing of runtime stack. The use of so called “preemption levels” associated with every task apart from the priority of the task allows for larger stack saving with an increase in the number of tasks with the same preemption level [16]. Multiple tasks can share the same preemption level and two or more jobs sharing the same preemption level can never occupy a stack space at the same time. In multiprocessor context, fixed priority scheduling with preemption thresholds (FPTS) [44] is an example of such a method. Thus, in an example application consisting of 100 jobs each requiring 10Kbytes of stack distributed over 10 preemption levels [16], the overall stack space requirement would be 100Kbytes in case of a single shared stack. When compared to the overall stack requirement of 1000Kbytes if the tasks are allocated private stacks, the shared stack offers 90% saving in the memory requirement for this example.

In FSLM, the preemption levels are assumed to be equal to the priorities of the respective tasks. Since the priorities of all the tasks on a core must be unique, in the worst-case scenario, all the tasks could potentially occupy the same stack space simultaneously. Hence, for FSLM, the worst-case memory requirement for the stacks is equal to the sum of all the individual task stack requirements in the core. However, the IRQ stack requirement would significantly reduce in case of the dual shared stack implementation when compared to the private stacks method.

Let \( \lambda^\text{twc}_{P_{k}} \) represent the worst-case total stack memory requirement for core \( P_{k} \) while using private stacks for all tasks in the core. The value can be calculated as follows:

\[
\lambda^\text{twc}_{P_{k}} = \sum_{\forall i, \tau \in T_{P_{k}}} (\lambda_{i} + \lambda^\text{IRQ}).
\]

(5.7)

Now, by using the dual shared stacks, let \( \hat{\lambda}^\text{twc}_{P_{k}} \) represent the worst-case total stack memory requirement. Unlike the previous case, \( \lambda^\text{IRQ} \) needs to be accommodated only twice, i.e, once in each shared stack. The value can be calculated as follows:

\[
\hat{\lambda}^\text{twc}_{P_{k}} = (2 \times \lambda^\text{IRQ}) + \sum_{\forall i, \tau \in T_{P_{k}}} \lambda_{i}.
\]

(5.8)

The total savings in memory (\( \Delta \lambda_{P_{k}} \)) can be calculated as the difference between the above two terms:

\[
\Delta \lambda_{P_{k}} = \lambda^\text{twc}_{P_{k}} - \hat{\lambda}^\text{twc}_{P_{k}} = \max\{0, (|T_{P_{k}}| - 2)\} \times \lambda^\text{IRQ},
\]

(5.9)

where \( |T_{P_{k}}| \) denotes the cardinality of the set \( T_{P_{k}} \), representing the number of tasks in the processor \( P_{k} \). Hence, it can be observed that the reduction in memory requirement while using dual shared stack when compared to private stacks becomes more significant with an increase in the number of tasks in the core. The findings from this chapter were presented as a Work-in-Progress session paper [11].
Chapter 6

FSLM tool

In this chapter, a brief description of a utility tool named “Flex-Tool” created for automatically making the required configuration changes to FSLM implementation of Erika is presented.

6.1 Introduction

The native implementation of Erika Enterprise with MSRP uses the RT-Druid code-generator plug-ins to configure the operating system kernel based on the user inputs from the OIL file. However, for the implementation of FSLM on Erika, features such as changing the spin-lock priority of each core and modifying the stack information to accommodate dual shared stacks on each core also need to be configured based on the application. The changes for implementing FSLM have been made outside the context of RT-Druid code-generation plug-ins since this implementation is intended as a proof of concept. Therefore, in order to automate certain aspects of the RTOS kernel configuration, a new utility tool is developed. In this chapter, a brief description of the features of the tool is presented.

6.1.1 Spin-lock parameters in FSLM implementation

A set of parameters were newly defined as “extern” variables in the Erika kernel file to facilitate the implementation of FSLM. A subset of such variables that are required to be configured on each core during compilation are listed below:

- \( EE\_th\_spin\_prio \) - Array containing the value of the spin-lock priority for each task in the core. Hence the number of entries in this array is equal to the number of tasks mapped to the core. Since in FSLM, the same spin-lock priority is used for all the tasks in the core, all entries of this array will contain the same value.

- \( GlobalTaskID \) - Array containing a unique task ID for every task on the core. Number of entries in this array is equal to the number of tasks in the core. For all the cores in the system, the values contained in the array are unique. This global task ID is used to manage the FIFO ordering for global resource requests while using the variable \( ResourceQ \) as explained in Chapter 4.1.2.

The file \( ee\_internal.h \) in the Erika Enterprise kernel contains the declaration of the above variables. Listing 6.1 shows the snippet from \( ee\_internal.h \) containing the declarations.

```
/*spin-lock priority for each task in the core*/
extern const int EE\_th\_spin\_prio[];

/*unique global task ID for every task*/
extern const int GlobalTaskID[];
```

Listing 6.1: Snippet of “extern” configuration parameters from \( ee\_internal.h \).

The initialization of the listed variables are performed in the respective main files of each core in the application. Listing 6.2 shows an example snippet of initialization from one of the cores. The snippet shows a core with three tasks \( \tau_3, \tau_4 \) and \( \tau_5 \) with priorities 3, 4 and 5 respectively defined in the OIL file. The OIL file also contained the definition for two resources. However, Erika Enterprise internally converts the priorities \( \pi_i \) of the tasks in a core into \( 2^{\pi_i} \), starting from
Hence, the priorities of the tasks $\tau_3$, $\tau_4$ and $\tau_5$ are converted by RT-Druid while code-generation as 0x01 ($2^0$), 0x02 ($2^1$) and 0x04 ($2^2$) respectively. Thus the spin-lock priorities entered in the application file (user-defined) for the variable `EE_th_spin_prio[]` must also conform to the internal representation. Thus, the snippet shows the spin-lock priority set to 0x02, i.e, the priority of task $\tau_4$. The variable `EE_resource_task[]` is initialized with two variables for the two resources in the system.

```c
/*spin-lock priority for each task in the core*/
const int EE_th_spin_prio[] = {0x2,0x2,0x2};

/*unique global task ID for every task*/
const int GlobalTaskID[] = {3,4,5};

/*task to resource mapping during resource access*/
EE_TID EE_resource_task[] = {-1,-1};
```

LISTING 6.2: Snippet of example initialization of variables in `cpu1_main.c`.

Since these variables are initialized in the user-defined files, the values must be updated before the compilation of the application based on the configuration options found in the OIL file.

### 6.1.2 Stack memory parameters

The configuration of dual shared stacks for FSLM requires the modification of the auto-generated `eecfg.c` files on each core as explained in Chapter 5.4.2. In order for the `eecfg.c` files to be generated appropriately, the OIL file has to be updated with the proper stack settings for all tasks.

Thus, for the complete implementation of FSLM to work seamlessly, the modification of OIL file, user-defined application files and the auto-generated configuration files must be performed at different stages of the compilation and building process of the Erika Enterprise kernel. “Flex-Tool” is a utility tool that is created to automatically modify the above files at appropriate times during the build process to avoid complications for the user.

### 6.2 Requirements

The Flex-Tool is developed to merely assist the software build process for the FSLM implementation of Erika Enterprise. Although the development of this tool is not among the primary goals of this project, it is developed to automate the changes to the original build flow of native Erika Enterprise in order to accommodate FSLM in Erika without making changes to the original tool chain and makefiles provided by Evidence S.r.l (developers of Erika Enterprise). The high-level requirements of the Flex-Tool are listed below:

1. Parsing the OIL file
   (a) Identify and read the contents of the following objects in the OIL file.
      i) CPU_DATA sections
      ii) RESOURCE objects
      iii) TASK objects
   (b) Classify resources into local and global resources
   (c) Calculate $CP_1$, $\tilde{CP}$ and HP priority levels for each core.
   (d) Determine the stack allocation for dual shared stacks on each core based on user provided spin-lock priorities.

2. Updating the OIL file
   (a) Identify the STACK attribute under TASK objects for all tasks in the OIL file.
   (b) Update the value of STACK attribute as SHARED for all tasks in the core with priority less than or equal to the user provided spin-lock priority.

$\tilde{CP}$ denotes the highest resource ceiling of any resource (local or global) in a core.
6.3 Design and implementation

(c) Update the value of STACK attribute as PRIVATE for all tasks in the core with priority greater than the user provided spin-lock priority.

3. Updating the CPU source files
   (a) Identify the following variable initializations in each CPU source file in the application:
      i) EE_th_spin_prio[]
      ii) GlobalTaskID[]
   (b) Initialize the above variables to appropriate values.

4. Updating the eecfg.c files
   (a) Identify the following variable initializations in the auto-generated code of eecfg.c file for each CPU:
      i) EE_hal_thread_tos[]
      ii) EE_nios2_system_tos[]
   (b) Modify the above variables appropriately to create dual shared stacks.

5. User interaction
   (a) Display the CP, $\tilde{C}_P$ and HP priority levels for each core.
   (b) Accept user input for one spin-lock priority value per core.
   (c) Convert the user provided spin-lock priority values to Erika format$^2$.
   (d) Prompt user to initialize RT-Druid
   (e) Accept acknowledgement from user indicating the completion of RT-Druid code generation.
   (f) Prompt user to execute elf file generation using Eclipse build tools upon completion.

$C_P$ denotes the highest resource ceiling of any resource (local or global) in a core. Although this special spin-lock priority is not a focus of this thesis, it has been explored in the analysis of FSLM [7]. Hence, for historic reasons, it is calculated and presented to the user.

6.3 Design and implementation

The context of the Flex-Tool in the overall software build process of the FSLM implementation Erika Enterprise is illustrated in Figure 6.1. The figure illustrates the files that are modified by the Flex-Tool. It is to be noted that the modifications performed by Flex-Tool occur during different stages of the software build process and Figure 6.1 only shows the high-level context of the utility tool. The tool was built on Python 3.5.2.

6.3.1 Working

Figure 6.2 shows a sequence diagram depicting the actions performed by the various components in the build process. The tasks performed by the Flex-Tool during the various stages of the build process are illustrated in the figure. It can be seen that the activities of the Flex-Tool can be divided into three phases. A brief explanation is provided regarding the operation of the Flex-Tool based on the sequence diagram:

Phase 1

- When the tool is initialized, parsing of the OIL file is performed to extract the information regarding the CPUs, tasks and resources in the application.
- The task priorities and resource map is then used to compute the priorities $C_P$, $\tilde{C}_P$ and HP for each core.
- The tool then prompts the user to input the values for spin-lock priority within the calculated range for each CPU in the system.

$^2$Erika Enterprise internally uses $2^i$ where $i = 0, 1, ..., 7$ to denote priority of tasks
Phase 2

- The user provided spin-lock priorities are validated for correctness and converted to conform with the internal Erika Enterprise priority format.
- Based on the spin-lock priority value for each core, the allocation of tasks to dual shared stacks is also determined.
- The converted spin-lock priorities are then updated in the CPU source files. The data structures listed in Section 6.1.1 are also updated in the CPU source files.
- The `STACK` attribute of the `TASK` objects in the OIL file are modified such that the tasks with priority up to spin-lock priority are set to `PRIVATE` and the rest of the tasks are defined with `PRIVATE` stack.
- The Flex-Tool then prompts the user to initiate RT-Druid code-generation.

Phase 3

- The user indicates the completion of the RT-Druid code-generation operation following which the `eecfg.c` files for all CPUs have now been generated.
6.3. Design and implementation

The Flex-Tool now modifies the contents of the stack declaration section of each CPU in the respective `eecfg.c` file as described in Chapter 5.4.2.

The Flex-Tool finally indicates the completion of the file updates and prompts the user to generate the `Binary.elf` file for programming the FPGA using the Eclipse build tools provided by Erika.

6.3.2 Example execution scenario

An example execution of the Flex-Tool is illustrated in Figure 6.3. The screen-capture from the execution shows the different phases of execution. In “Phase 1”, the user is presented with the option to select spin-lock priorities for each core in the range [CP, HP].

An interesting effect of the Flex-Tool and the method of implementation of dual shared stacks can be observed in the “Phase 2” of the figure. The method described in Chapter 5.4.2 is used for the configuration of dual shared stacks. From the example in Figure 6.3, for CPU 0 and CPU 3, it can be seen that there are at least 2 tasks allocated to the second shared stack. Whereas, for CPU 1, there is only one task allocated to the second shared stack and in CPU 2, there are none. As a result, for CPU 1 and CPU 2, the stack configuration resulting out of the modification to OIL file in “Phase 2” is sufficient. The `eecfg.c` files generated by RT-Druid for CPU 1 and CPU 2 are not modified as can be seen in “Phase 3” of the screen-capture. Only for CPU 0 and CPU 3, the respective `eecfg.c` files are updated with appropriate settings to incorporate dual shared stacks.

6.3.3 Documentation

For the sake of brevity of the report and due to the fact that Flex-Tool was developed as an auxiliary tool to automate the build process during the project, implementation details of the tool have been moved to the Appendix.
The complete documentation for the Flex-Tool can be found in Appendix B. A user guide detailing the usage of the tool has also been provided. A web version of the detailed documentation can be found at https://srimuthu.github.io/fslm-tool-doc/.
Chapter 7

Overheads

The focus of this chapter is to provide a generalized method for identifying the implementation overheads related to Resource Access Protocols (RAPs) with focus on the original Erika Enterprise implementation of MSRP, older implementation of FSLM [3] and the new implementation of FSLM developed as a part of this study. The identified overheads are then measured on the hardware platform. The overhead components are then incorporated into the existing schedulability analysis of MSRP and FSLM with the intention to serve as a guideline for the choice of appropriate RAP for a specific application.

7.1 Classification of overhead scenarios

Runtime overheads related to the implementation of the RAP occur during resource request, the instant at which resource access is granted, and during resource release.

Figures 7.1a and 7.1b illustrate example time-lines showing the seven possible overhead scenarios in MSRP and FSLM respectively. The system illustrated in the figures consists of four tasks \( (\tau_0, \tau_1, \tau_2, \tau_3) \) mapped onto three processors \( (P_0, P_1, P_2) \) and a shared global resource. Task \( \tau_0 \) is allocated to \( P_0 \), \( \tau_1 \) and \( \tau_2 \) to \( P_1 \), and \( \tau_3 \) to \( P_2 \). Tasks \( \tau_0, \tau_1, \) and \( \tau_3 \) use a global shared resource.

In core \( P_1 \), the spinning priority of the core is less than the priority of the task \( \tau_2 \), meaning that \( \tau_2 \) can preempt \( \tau_1 \) while it is spinning on its resource request. The encountered overheads are classified and labeled in Figure 7.1 according to the phase at which they occur within the context of the RAP. A description of all the scenarios is provided below:

**Resource Request:**

There are two overheads scenarios associated with a resource request depending upon the availability of the resource at the time of request.

1. \( RQ_1 \) - Denotes the runtime overheads that occur if the resource request is made when the resource is available.

2. \( RQ_2 \) - Denotes the runtime overheads that occur if the resource request is made when the resource is not available, i.e. when the resource is held by a different task.

**Resource Access:**

The following scenarios \( AC_1, AC_2 \) and \( AC_3 \) can occur only when preceded by \( RQ_2 \). If the resource request was made when the resource is available \( (RQ_1) \), there are no measurable implementation overheads during resource access since the task is free to claim the resource immediately.

3. \( AC_1 \) - Resource access is granted when the requesting task is at the top of the stack, i.e. it has not been preempted. The task requesting the resource under this scenario is actively spinning.

4. \( AC_2 \) - Resource access is granted when the task requesting the resource is not at the top of the stack, i.e. it has been preempted while spinning by a higher priority task. Context switch has to occur in this scenario for the requesting task to execute the global critical section \( (gcs) \).

1 The numbers in the time-line are for reference purposes only. The relative sizes of overheads are not to scale.

2 Time-stamp 15 - illustrates the worst-case blocking context switch scenario \( AC_3 \).
5. $AC_3$ - Resource access can be granted at an infinitesimally small time $\epsilon$ after preemption occurs during spinning. The occurrence of a preemption begins with a context switch out of the spinning task into the higher priority preempting task. The context switch occurs non-preemptively and cannot be interrupted either. Hence, if the resource becomes available an instant $\epsilon$ after preemption, the interrupt handler and subsequently the resource access will be blocked until the context switch completes. Time-stamp 15 in Figure 7.1b on $P_1$ illustrates $AC_3$. The resource is granted to $\tau_1$ an instant $\epsilon$ after the preemption by $\tau_2$.

Resource Release:
Resource release scenarios can also be classified into two categories based on pending requests for the resource being released.

6. $RL_1$ - Resource is released when there are no pending requests for the same resource.

7. $RL_2$ - Resource is released when there are one or subsequent pending requests for the resource being released by tasks in other cores. Hence, the releasing core has to notify the
7.2 Identification of overhead components

The individual overhead components that contribute to the seven scenarios mentioned in Section 7.1 are identified in this section pertaining to the new implementation of FSLM developed for this study. The analysis of the implementation to identify the contributing overhead components resulted in the identification of seven such individual components.

7.2.1 Resource request overheads

The overhead components that contribute to scenarios \(RQ_1\) and \(RQ_2\) are identified and defined below:

**Def. 3.** \(OH_{RQ,G,adm}^{RQ,G}\) - Overhead associated with raising the system ceiling to non-preemptive priority and updating the administrative data structures (book keeping) upon a global resource request.

**Def. 4.** \(OH_{RQ,G,spn}^{RQ,G}\) - Overhead associated with updating the system ceiling to the spin-lock priority of the core when the resource is unavailable.

![Figure 7.2: Resource request overheads in FSLM (new implementation).](image)

Figure 7.2 illustrates the overheads \(OH_{RQ,G,adm}^{RQ,G}\) and \(OH_{RQ,G,spn}^{RQ,G}\) with respect to the new implementation of FSLM. It can be observed that \(OH_{RQ,G,spn}^{RQ,G}\) is contained in the conditional branches thereby including or excluding the overhead component based on the veracity of the condition. The while loop in Figure 7.2 denotes the actual spinning time where the core waits for the resource to become available, indicated by the spin-lock variable.

7.2.2 Resource access overheads

The overhead components that contribute to scenarios \(AC_1\), \(AC_2\) and \(AC_3\) are identified and defined below:
Def. 5. \( OH_{hdl} \) - Overhead associated with handling an incoming inter-core interrupt. This includes identifying the resource on which the spinning task on the incoming core is blocked, raising the system ceiling to non-preemptive priority and resetting the spin-lock upon the execution of the interrupt handler.

Def. 6. \( OH_{cs} \) - Context switching overhead incurred upon the receipt of an inter-core interrupt if the spinning task was preempted by a higher priority task.

Def. 7. \( OH_{bcs} \) - Blocking context switching overhead incurred if an interrupt from a remote core is received at an infinitesimally small time \( \epsilon \) after preemption by a higher priority task during spinning occurs, causing the interrupt handler to be blocked until the context switch completes.

Figure 7.3 illustrates the overheads \( OH_{hdl} \) and \( OH_{cs} \) with respect to the new implementation of FSLM. It can be noted that \( OH_{cs} \) is incurred only if a preemption had occurred while the task blocked on the global resource was spinning. Figure 7.4 illustrates an example scenario under which the overhead component \( OH_{bcs} \) appears. Example 4 illustrates the scenario in detail:

**Example 4.** Consider a two processor system consisting of processors \( P_0 \) and \( P_1 \) as shown in Figure 7.4. Task \( \tau_0 \) is mapped to \( P_0 \) while tasks \( \tau_1 \) and \( \tau_2 \) are mapped to \( P_1 \). Tasks \( \tau_0 \) and \( \tau_1 \) require access to the same shared global resource \( R_0 \). The task \( \tau_2 \) has a higher priority than \( \tau_1 \) and the spin-lock priority of \( P_1 \) is CP, i.e., the priority of task \( \tau_1 \). This indicates that \( \tau_2 \) can preempt \( \tau_1 \) while it spins. \( \tau_0 \) arrives at \( t = 0 \) and issues a resource request to \( R_0 \) at \( t = 2 \). Since the resource is available, \( \tau_0 \) proceeds to execute its global critical section (gcs) after incurring only \( OH_{RQ, adm} \) and \( OH_{RQ, spin} \) and starts spinning at CP priority level on \( P_1 \). At \( t = 8 \), task \( \tau_2 \) arrives, initiating the context switch out of the spinning task \( \tau_1 \). However, in the meanwhile, \( \tau_0 \) completes the resource access and sends a DI to core \( P_1 \) at an infinitesimally small time \( \epsilon \) after \( t = 8 \), i.e., at \( t = 10 \). Since the context switch routine is executed with interrupts disabled, \( P_1 \) is blocked from processing the incoming DI until the completion of the context switch which is denoted as \( OH_{bcs} \). After the completion of \( OH_{bcs} \) at \( t = 10.5 \), processor \( P_1 \) incurs \( OH_{RQ, adm} \) and \( OH_{RQ, spin} \) to process the incoming pending DI. Following the interrupt handler, \( P_1 \) also incurs \( OH_{cs} \) to context switch back to \( \tau_1 \) and then proceeds to execute its gcs. Upon the release of the resource and completion of \( OH_{RL, adm} \) at \( t = 17 \), regular preemption related context switch occurs and the task \( \tau_2 \) proceeds to execute.

Under other scenarios, unless the preemption occurs at \( \epsilon \) before the receipt of a DI, the context switching overhead is simply a part of the spinning time and the term \( OH_{bcs} \) does not appear. An

---

The numbers in the time-line are for reference purposes only. The relative sizes of overheads are not to scale.
7.2. Identification of overhead components

example of this can be seen at $t = 26$ in Figure 7.1b. Regular context switching overheads that are not within the scope of the RAP are not included in this study. The inclusion of context switching overheads has been studied extensively in other works and recommendations are available in works such as [16]. The low-level RTOS induced overheads such as the ones due to usage of hardware mutexes discussed in Chapter 9 are also excluded from this analysis.

7.2.3 Resource release overheads

The overhead components that contribute to scenarios $RL_1$ and $RL_2$ are identified and defined below:

**Def. 8.** $OH^{RLG}_{adm}$ - lowering the system ceiling to the original priority and updating the administrative data structures (book keeping) while releasing a global resource.

**Def. 9.** $OH^{DI}_{snd}$ - sending a Dedicated Interrupt (DI) to notify the release of a resource to a remote core.

Figure 7.5 illustrates the overheads $OH^{RLG}_{adm}$ and $OH^{DI}_{snd}$ with respect to the new implementation of FSLM. The context switching overheads that occur following the completion of the resource release are not considered as a part of the overheads since it is not within the scope of the RAP (for both MSRP and FSLM).

Figure 7.4: Example time-line scenario illustrating $OH^{DI}_{ snd}$.
Chapter 7. Overheads

7.3 Incorporating the overhead components in scenarios

Since the scenarios in which overheads can occur and also the individual overhead components contributing the scenarios have been identified, it is possible to represent the scenarios as a composition of the overhead components.

\[
RQ_1 = OH_{RQ,G}^{adm} \\
AC_1 = OH_{ccoli}^{DI} \\
AC_3 = AC_2 + OH_{ocols}^{DI} \\
RL_2 = RL_1 + OH_{and}^{DI}
\]

\[
RQ_2 = RQ_1 + OH_{spin}^{RQ,G} \\
AC_2 = AC_1 + OH_{coli}^{DI} \\
RL_1 = OH_{adm}^{RL,G}
\]

The worst case resource access scenario in terms of the implementation related runtime overheads for any task is a combination of \( RQ_2, AC_3 \) and \( RL_2 \). This combination of scenarios consists of all the overhead components listed in Section 7.2.

7.4 Overheads in MSRP

To facilitate a comparison between the implementations of FSLM and MSRP, it is also desirable to identify and characterize the implementation overheads in the native MSRP implementation provided in the Erika Enterprise RTOS. This section identifies the implementation overheads in the MSRP implementation of Erika Enterprise.

The native MSRP implementation uses a global spin-lock based method for global resource access as explained in Chapter 4.1.1. FIFO ordering of resource access is provided by the use of global polling-bit and “spin-status” data structure as explained. Since MSRP does not use an intercore communication mechanism to notify resource release, there are no ISR related overheads as in the case of FSLM.

The native implementation of MSRP only incurs \( OH_{RQ,G}^{adm} \) and \( OH_{adm}^{RL,G} \), which are defined by Definitions 3 and 8 respectively. Although the definitions of the terms are identical to that of FSLM, due to the inherent differences in the implementations of FSLM and MSRP, the values are expected to be different. Figures 7.6 and 7.7 illustrate the overheads \( OH_{adm}^{RQ,G} \) and \( OH_{adm}^{RL,G} \) with respect to the native MSRP implementation in Erika Enterprise.

![Figure 7.6: Resource access overheads in MSRP (native implementation).](image)

During resource request, \( RQ_2 \) does not occur since spinning always happens at HP and MSRP does not involve setting of a local spin-lock variable as in FSLM.
7.5. Extending the schedulability analysis

This section details a method for inclusion of the overheads in the schedulability analysis of FSLM for CP and HP spin-lock approaches, similar to the original analysis of FSLM performed in [2]. An extension of the analysis to include overheads is also performed for MSRP to aid the comparison between MSRP and FSLM. Figure 7.8 provides an overview of all the identified overhead components with respect to the order in which they typically occur within a resource access scenario. The identified overhead components are expected to remain constant for a given multiprocessor system. The overheads are independent of the tasks or the cores calling the tasks meaning that the overheads are characteristic of the given implementation and platform.

7.5.1 MSRP

The upper bound including overheads $B_L^t$ for LBL experienced by a task $\tau_i \in T_{P_k}$ is given by:

$$B_L^t = \max_{\forall j, l : \pi_j < \pi_i, \tau_j \in T_{P_k} \land R_l \in R_i} \{C_{s_j,l} + \overline{OH}_{LBL}^{MSRP}\},$$  (7.1)

where $\overline{OH}_{LBL}^{MSRP}$ represent the overheads related to requesting and releasing local resources, i.e.

$$\overline{OH}_{LBL}^{MSRP} = OH_{adm}^{RQL} + OH_{adm}^{RLL}. $$  (7.2)
The terms $OH_{adm}^{RQ,L}$ and $OH_{adm}^{RL,L}$ represent the administrative overheads incurred when local resource request and release occur respectively. These overheads are identified and characterized similar to $OH_{adm}^{RQ,G}$ and $OH_{adm}^{RL,G}$.

Similar to $\hat{OH}_{MSRP}^{MSRP}$, a term $\hat{OH}_{MSRP}^{LBG}$ for global resources is defined. Since MSRP only incurs the terms $OH_{adm}^{RQ,G}$ and $OH_{adm}^{RL,G}$ for global resource access as described in Section 7.4, $\hat{OH}_{MSRP}^{LBG}$ is defined as follows:

$$\hat{OH}_{MSRP}^{LBG} = OH_{adm}^{RQ,G} + OH_{adm}^{RL,G}.$$ (7.3)

The upper bound including overheads $\hat{B}_i^G$ for LBG experienced by a task $\tau_i \in T_{Pk}$ under MSRP is now given by:

$$\hat{B}_i^G = \max \{ Cs_{j,q} + \hat{spin}_{P_{k,q}} + \hat{OH}_{MSRP}^{LBG} \},$$ (7.4)

where $\hat{spin}_{P_{k,q}}$ is given by

$$\hat{spin}_{P_{k,q}} = \sum_{\forall P_r \neq P_k} \max_{\forall \tau_j \in T_{P_r,q}} \{ Cs_{j,q} + \hat{OH}_{RHT}^{MSRP} \},$$ (7.5)

where $\hat{OH}_{RHT}^{MSRP}$ represents the sum of overheads associated with the Remote Holding Time and is defined as follows:

$$\hat{OH}_{RHT}^{MSRP} = OH_{adm}^{RL,G}.$$ (7.6)

Remote Holding Time denotes the maximum holding time of a task $\tau_j$ on a different processor $P_r$ for the same resource $R_q$. Since the resource requests are FIFO ordered, the tasks on other cores that request access to $R_q$ earlier than $\tau_i$ are granted access contributing to the extension of the spinning time of $\tau_i$. Since the tasks on other cores must make the request before $\tau_i$ to contribute to RHT, $OH_{adm}^{RQ,G}$ does not contribute to RHT leading to Equation 7.6.

The upper bound on the total pi-blocking with overheads included ($\hat{B}_i$) experienced by a task $\tau_i \in T_{Pk}$ under MSRP is now given by

$$\hat{B}_i = \max \{ \hat{B}_i^L, \hat{B}_i^G \}.$$ (7.7)

The computation time $C_i$ of a task $\tau_i$ is measured by executing the task $\tau_i$ in isolation. Hence, the value of $C_i$ measured for every task is expected to include runtime overheads that are always
7.5. Extending the schedulability analysis

present. The computation time \( C_i \) of a task \( \tau_i \) under MSRP is defined by the following equation:

\[
C_i = C_{i}^{NCS} + \sum_{\forall l: R_l \in RS_L^i} \{ n_{i,l}^L \times (Cs_{i,l} + \hat{OH}_{LBD}^{MSRP}) \}
\]

\[+ \sum_{\forall q: R_q \in RS_G^i} \{ n_{i,q}^G \times (Cs_{i,q} + \hat{OH}_{LBD}^{MSRP}) \}, \tag{7.8}\]

where \( C_{i}^{NCS} \) represents the computation time of the non-critical section parts of the task and \( n_{i,l}^L \) represents the maximum number of requests by \( \tau_i \) for a local resource \( R_l \). Since no additional overheads other than the ones already included in \( C_i \) are incurred by the running task in MSRP, the computation time of a task \( \tau_i \) including all overheads \( C_{i}^{OH} \) is defined as:

\[
C_{i}^{OH} = C_i \tag{7.9}\]

The inflated computation time \( \acute{C_i} \) of task \( \tau_i \) is defined as:

\[
\acute{C_i} = C_{i}^{OH} + \acute{\text{spin}}_i \tag{7.10}\]

where the term of \( \acute{\text{spin}}_i \) is upper bounded by

\[
\acute{\text{spin}}_i = \sum_{\forall q: R_q \in RS_G^i \land \tau_i, \tau_j \in T_{Pk}} n_{i,q}^G \times \acute{\text{spin}}_{P_k,q}. \tag{7.11}\]

The worst-case response time including overheads \( W_{R_i} \) of task \( \tau_i \) is given by the smallest positive solution of the following recursive equation

\[
W_{R_i} = C_{i}^{OH} + \acute{B_i} + \sum_{\forall h, \pi_h > \pi_i \land \tau_i, \tau_j \in T_{Pk}} \left[ \frac{W_{R_j}}{T_h} \right] \acute{C_h}. \tag{7.12}\]

### 7.5.2 FSLM-HP spin-lock approach

The FSLM-HP spin-lock approach is similar to MSRP, but incurs additional overheads due to the implementation using local spin-lock variables and inter-core interrupts. For the upper bound on LBL, (7.1) can be reused. To incorporate the overheads into LBG, a term \( \hat{OH}_{LBD}^{HP} \) is defined, which includes all overheads, except the context switching overheads:

\[
\hat{OH}_{LBD}^{HP} = \{ OH_{adm}^{RQ,G} + OH_{spn}^{RQ,G} + OH_{hd}^{DL} + OH_{adm}^{RL,G} + OH_{snd}^{DL} \}. \tag{7.13}\]

The upper bound including overheads \( \acute{B_i}^G \) for LBG experienced by a task \( \tau_i \in T_{Pk} \) under FSLM-HP is given by:

\[
\acute{B_i}^G = \max_{\forall j: \pi_j < \pi_i \land \tau_i, \tau_j \in T_{Pk} \land R_q \in RS_{ii}^j} \left\{ Cs_{j,q} + \acute{\text{spin}}_{P_k,q} + \hat{OH}_{LBD}^{HP} \right\}, \tag{7.14}\]

where \( \acute{\text{spin}}_{P_k,q} \) is given by (7.15). The subtracted term in (7.15) denotes that in the worst case scenario, one of the remote tasks will always have access to the resource and hence would not incur inter-core interrupt handling overheads. Figure 7.9 shows an example time-line illustrating \( \acute{\text{spin}}_{P_k,q} \) described in Example 5.

\[
\acute{\text{spin}}_{P_k,q} = \max \left\{ 0, \left[ \sum_{\forall q: R_q \notin P_k} \max_{\forall \tau_j \in T_{P_k,q}} \left( Cs_{j,q} + \hat{OH}_{RHT}^{HP} \right) \right] - \left[ OH_{hd}^{DL} \right] \right\}, \tag{7.15}\]
where $\hat{OH}_{\text{RHT}}^{\text{HP}}$ represents the sum of overheads associated with the remote holding time under FSLM-HP and is defined as follows:

$$\hat{OH}_{\text{RHT}}^{\text{HP}} = OH_{\text{hdl}}^{\text{DI}} + OH_{\text{adm}}^{\text{RL}} + OH_{\text{snd}}^{\text{G}}. \quad (7.16)$$

![Figure 7.9: Example time-line scenario illustrating $\hat{\text{spin}}_{P_k,q}$ for FSLM-HP.](image)

**Example 5.** Consider a four core system with cores $P_0$ to $P_3$. Tasks $\tau_0$ to $\tau_3$ require access to the same global resource $R_q$ and are assigned to cores $P_0$ to $P_3$ respectively as shown in Figure 7.9. Task $\tau_0$ issues the resource request for $R_q$ at time $t = 0$. However, an infinitesimally small instant $\epsilon$ before $t = 0$, tasks $\tau_3$, $\tau_2$ and $\tau_1$ issue resource requests for $R_q$ in order. As a result, at $t = 1$, $\tau_3$ acquires the resource and begins execution of the global critical section while the other tasks start to spin. The remote holding time incurred by $\tau_0$ due to the other tasks is observed. From the figure, it can be seen that from $t = 1$, all the global critical sections of tasks $\tau_1$ through $\tau_3$ and the overheads $OH_{\text{adm}}^{\text{RL}}$ and $OH_{\text{snd}}^{\text{G}}$ contribute to the spinning time of task $\tau_0$. Since the first task to request the resource ($\tau_3$ in this example) will always access the resource without spinning, it would not incur $OH_{\text{hdl}}^{\text{DI}}$, as defined by Equation (7.15). Since the spin-lock priority in this case is HP, the context switching overheads $OH_{\text{bcs}}^{\text{DI}}$ and $OH_{\text{cs}}^{\text{DI}}$ are not a part of the remote holding time.

The upper bound on the total pi-blocking with overheads included ($\hat{B}_i$) experienced by a task $\tau_i \in T_k$, under FSLM is similar to (7.7).

The computation time $C_i$ of a task $\tau_i$ as measured by executing in isolation is defined by (7.8). Unlike MSRP, under FSLM, the tasks experience further overheads associated with every global resource request other than the ones incurred when the task is executed in isolation. To accommodate for the other RAP related overheads in the computation time, the term $C_i^{\text{OH}}$ is introduced and defined as follows:

$$C_i^{\text{OH}} = C_i + \sum_{q: R_q \in RS_k^G} n_{i,q}^{RG} \times \hat{OH}_{\text{CT}}^{\text{HP}}, \quad (7.17)$$

where $\hat{OH}_{\text{CT}}^{\text{HP}}$ represents the sum of all overheads other than the ones already encountered as a part of $C_i$ incurred during the computation time, which is given by:

$$\hat{OH}_{\text{CT}}^{\text{HP}} = OH_{\text{spn}}^{\text{RG}} + OH_{\text{hdl}}^{\text{DI}} + OH_{\text{cs}}^{\text{DI}}. \quad (7.18)$$

The inflated computation time $\hat{C}_i$, upper bound for $\hat{\text{spin}}_i$ and the worst-case response time including overheads $WR_i$ of task $\tau_i$ are calculated using (7.10), (7.11) and (7.12) respectively.

---

*The numbers in the time-line are for reference purposes only. The relative sizes of overheads are not to scale.*
7.5.3 FSLM-CP spin-lock approach

In FSLM-CP, for upper bound on $\hat{B}_i$, (7.1) can be used. In FSLM-CP, since the spin-lock priority of a core does not necessarily have to be the priority of the highest priority task on the core, there is a possibility of the spinning task being preempted by higher priority tasks leading to context switching overheads $O_{h_{bcs}}$ and $O_{h_{cs}}$. To accommodate for the additional context switching costs whenever $\pi^G_{P_k} < \pi^\text{cap}_{P_k}$, a dedicated $O_{h_{CS}}^{\text{CP}}$ is defined:

$$\overline{O_{h_{CS}}^{\text{CP}}} = O_{h_{bcs}} + O_{h_{cs}}^\text{DI}.$$  \hspace{1cm} (7.19)

For the calculation of upper bound including overheads $\hat{B}_i^G$ for LBG experienced by a task $\tau_i \in T_{P_k}$, the overheads term $\overline{O_{h_{LBG}}^{\text{CP}}}$ is determined. Since the inclusion of context switching overheads during spinning depends on the spin-lock priority of the core, the definition consists of two cases as defined below:

$$\overline{O_{h_{LBG}}^{\text{CP}}} = \begin{cases} O_{h_{LBG}}^\text{HP} & \text{if } \pi^G_{P_k} = \pi^\text{cap}_{P_k}, \\ O_{h_{LBG}}^\text{HP} + \overline{O_{h_{CS}}^{\text{CP}}} & \text{if } \pi^G_{P_k} < \pi^\text{cap}_{P_k}. \end{cases}$$  \hspace{1cm} (7.20)

The upper bound on LBG experienced by a task $\tau_i \in T_{P_k}$ for the CP spin-lock approach is defined as follows:

$$\hat{B}_i^G = \max_{\forall j,q, \pi^G_j < \pi^G_i, \forall \tau_i, \tau_q \in T_{P_k}, \forall R_q \in RS^i_q} \left\{ C_{s,j,q} + \overline{O_{h_{LBG}}^{\text{CP}}} \right\}$$  \hspace{1cm} (7.21)

where

$$sp_{\text{in}, P_k, q} = \max \left\{ 0, \left[ \sum_{\forall \tau_q \in T_{P_k}} \text{max} \left\{ C_{s,j,q} + \overline{O_{h_{	ext{RHT}, P_q}}^{\text{CP}}} \right\} \right]^{\text{CP}} \right\}$$  \hspace{1cm} (7.22)

and where $\overline{O_{h_{RHT}, P_q}}$ represents the sum of overheads associated with the remote holding time on core $P_q$:

$$\overline{O_{h_{RHT}, P_q}} = \begin{cases} O_{h_{	ext{RHT}}}^\text{HP} & \text{if } \pi^G_{P_i} = \pi^\text{cap}_{P_i} \\ O_{h_{	ext{RHT}}}^\text{HP} + \overline{O_{h_{CS}}^{\text{CP}}} & \text{if } \pi^G_{P_i} < \pi^\text{cap}_{P_i}. \end{cases}$$  \hspace{1cm} (7.23)

The value of $sp_{\text{in}, P_k, q}$ includes the remote holding time caused by the other cores in the system with tasks using the same resource $R_q$. Although it is similar to the scenario depicted by Figure 7.9 and Example 5, for FSLM-CP, additional context switching overheads whenever $\pi^G_{P_k} < \pi^\text{cap}_{P_k}$ must also be taken into account. The subtracted term in (7.23) accounts for the overheads that are to be deducted from the first remote core to acquire the resource $R_q$ without spinning. Since in the worst-case, the deducted value must be minimum for the worst-case value to be maximum, even if there exists one other core where the CP happens to be equal to HP, that core is considered to access the resource $R_q$ first, thereby resulting in the maximum possible worst-case value for $sp_{\text{in}, P_k, q}$.

The total pi-blocking $B_i$ experienced by a task $\tau_i \in T_{P_k}$ has the following upper bound:

$$\hat{B}_i = \begin{cases} \hat{B}_i^L + \hat{B}_i^G & \text{if } \pi_i > \pi^G_{P_k} + 1, \\ \max\{ \hat{B}_i^L + \hat{B}_i^G \} & \text{if } \pi_i \leq \pi^G_{P_k} + 1. \end{cases}$$  \hspace{1cm} (7.25)
Chapter 7. Overheads

The computation time $C_i$ of a task $\tau_i \in TP_k$ executed in isolation is similar to (7.8). The computation time with overheads included $C_i^{OH}$ is defined as follows:

$$C_i^{OH} = C_i + \sum_{\forall q : R_q \in RS_i^G} n_i^G \times \widehat{OH}_{CT}^{CP},$$ \hspace{1cm} (7.26)

where $\widehat{OH}_{CT}^{CP}$ represents all the overheads encountered during as a part of the computation time other than the ones already included as a part of $C_i$ and is defined as:

$$\widehat{OH}_{CT}^{CP} = \begin{cases} \widehat{OH}_{CT}^{HP} & \text{if } \pi P_k = \pi_{\text{max}} P_k, \\ \widehat{OH}_{CT}^{HP} + \widehat{OH}_{CT}^{CS} & \text{if } \pi P_k < \pi_{\text{max}} P_k. \end{cases}$$ \hspace{1cm} (7.27)

The computation time $C_i$, inflated computation time $\check{C}_i$, upper bound for $\hat{\text{spin}}_i$, and the worst-case response time including overheads $\check{WR}_i$ of task $\tau_i$ are calculated using (7.8), (7.10), (7.11) and (7.12) respectively.

This concludes the extension of schedulability analysis for the inclusion of implementation overheads in MSRP, FSLM-CP and FSLM-HP approaches. The extended equations presented above can be utilized for performing worst-case schedulability analysis on task sets to determine the most suitable RAP for the application. The results from this chapter were presented as a paper [10].
Chapter 8

Measurements and analysis

In this chapter, the measured values for the implementation overheads in the native MSRP, old implementation of FSLM and the FSLM implementation developed for this study are presented. A description of the software and hardware artifacts used for performing the measurements is also provided.

8.1 Hardware setup

The measurement of the implementation overheads involve monitoring of specific sections of the application code during runtime to identify the execution times to serve as an estimate for the relative sizes of the overheads. By consulting the Nios II application profiling guide [45], the following potential options were identified for performing the measurements:

1. GNU profiler
2. High resolution timer
3. Performance counter peripheral

A brief exploration of the above options is performed to identify the most suitable method for profiling the runtime overheads for the FSLM implementation on Erika Enterprise.

8.1.1 GNU profiler

Unlike the other two methods, using the GNU profiler does not require any hardware changes to the Nios II system. The measurement is performed by calling the functions in the profiler library through the compiler instrumentation of application code and hence does not require any dedicated hardware peripherals.

Although GNU profiler presents an advantage of not requiring hardware changes, profiling cannot be done for individual functions. It is intended to be used for profiling the entire application. Statistical estimates for the time spent on executing individual functions are then presented by the profiler by interpolations of the periodic sampling of the program counter. Hence, it presents only an estimation of where the CPU time is spent. Since the measurement of implementation overheads requires accurately identifying the execution time of individual sections within functions, GNU profiler is unsuitable as a measurement solution for the presented problem.

8.1.2 High resolution timers

High resolution timers are dedicated hardware on the system and hence require the use of Logic Elements (LEs) on the FPGA. Timers have the advantage of being implemented with a smaller number of LEs when compared to performance counter cores. Software calls to timer routines must then be included in the source code of the application to measure the sections of the code. The instrumentation in the source code must be performed manually and the software intrusion is less pervasive since it involves only calls to read the timer before and after the section of code that needs to be measured.

The number of sections of code that can be measured simultaneously using a timer is typically unrestricted because the execution time of each section while using the timer is measured by...
reading the timer before and after the desired code section. The values are then stored in variables which can be output to the terminal using the JTAG UART port.

However, the timer suffers from the following drawbacks. The calls to read the timer before and after every code section consumes more CPU cycles than using performance counters for measurement. The timer hardware offered by Quartus II has a highest possible resolution of $1\mu s$. Thus, for a 50MHz system like the one used for the study, the highest resolution possible in terms of number of clock cycles by using a high resolution timer is 50 cycles. This means that any value less than 50 cycles cannot be measured. The most important drawback of a timer is that it introduces an additional interrupt per hardware time on the core. This significantly increases the interrupt latency on the system. Since the implementation of FSLM relies upon interrupts for inter-core communication, use of timers could potentially result in interference with the values of the execution times being measured.

### 8.1.3 Performance counter cores

The performance counter cores require significantly more number of LEs (between 670 and 1345 LEs) when compared to high resolution timers. According to [41], the performance counter core is unobtrusive, requiring a single instruction to start and stop profiling and is appropriate for high-precision measurements of targeted sections of code. Each performance counter core can track up to seven sections of code.

The profiling guide [45], mentions that on the Nios II development platform, the performance counters are the most efficient means of performing measurements other than a completely hardware based solution. The performance counters require the sections of code being measured to be enclosed by `BEGIN` and `END` macros. The resolution offered is an individual clock cycle. Hence, for the 50MHz system under consideration, performance counters offer a resolution of $20\text{ns}$ when compared to $1\mu s$ offered by the high resolution timer. Moreover, performance counters use dedicated registers to store the profiling values and hence do not require RAM for storing the values unlike high resolution timers.

Every performance counter can measure only up to seven sections of code simultaneously. However, for the task under consideration on the new Altera DE2-115 development board, there are a significant number of LEs left unused that can be used for performance counter cores and the sections of code contributing to the runtime overheads associated with the RAP can be measured.

Due to the above reasons, the performance counter cores are chosen as the desired means for performing measurements for this study. Table 8.1 summarizes the findings from the survey of the available measurement options.

<table>
<thead>
<tr>
<th></th>
<th>GNU profiler</th>
<th>High resolution timer</th>
<th>Performance counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of LEs</td>
<td>none</td>
<td>120 to 200</td>
<td>670 to 1345</td>
</tr>
<tr>
<td>Resolution (in cycles at 50MHz)</td>
<td>NA</td>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>Number of code sections measured simultaneously</td>
<td>NA</td>
<td>Unlimited</td>
<td>7</td>
</tr>
<tr>
<td>Latency</td>
<td>high</td>
<td>medium</td>
<td>low</td>
</tr>
<tr>
<td>RAM usage</td>
<td>high</td>
<td>low</td>
<td>none</td>
</tr>
</tbody>
</table>

**TABLE 8.1: Comparison of measurement methods.**

### Performance counter cores setup

A brief description of the features of the performance counter cores and their inclusion in the hardware system presented. A list of prominent features of the performance counter cores is as follows:

- The performance counter core contains two counters for every section
  - i) Execution time: 64-bit clock cycle counter.
  - ii) Events: A 32-bit event (occurrence) counter.
The sections counters are grouped under a global counter. The section counters are enabled only when the global counter is running.

Performance counter core uses Avalon Memory-Mapped (Avalon-MM)[41] slave interface to access memory-mapped registers. The timers are started and stopped by writing to the registers. The time and event counts are retrieved by reading from the registers.

Two performance counter cores capable of measuring up to 3 individual sections each were added to the hardware design of the project. Figure 8.1 shows the inclusion of the 2 performance counter cores in the system design. It can be noted that the two performance counters are connected to all 4 cores in the system thereby enabling all the cores to use both the performance counters as needed.

**Figure 8.1: Performance counters in SOPC design**

| Table 8.2: Resource usage of performance counters. |
|-------------------|-----------------|-----------------|
| **LEs available in DE2-115** | Overall system | Performance counters |
| **LEs used (Combinational functions)** | 114480 | counter 0 | counter 1 |
| **LEs used (Logic registers)** | 12968 | 679 | 696 |
| **LEs available in DE2-115** | 8175 | 420 | 420 |

Table 8.2 summarizes the resource usage of the two performance counters in the overall system. It can be seen that on the system considered for the study, the use of performance counters does not raise concerns regarding consumption of LEs. The overall system merely uses 12% of the total LEs available on the FPGA. This serves to show that the hardware used for performance measurements do not impede the performance of the rest of the system in terms of utilization of Logic Elements.

### 8.2 Software setup

To enable measurement and monitoring of specific sections of the code in real-time, a set of software changes were performed. This section briefly summarizes the software setup used for performing the measurements on the platform.

#### 8.2.1 Software programming model for performance counter cores

The performance counter cores can be controlled with a set of C macros that are provided by Altera as a part of the Hardware Abstraction Layer for Nios II based systems. The set of functions, macros and constants provided by the Nios II Application Program Interface (API) for the performance counter cores are listed in Table 8.3. The complete description of each macro and function related to performance counters can be found in [41].
Chapter 8. Measurements and analysis

<table>
<thead>
<tr>
<th>Name</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>PERF_RESET()</td>
<td>Stops and disables all counters, resetting them to 0.</td>
</tr>
<tr>
<td>PERF_START_MEASURING()</td>
<td>Starts the global counter and enables section counters.</td>
</tr>
<tr>
<td>PERF_STOP_MEASURING()</td>
<td>Stops the global counter and disables section counters.</td>
</tr>
<tr>
<td>PERF_BEGIN()</td>
<td>Starts timing a code section.</td>
</tr>
<tr>
<td>PERF_END()</td>
<td>Stops timing a code section.</td>
</tr>
<tr>
<td>perf_print_formatted_report()</td>
<td>Sends a formatted summary of the profiling results to stdout.</td>
</tr>
<tr>
<td>perf_get_total_time()</td>
<td>Returns the aggregate global profiling time in clock cycles.</td>
</tr>
<tr>
<td>perf_get_section_time()</td>
<td>Returns the aggregate time for one section in clock cycles.</td>
</tr>
<tr>
<td>perf_get_num_starts()</td>
<td>Returns the number of counter events.</td>
</tr>
<tr>
<td>alt_get_cpu_freq()</td>
<td>Returns the CPU frequency in Hz.</td>
</tr>
</tbody>
</table>

Table 8.3: Performance counter macros and functions.

8.2.2 Setup for monitoring real-time performance

The C macros provided in the HAL for the performance counters offer means to measure the execution times of sections of code. In order to monitor the measured values, additional software changes were performed.

Measurement variables in the kernel

The functions EE_oo_GetResource() and EE_oo_ReleaseResource() containing the sections of code pertaining to the overheads identified in Chapter 7 are implemented in the Erika Enterprise kernel and are exposed to the application code through the API. Therefore, to perform measurements on sections of the implemented code, the C macros of the performance counter core will be embedded in the kernel files. The measured values must be communicated to the terminal for monitoring. As indicated in Figure 8.1, the JTAG UART port is attached to CPU 0 in the system. This port is intended to serve as the means for communicating the measured values.

The measurements in the kernel will be performed on the instantiations of the kernel on each CPU in the system. To use the JTAG port, CPU 0 must therefore have access to the measured values from all other CPUs. In order to facilitate this, a shared data structure is created in the kernel for storing the measured values, which is then accessed by CPU 0 periodically and printed out to the terminal.

A new file named ee_fslm_measure.h is created in the Erika Enterprise directory. The file contains the data structures used for performing measurements. Listing 8.1 shows the data structures defined in the newly created header file ee_fslm_measure.h.

```c
#define NUM_MEASUREMENT_PARAMS 5
#define NUM_CPU 4
extern EE_UINT32 MeasureQ[NUM_CPU];
```

Listing 8.1: Snippet from ee_fslm_measure.h showing the measurement data structures.

Shared measurement variables in the application

The data structures declared in ee_fslm_measure.h must be defined as shared data structures in the application in order for them to be accessed by all CPUs in the system. The shared data structure definition is performed in the application by creating a file named fslm.h, where the “extern” data structures declared in the kernel are defined as shared data structures. Listing 8.2
8.2. Software setup

shows the snippet from fslm.h in the application level where the data structures are defined as shared. From Line 13 of Listing 8.2, it can be seen that MeasureQ is a shared 2 dimensional integer array that has NUM_CPU \times NUM_MEASUREMENT_PARAMS elements.

```
#include <stdlib.h>

extern EE_UINT32 MQ_cpu0[NUM_MEASUREMENT_PARAMS];
extern EE_UINT32 MQ_cpu1[NUM_MEASUREMENT_PARAMS];
extern EE_UINT32 MQ_cpu2[NUM_MEASUREMENT_PARAMS];
extern EE_UINT32 MQ_cpu3[NUM_MEASUREMENT_PARAMS];

EE_UINT32 EE_SHARED_DATA(MQ_cpu0)[NUM_MEASUREMENT_PARAMS];
EE_UINT32 EE_SHARED_DATA(MQ_cpu1)[NUM_MEASUREMENT_PARAMS];
EE_UINT32 EE_SHARED_DATA(MQ_cpu2)[NUM_MEASUREMENT_PARAMS];
EE_UINT32 EE_SHARED_DATA(MQ_cpu3)[NUM_MEASUREMENT_PARAMS];

extern EE_UINT32 * MeasureQ[NUM_CPU];

EE_UINT32 * const EE_SHARED_DATA(MeasureQ)[NUM_CPU] = {
MQ_cpu0,
MQ_cpu1,
MQ_cpu2,
MQ_cpu3,
};
```

LISTING 8.2: Snippet from fslm.h showing the measurement data structures.

**Initializing the shared measurement variables**

The measurement data structure needs to be initialized during the system startup. Since CPU 0 is defined as the master CPU, the initialization is performed in the application code of CPU 0. Listing 8.3 shows the snippet from cpu0_main.c where all the values in MeasureQ are initialized to 0 during the system startup.

```
for(i=0; i<NUM_CPU; i++){
    for(j=0; j<NUM_MEASUREMENT_PARAMS; j++)
    EE_SHARED_DATA_MeasureQ[i][j] = 0x00;
}
```

LISTING 8.3: Snippet from cpu0_main.c showing the initialization of MeasureQ.

**Monitoring the values**

While the system is running, CPU 0 is also in-charge of periodically printing the contents of MeasureQ to the terminal through the JTAG UART port for monitoring. Figure 8.2 shows a sample output from the terminal produced while listening on the JTAG UART port of CPU 0. The information from the terminal provides the execution time of the overheads measured during runtime. The information is analyzed as follows:

- The value “66” occurs as the 4th element (index = 3) in the MeasureQ of CPU 1. From Listing 8.5, it can be seen that index 3 corresponds to the EE_di_send() function. Thus the hex-value “66” corresponds to 102 cycles, representing the execution time of EE_di_send() function, \( OH_{snd} \).
- Similarly the value “f3” occurs as the 3rd element (index = 2) in the MeasureQ of CPU 2. From Listing 8.5, it can be seen that index 2 corresponds to \( OH_{hd1} \). Thus the hex-value “f3” corresponds to 243 cycles, representing the execution time of \( OH_{hd1} \).
The use of a dedicated core (CPU 0) to communicate the measured values eliminates any influence on the measured values due to the measurement setup. As an alternative, the measurement values can also be stored in the global memory and output at the end of the execution. But this method may lead to overflowing global memory for longer periods of execution while presenting no significant advantage over the presented method of rewriting a predetermined set of measurement values during the execution while using a dedicated core to read and periodically output the values.

### 8.2.3 Code instrumentation

The actual code instrumentation performed to mark the beginning and end of the sections of code to be measured is presented in this section. Using the performance counter core API functions listed in Table 8.3, the parts of the code identified as overhead components in Chapter 7.2 are instrumented. Listing 8.4 shows the measurement of the overhead $OH_{\text{snd}}$ in the function `EE_hal_spin_out()` (which calls `EE_di_send()`) using the performance counter core 1. Line 21 shows that section 0 of performance counter core 1 is being used for the measurement. It can also be noted that since $OH_{\text{snd}}$ occurs during the measurement of $OH_{\text{adm}}$, the measurement of $OH_{\text{adm}}$ is paused and resumed in Lines 14 and 40 respectively. In Line 35, it can be seen that the value of the counter is written to the corresponding element in `MeasureQ`.

```c
__INLINE__ void __ALWAYS_INLINE__ EE_hal_spin_out(EE_TYPESPIN m) {
    EE_UINT32 task2notify=0;
    EE_altera_mutex_spin_in();
    task2notify=ResourceQ[m][EE_CURRENTCPU];
    ResourceQ[m][EE_CURRENTCPU]=0xa0;
    EE_altera_mutex_spin_out();
    if(task2notify!=GlobalTaskID[EE_stkfirst]) {
        /*------Stop measuring Rel admin overhead------*/
        #ifdef MF_REL_ADMIN
        PERF_END(PERFORMANCE_COUNTER_1_BASE,0);
        #endif
        /*------Start measuring DI send overhead------*/
        #ifdef MF_INTR_SEND
        PERF_RESET(PERFORMANCE_COUNTER_1_BASE);
        PERF_START_MEASURING(PERFORMANCE_COUNTER_1_BASE);
        PERF_BEGIN(PERFORMANCE_COUNTER_1_BASE,0);
        #endif
        /*------The section that is measured---------*/
        register EE_TYPERN_PARAM par;
        par.pending = 1;
        EE_di_send(task2notify);
        /*-----------------------------------------*/
        #ifdef MF_INTR_SEND
        PERF_END(PERFORMANCE_COUNTER_1_BASE,0);
        PERF_STOP_MEASURING(PERFORMANCE_COUNTER_1_BASE);
        MeasureQ[EE_CURRENTCPU][MF_INTR_SEND] = \
        perf_get_section_time((void *)PERFORMANCE_COUNTER_1_BASE, 0);
        #endif
        /*------Resume measuring Rel admin overhead------*/
        #ifdef MF_REL_ADMIN
        PERF_BEGIN(PERFORMANCE_COUNTER_1_BASE,0);
        #endif
    }
}
```

**Listing 8.4:** Snippet of `EE_hal_spin_out()` showing the code instrumentation for measurement of $OH_{\text{snd}}^{\text{DI}}$.

It can be observed from Listing 8.4 that the measurement-related code instrumentation is guarded by the macro `MF_INTR_SEND`. A set of such macros corresponding to each type of overhead are defined in the file `ee_fslm_measure.h` to provide control over the sections of code that are being measured. By commenting and uncommenting the different macros, it is possible to include or exclude sections of code from being profiled. Listing 8.5 shows the definition of the macros. The comments specify the performance counter core ID and the section counter ID that each of the sections use. The user must ensure that no two macros that use the same performance...
counter core are enabled simultaneously to avoid overwriting and loss of measurement data. The listing however, has all macros activated for illustration purposes.

```
/*
OVERHEAD MEASUREMENTS
Uncomment as per requirement
Beware !! DO NOT UNCOMMENT 2 OR MORE MACROS THAT USE
THE SAME PERFORMANCE COUNTER
CHECK THE IMPLEMENTED CODE TO CHANGE THE PERFORMANCE
COUNTER AS PER REQUIREMENT
0 0 S0 -> means -> COUNTER 0, SECTION 0
Valid Combinations: (0,1,4) (2,3)
*/
#define MF_REQ_ADMIN 0 //C0 S0
#define MF_REQ_SPIN 1 //C0 S1
#define MF_INTR_HANDLER 2 //C0 S0
#define MF_INTR_SEND 3 //C1 S0
#define MF_REL_ADMIN 4 //C1 S0
```

Listing 8.5: Snippet from ee_fslm_measure.h showing the measurement macros.

For example, from Listings 8.4 and 8.5, it can be observed that, if the execution time of EE_di_send() function is being measured on CPU 3, then the measured value would be written to MeasureQ[3][3].

### 8.2.4 The application

![Figure 8.3: Structure of the example application used for measurements.](image)

Figure 8.3 shows the structure of the example application used for performing the measurement experiments. The system consists of four processors $P_0$, $P_1$, $P_2$, and $P_3$ where $P_0$ acts as the master CPU to which the JTAG UART port is connected. The data structure $\text{MeasureQ}$ created in the shared memory between the cores facilitates the storage of measurement results which are then printed out to the terminal by $P_0$ as explained in the previous sections of this Chapter. A set of two global resources $R_0^G$ and $R_1^G$ were used as shown in the figure. The tasks were designed such that they can be triggered by using the on-board buttons and switches on the DE2-115 FPGA board. The tasks $\tau_1$ through $\tau_{12}$ were then triggered in specific sequences such that the identified overhead scenarios were encountered and measured.

### 8.3 Measuring the overheads

The hardware and software setup described in the previous sections are used to perform the measurement of the seven identified overhead components. However, the context switching overheads are calculated using a different approach which is described in this section. The results of the measurements are then presented in this section.
Chapter 8. Measurements and analysis

8.3.1 Context switching overheads

The context switching code segments in Erika are implemented in assembly language. Hence, the C macros used for annotating the sections of code to be measured cannot be used for these sections of code implemented in assembly language. The context switching overheads $OH_{bcs}^{DI}$ and $OH_{cs}^{DI}$ were calculated using the CPU performance equation [46]:

$$CPU_{time}^{cs} = IC \times CPI \times T_{cycle}, \quad (8.1)$$

where $CPU_{time}^{cs}$ is the execution time of the program being measured (context switching code section in this case), $IC$ is the number of instructions in the program being measured, $CPI$ is the average number of cycles per instruction and $T_{cycle}$ is the time per cycle.

For the system under consideration, the number of instructions ($IC$) in the assembly code pertaining to context switching was found to be 41. Since the frequency of the clock used is 50MHz, the value of $T_{cycle} = 20\mu s$. To calculate $CPI$, the composition of the types of instructions is analyzed. The Altera instruction set manual for Nios II core [47] was used to determine the composition, which was found to be as presented in Table 8.4.

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Percentage composition</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>29%</td>
<td>5</td>
</tr>
<tr>
<td>Store</td>
<td>29%</td>
<td>4</td>
</tr>
<tr>
<td>R-type</td>
<td>35%</td>
<td>4</td>
</tr>
<tr>
<td>I-type</td>
<td>5%</td>
<td>3</td>
</tr>
<tr>
<td>J-type</td>
<td>2%</td>
<td>3</td>
</tr>
</tbody>
</table>

**Table 8.4: CPI calculation for context switching overheads**

R-type instructions are used when the required data values are located in the CPU registers or data values are to be written to the CPU registers. I-type instructions have an immediate operand and J-type instructions are used when jumps are required. Since the system uses a 32-bit RISC sub-scalar (no pipelining) processor, the CPI is calculated as follows:

$$CPI = \frac{(5 \times 29) + (4 \times 29) + (4 \times 35) + (3 \times 5) + (3 \times 2)}{100} = 4.22$$

Using (8.1), $CPU_{time}^{cs}$ was calculated as $3.44\mu s$. In terms of number of clock cycles ($CPU_{cycles}^{cs}$),

$$CPU_{cycles}^{cs} = \lceil IC \times CPI \rceil = 173 \text{ cycles}$$

Since the same section of assembler code is called for both $OH_{bcs}^{DI}$ and $OH_{cs}^{DI}$, the execution time in terms of number of cycles for both the overheads is equal to 173 cycles.

8.3.2 Measurement results

The measured overheads in the implementations of MSRP (native Erika Enterprise implementation), the old implementation of FSLM [3] and the new implementation of FSLM developed for this project are summarized in Table 8.5.

The experiments were repeated 50 times on each core and the table shows the mean and maximum values along with the standard deviation (SD) from the obtained data for each of the overheads. A significant reduction in $OH_{bcs}^{DI}$ and $OH_{cs}^{DI}$ over the old implementation [3] can be observed in the new implementation of FSLM developed for this project.

The relative sizes of the cumulative overheads (sum of all individual overhead components) in the three implementations are visualized in Figure 8.4. It can be observed from the figure that the overall decrease in the execution time of overheads in the new implementation of FSLM when compared to the old implementation [3] is 57.22%.
Predictability

From the standard deviation values in Table 8.5, it can be observed that the overhead values of the new FSLM implementation when compared to old FSLM implementation are less dispersed, especially with respect to $OH_{DI_{hdl}}$ and $OH_{DI_{snd}}$. Figure 8.5 shows the deviation from the mean value (clock cycles) of the sum of all overheads in MSRP and FSLM (old and new) implementations visualized as a normal distribution. It can clearly be observed that when compared to the old implementation, the runtime overheads in the new implementation are more predictable, i.e., less dispersed. This is mainly attributed to the removal of shared RN message buffers in the new implementation. However, in comparison to MSRP, the values of FSLM new implementation are more dispersed due to the inherent differences between the implementations of MSRP and FSLM, with MSRP being less prone to runtime overheads as described in Chapter 7. Specifically, in the new implementation of FSLM, fluctuations in the DI related overheads occur due to the pair of origin and destination cores involved in the communication as explained in Chapter 4.4. Since the resource access and release is tracked by shared data structures $ResourceQ$ and $TailQ$ that are guarded by a mutex, minor fluctuations in execution times are observed.

8.4 Analytical evaluation

The overheads have been identified, characterized and measured in the previous sections of the report. It is desirable to compare the approaches of MSRP and FSLM while incorporating the measured implementation overheads to determine a guideline for the choice of resource access protocol for a given application using an example scenario.
8.4.1 Analysis scenario

This section contains the description of the example scenario used for the analytical evaluation. The goal of the analysis is to study the effects of worst-case local and remote blocking on a high priority task that does not use any global resources. The analysis is performed for MSRP, FSLM-CP and FSLM-HP approaches. The values of implementation overheads measured for the new implementation of FSLM and the native implementation of MSRP are used in the calculations.

Consider a system with \( m \) processors \( P_0, P_1, \ldots, P_{m-1} \). Tasks \( \{ \tau_{\text{hp}}, \tau_{\text{cp}} \} \) are mapped on to \( P_{m-1} \) while tasks \( \tau_0, \tau_1, \ldots, \tau_{m-2} \) are mapped on to \( P_0, P_1, \ldots, P_{m-2} \) respectively. All tasks except \( \tau_{\text{hp}} \) require access to a shared global resource \( R_q \) and are identical to each other in terms of global resource usage with their global critical section (\( gcs \)) occurring after a non-critical section which is of the same length for all tasks.

The tasks \( \tau_0, \tau_1, \ldots, \tau_{m-2} \) and \( \tau_{\text{cp}} \) request the global resource within an infinitesimally small time \( \epsilon \) in such a way that \( \tau_0 \) always accesses the resource first, followed by \( \tau_1 \) and so on and then finally by \( \tau_{\text{cp}} \). On processor \( P_{m-1} \), the high priority task \( \tau_{\text{hp}} \) is set to activate at the instant when \( \tau_{\text{cp}} \) makes the resource request. Since the resource request sections of the code are executed non-preemptively, \( RQ_2 \) is always encountered by \( \tau_{\text{hp}} \). This setup can be adopted to anywhere between 2 to 16 processors to perform the analysis \( \tau_{\text{hp}} \). Figures 8.6a and 8.6b show the above setup for a system with 3 processors for FSLM-HP and FSLM-CP approaches respectively.

Goal

The objective of the analysis is to observe the response time of the highest priority task \( \tau_{\text{hp}} \) by varying the global critical section (\( gcs \)) lengths of all the remaining tasks in the system under MSRP, FSLM-HP and FSLM-CP approaches while taking into account the measured implementation overheads.

Constants

The following parameters are maintained constant throughout the analysis for FSLM:

i) Computation time of \( \tau_{\text{hp}}, C_{\text{hp}} = 5000 \) cycles (100\( \mu \)s at 50MHz)

ii) \( \forall i : (0 \leq i \leq m-2) \land \tau_{\text{cp}} \quad C_{i}^{\text{NCS}} = 3000 \) cycles (60\( \mu \)s at 50MHz)
8.4. Analytical evaluation

\[
\begin{align*}
\text{(A) FSLM-HP.} & \\
\text{(B) FSLM-CP.} & 
\end{align*}
\]

**Figure 8.6: Analytical evaluation scenario.**

iii) \( RQ_1 = OH_{adm}^{RQ,G} = 369 \) cycles

iv) \( RQ_2 = RQ_1 + OH_{adm}^{RQ,G} = 383 \) cycles

v) \( AC_1 = OH_{hl}^{DI} = 122 \) cycles

vi) \( AC_2 = AC_1 + OH_{cs}^{DI} = 295 \) cycles

vii) \( RL_1 = OH_{adm}^{RL,G} = 373 \) cycles

viii) \( RL_2 = RL_1 + OH_{cs}^{DI} = 477 \) cycles

And for MSRP:

i) \( OH_{adm}^{RQ,G} = 306 \) cycles

ii) \( OH_{adm}^{RL,G} = 270 \) cycles

It can be noted that all the maximum values from the obtained overhead measurement results in Table 8.5 has been used in the analysis. All overheads except \( OH_{cs}^{DI} \) are encountered as a part of the analysis scenario. Although \( OH_{cs}^{DI} \) is not shown in Figure 8.6b, it will be encountered when the resource access is granted to \( \tau_{cp} \) before the completion of \( \tau_{hp} \).

**Variables**

The following parameters are varied for performing the analysis:

i) The number of processors \( m \).

ii) For each system of \( m \) processors, the values of global critical section \( (gcs) \) lengths of all tasks in the system are varied uniformly between 5 and 4000 cycles (0.1 to 80\( \mu \)s at 50 MHz).

8.4.2 Analysis results

The results of the analysis under different conditions are presented in this section. It is to be noted that the presented results and interpretations are specific to the scenario illustrated in Section 8.4.1 and to the implementations of MSRP and FSLM (new) on Erika Enterprise as instantiated on the Altera DE2-115 FPGA platform. Thus, the values do not necessarily reflect the worst-case
response times of the highest priority task but just represent the response times under the scenario considered. Although conceptually, the findings can be generalized to MSRP and FSLM, the values of overheads represented in the scenario are characteristic of the platform used for this study.

2 Processors

The analysis is initially performed with \( m = 2 \) with tasks \( \tau_0 \) in \( P_0 \) and \( \{\tau_{cp}, \tau_{hp}\} \) in \( P_1 \). The response time of \( \tau_{hp} \) plotted as a function of varying \( gcs \) lengths of tasks \( \tau_0 \) and \( \tau_{cp} \) is shown in Figure 8.7.

![Figure 8.7](image)

**Figure 8.7:** The response time of \( \tau_{hp} \) as a function of the \( gcs \) lengths of lower priority and remote tasks under MSRP, FSLM-CP and FSLM-HP with 2 processors.

The following observations can be made from the plot in Figure 8.7. All the time measurements are calculated from the number of clock cycles at 50 MHz clock frequency.

i) MSRP results in better response times for \( \tau_{hp} \) than FSLM-CP for \( gcs \) lengths less than 310 cycles (6.20\( \mu s \)).

ii) For \( gcs \) lengths greater than 310 cycles, FSLM-CP gives better response times for \( \tau_{hp} \) than MSRP. The relative improvement in response time of \( \tau_{hp} \) in FSLM-CP over MSRP becomes more pronounced as the \( gcs \) length increases further.

iii) The response time of \( \tau_{hp} \) under FSLM-HP is observed to be parallel to MSRP. This is justified since conceptually MSRP and FSLM-HP are identical, but the implementation of FSLM-HP incurs additional overheads that are not part of MSRP.

iv) MSRP and FSLM-CP always outperform FSLM-HP in terms of the response time of \( \tau_{hp} \) for all \( gcs \) lengths under the given analysis scenario.

Due to the fact that under the given scenario, FSLM-CP outperforms MSRP for response time of \( \tau_{hp} \) for \( gcs \) lengths above 310 cycles, it is possible to create task sets that are schedulable by FSLM-CP but not by MSRP. For instance, consider the task set proposed in Example 6.

**Example 6.** For a system with tasks \( \tau_0 \) in \( P_0 \) and \( \{\tau_{cp}, \tau_{hp}\} \) in \( P_1 \), assume a \( gcs \) length of 2500 cycles (50\( \mu s \)) for tasks \( \tau_0 \) and \( \tau_{cp} \). The response time for the task \( \tau_{hp} \) using FSLM-CP would be 2190 cycles (43.80\( \mu s \)) faster than MSRP. A relative deadline \( D_{hp} = 10000 \) cycles (200\( \mu s \)) for \( \tau_{hp} \) would cause a deadline miss by 16.90\( \mu s \) in MSRP while FSLM-CP would complete the execution of 26.90\( \mu s \) before the deadline.
The above example illustrates that for a system with as low as 2 processors, with respect to the response time of high priority tasks, it might be beneficial to adopt FSLM-CP over MSRP even for relatively small gcs lengths.

3 Processors

The analysis is extended to three processors. Figure 8.8 shows the response time of \( \tau_{hp} \) for both 2 and 3 processor scenarios to aid comparison. It is now possible to make the following observations from Figure 8.8. The figure also illustrates by means of two vertical lines, the points at which FSLM-CP starts to outperform MSRP for both 2 and 3 processor scenarios.

i) Increasing the number of processors to 3 causes the remote holding time of the resource \( R_q \) to increase further. Thus, the gcs length at which FSLM-CP outperforms MSRP reduces to 20 cycles (0.40\( \mu \)s).

ii) FSLM-HP is still outperformed by MSRP and FSLM-CP in terms of response times of \( \tau_{hp} \) with the increase in number of processors from 2 to 3.

Thus, it can be observed that with the increase in the number of processors, under the given analysis scenario, the gcs length at which response time of \( \tau_{hp} \) in FSLM-CP outperforms MSRP decreases.

More than 3 Processors

The analysis is further extended to include up to 8 processors. Since FSLM-HP has been shown to be always outperformed by FSLM-CP and MSRP, it is not considered for this case. Figure 8.9 shows the response time of \( \tau_{hp} \) under MSRP and FSLM-CP for 2, 3, 4 and 8 processor systems.

It follows from the previous set of observations that the increase in the number of processors causes FSLM-CP to outperform MSRP for shorter gcs lengths.
Chapter 8. Measurements and analysis

Figure 8.9: The response time of $\tau_{hp}$ as a function of the gcs lengths of lower priority and remote tasks under MSRP and FSLM-CP with 2, 3, 4 and 8 processors.

i) For a system with 4 processors, FSLM-CP outperforms MSRP for all gcs lengths under the considered analysis scenario.

ii) For an 8 processor system, the difference in response times between MSRP and FSLM-CP is very significant as seen in Figure 8.9. The increased remote holding time causes the response time in MSRP to increase while FSLM-CP allows for the completion of $\tau_{hp}$ while $\tau_{cp}$ waits for the resource. Due to the analysis scenario considered, FSLM-CP delivers a best case response time of 5383 cycles for $\tau_{hp}$ since it always incurs $RQ_2$.

Thus, in summary, it can be concluded that for the analysis scenario considered, the response time of the high priority task is significantly better under FSLM-CP when compared to MSRP as the number of processors in the system increases.
Chapter 9

Low-level hardware locks

The underlying RTOS in a multiprocessor system on which any application is built uses low-level hardware locks (mutex) to facilitate exclusive access to resources among competing entities. In this chapter, the effects of such low-level hardware locks on the performance of the application are investigated with respect to the FSLM implementation on Erika Enterprise RTOS.

9.1 Hardware mutex options in Erika

In the hardware configuration of a multiprocessor system for the Nios II platform, an Altera Avalon Mutex Peripheral is required by Erika Enterprise according to [27]. Hence, in the hardware design phase of the system, a mutex peripheral connected to all four cores in the system was created as shown in Figure 9.1.

![Figure 9.1: Hardware definition of mutex peripheral.](image)

The base address of the defined Altera HAL mutex device is auto-generated at the end of the hardware design synthesis in the system library within the file `system.h`. The base address of the mutex must be specified in the OIL file using the parameter `NIOS2_MUTEX_BASE` as a part of the OS object. Currently, Erika supports the specification of only one mutex device. The potential drawbacks of this approach are further investigated in this chapter.

Usage of the mutex device in the OS

Any section of code requiring the modification of a data structure that is shared across multiple cores is typically accessed under mutual exclusion. The functions `EE_altera_mutex_spin_in()` and `EE_altera_mutex_spin_out()` defined in the kernel layer of Erika are used to enclose the section of code that has to be executed under mutual exclusion. The above functions are used to acquire and release the low-level hardware mutex respectively.

For example, in the FSLM implementation, the data structures `ResourceQ` and `TailQ` are shared among all the processors in the system and hence require mutual exclusive access in order to be modified. Listing 9.1 shows the definition of the function `EE_hal_spin_out()` that is called during resource release. The value in `ResourceQ` has to be updated to signal that the resource has been released. It can be seen that in Line 9, the update of `ResourceQ` is enclosed within the calls to acquire and release the hardware mutex. This ensures that the shared variable is always modified by at most one core at any given time under mutual exclusion.
Chapter 9. Low-level hardware locks

9.2 Drawback of using one mutex device for the system

Although the use of a mutex device in Erika guarantees mutual exclusive access to shared data structures, the presence of only one such device may act as a bottleneck under certain situations. This section explores one such scenario in which it might be beneficial to have support for more than one hardware mutex device.

9.2.1 Existing scenario

Example 7. Consider a simple two processor system with processors $P_0$ and $P_1$. Let the tasks $\{\tau_0, \tau_1\}$ be mapped on to $P_0$ while $\{\tau_2, \tau_3\}$ are mapped on to $P_1$. There are two global shared resources $R_0$ and $R_1$ with tasks $\tau_0, \tau_2$ accessing $R_0$ and tasks $\tau_1, \tau_3$ accessing $R_1$.

Consider a system as described in Example 7. Unlike the FSLM implementation used for the study, assume that instead of a ResourceQ variable that is defined as a 2 dimensional array, the system consists of individual one dimensional arrays for every resource in the system, where each array has $m$ elements where $m$ is the number of processors in the system. Hence, for the scenario mentioned in Example 7, assume that the following shared data structures exist instead of a single two dimensional ResourceQ array:

i) ResourceQ_R0[2] - Resource queue for $R_0$ with two elements, one for each core.

ii) ResourceQ_R1[2] - Resource queue for $R_1$ with two elements, one for each core.

Under the given scenario, let the task $\tau_0$ in $P_0$ and $\tau_3$ in $P_1$ issue simultaneous resource requests for $R_0$ and $R_1$ respectively. This situation triggers the following set of events:

1. Although tasks $\tau_0$ and $\tau_3$ require access to different data structures in the global memory, all shared data structures are protected by the same hardware mutex.

2. Both the tasks $\tau_0$ and $\tau_3$ try to acquire the mutex simultaneously as shown in Figure 9.2a.

3. The access to low-level hardware mutex is not ordered. Hence any of the competing tasks may acquire the mutex first. In this example, assume $\tau_0$ acquires the mutex first and proceeds to modify ResourceQ_R0[$P_0$] while the task $\tau_3$ is forced to wait for the mutex to become available as shown in Figure 9.2b.

4. Upon completion of the modification of ResourceQ_R0[$P_0$], the task $\tau_0$ releases the hardware mutex as shown in Figure 9.3a.

5. Task $\tau_3$ now acquires the mutex first and proceeds to modify ResourceQ_R1[$P_1$]

It can be observed from this scenario that although $\tau_0$ and $\tau_3$ require access to different data structures in the global memory, since both the data structures are protected by the same hardware mutex, one of the tasks is always forced to wait for the other task to complete its mutual exclusive access.
9.2. Drawback of using one mutex device for the system

Both $\tau_0$ and $\tau_3$ try to acquire the mutex.

Figure 9.2: Using the same mutex for two unrelated shared data structures (steps 1 and 2).

Task $\tau_0$ releases the mutex.

Figure 9.3: Using the same mutex for two unrelated shared data structures (steps 3 and 4).

9.2.2 Alternative solution

The same scenario is explored under the condition that different shared data structures are now protected with separate hardware mutexes. Extreme care must be taken in such a scenario to ensure that the appropriate data structures are always protected by the correct hardware mutex since any failure to do so could result in race conditions. While using 2 hardware mutexes to protect $\text{ResourceQ}_R0$ and $\text{ResourceQ}_R1$, the following series of events occur:

1. Both the tasks $\tau_0$ and $\tau_3$ try to access different shared data structures through protected by different mutexes. From Figure 9.4a, it can now be seen that $\text{ResourceQ}_R0$ is protected by MUTEX 0 while $\text{ResourceQ}_R1$ is protected by MUTEX 1.
88

Chapter 9. Low-level hardware locks

(a) Both $\tau_0$ and $\tau_3$ try to access different shared data structures protected by different mutexes.

(b) Both $\tau_0$ and $\tau_3$ acquire respective mutexes without waiting.

FIGURE 9.4: Using individual mutex for each shared data structure.

2. Since there is no contention between the tasks on cores $P_0$ and $P_1$ for the same mutex, both the tasks $\tau_0$ and $\tau_3$ acquire the respective mutexes and proceed to update $\text{ResourceQ}_R0$ and $\text{ResourceQ}_R1$ respectively as shown in Figure 9.4b.

The above cases demonstrate the potential drawback associated with protecting unrelated shared data structures using the same hardware mutex. Especially since the access to the hardware mutex is unordered, it is possible for a core to be starved while waiting for a mutex due to multiple sequential mutex acquisition requests by other processors.

9.2.3 Motivation for existing design by Erika Enterprise

Although the above indicated problem appears to be straightforward, the motivation for introducing only one hardware mutex for an entire multiprocessor system by Erika Enterprise is inferred from [27].

Erika Enterprise was initially created as a uni-processor RTOS and later extended to include multi-processor support. As a result, RT-Druid and Erika Enterprise were modified to include support for multi-processors while relatively maintaining the same API of the uni-processor variant in order to facilitate easier migration of existing uni-processor applications to multi-processor systems. This decision allows the developer to work with the same API while the multiprocessor synchronization primitives such as the use of mutexes are abstracted in the kernel, away from the control of the application. Further reasons include reducing the dependencies on the hardware platforms itself and simplified design since the probability of occurrence of such a contention is very low. Thus, the use of a single hardware mutex allows for the above advantages. However, as demonstrated in this chapter, using separate hardware mutexes to protect the resource queue pertaining to each individual resource might result in the improvement of predictability of the RTOS related overheads.

9.3 Experimental measurement

In Section 9.2, it has been established that the unpredictability of RTOS primitives increases when a single hardware mutex is used to protect unrelated shared data structures. It is now desirable to measure the impact of such a scenario on the Altera DE2-115 platform to gain insights in to the magnitude of the induced unpredictability.
9.3. Experimental measurement

9.3.1 Measurement setup

The measurement setup described in Chapter 8 under Sections 8.1 and 8.2 is reused. However, since the experiment involves measurement of effects due to simultaneous requests to same hardware mutex, a few changes are made to the measurement setup.

Hardware addition

Since two tasks on different cores have to simultaneously access the same hardware mutex, manual triggering of tasks using buttons and switches will be inaccurate. Hence, the tasks are set to be triggered by hardware timer units. “Interval timer” hardware units with a timer resolution of 1 second were added to cores 1 and 2 in the existing hardware design as shown in Figure 9.5.

Software additions in the application

To trigger the tasks based on the hardware timer signals, the alarm object alt_alarm is used in the application. The hardware timers issue an interrupt every second since the resolution has been set to 1 second. The alarm object can now be configured to activate the tasks in integral multiples of 1 second. Listing 9.2 shows the snippet from CPU 1 for automatically activating task 3 once in every 10 seconds. The listing shows the definition of the alarm expiry callback, instantiation and activation of the alarm. Similar changes are made on CPU 2 to automatically activate another task once in every 10 seconds. CPU 1 and CPU 2 will be used for performing the experiment while CPU 0 will be used for printing the measured values to the terminal through the JTAG UART port.

```
// Define alarm expiry to 10 seconds
#define TASK3_INTERVAL 10*alt_ticks_per_second()

// Callback to be executed on alarm expiry
void Task3_alarm_callback (void* arg)
{
    ActivateTask(task3);
    return TASK3_INTERVAL;
}

int main(){
    //Declare the alarm
    alt_alarm task3Alarm;
    StartOS();
    //Start the alarm
    alt_alarm_start (&task3Alarm, TASK3_INTERVAL,
        Task3_alarm_callback, NULL);
    /*Rest of the main function*/
}
```

Listing 9.2: Snippet of example alarm initialization in cpu1_main.c.

CPU 1 and CPU 2 will be used for performing the experiment while CPU 0 will be used for printing the measured values to the terminal through the JTAG UART port. Two identical tasks on CPU 1 and CPU 2 are used for performing the measurements.

Software changes in the kernel

Due to the inherent inaccuracy of the two different timer hardwares in synchronizing with each other, in order for the two tasks on different cores to request the mutex at the same time, the
desired section of the code with the mutex request and release is executed multiple times to allow for overlap to occur. For the purposes of the experiment, the following code snippet illustrated in Listing 9.3 was added to the EE_oo_GetResource() function.

```
#ifdef MF_MUTEX
int dummy = 0;
int iter = 0;

//Use performance counter 0 for core 1
//Use performance counter 1 for core 2
if(EE_CURRENTCPU == 1){
  PERF_RESET(PERFORMANCE_COUNTER_0_BASE);
  PERF_START_MEASURING(PERFORMANCE_COUNTER_0_BASE);
} else{
  PERF_RESET(PERFORMANCE_COUNTER_1_BASE);
  PERF_START_MEASURING(PERFORMANCE_COUNTER_1_BASE);
}

//Repeat acquisition and release of mutex 100000 times
for(iter=0;iter<100000;iter++){
  if(EE_CURRENTCPU == 1){
    PERF_BEGIN(PERFORMANCE_COUNTER_0_BASE,0);
  } else{
    PERF_BEGIN(PERFORMANCE_COUNTER_1_BASE,0);
  }
  /*------The section that is measured-------*/
  EE_altera_mutex_spin_in();
  dummy++;
  EE_altera_mutex_spin_out();
  /*-----------------------------------------*/
  if(EE_CURRENTCPU == 1){
    PERF_END(PERFORMANCE_COUNTER_0_BASE,0);
  } else{
    PERF_END(PERFORMANCE_COUNTER_1_BASE,0);
  }
}

if(EE_CURRENTCPU == 1){
  PERF_STOP_MEASURING(PERFORMANCE_COUNTER_0_BASE);
  MeasureQ[EE_CURRENTCPU][0] = perf_get_section_time((void *)PERFORMANCE_COUNTER_0_BASE, 0);
  MeasureQ[EE_CURRENTCPU][1] = perf_get_num_starts((void *)PERFORMANCE_COUNTER_0_BASE, 0);
  MeasureQ[EE_CURRENTCPU][2] = dummy;
} else{
  PERF_STOP_MEASURING(PERFORMANCE_COUNTER_1_BASE);
  MeasureQ[EE_CURRENTCPU][0] = perf_get_section_time((void *)PERFORMANCE_COUNTER_1_BASE, 0);
  MeasureQ[EE_CURRENTCPU][1] = perf_get_num_starts((void *)PERFORMANCE_COUNTER_1_BASE, 0);
  MeasureQ[EE_CURRENTCPU][2] = dummy;
}
#endif MF_MUTEX
```

Listing 9.3: Mutex interference measurement code setup snippet.

The following features can be observed in the above listing:

- The snippet is added to the EE_oo_GetResource() function so that the section of code is executed every time a task tries to request a resource.

- Since this code is defined in the kernel and will be executed by different instantiations of the same code on different cores, it is not possible to use the same performance counter to measure the code section. This would result in both the cores using the same performance counter thereby distorting the measurement values. Hence, as seen in the listing, the code is instrumented such that any measurement by CPU 1 uses performance counter 0 while any measurement by CPU 2 uses performance counter 1. By this way, two different cores can simultaneously measure the same code section using different performance counters without interference.

- To ensure sufficient overlap between CPU 1 and CPU 2 while executing the same section of code, the section of code with acquisition and release of mutex is repeated 100000 times.

- The section of code enclosed by the mutex is a simple increment statement. This is analogous to most mutex protected code sections as they are typically restricted to making an update to a single shared data structure. It is not common to find long sections of code guarded by the mutex.
9.3. Experimental measurement

9.3.2 Measurement results

The experiment is carried out in three phases which are described as follows:

1. During the first phase, only the timer on CPU 1 is enabled and the time taken for executing the code section guarded by the mutex for $10^5$ iterations is noted. For 20 activation of the task, without contention for the mutex, the average value was found to be 4100038 cycles (4.1 x $10^6$ cycles approx.) with a standard deviation of 0.

2. The experiment is now performed with only the timer on CPU 2 enabled. For 20 activations the value on CPU 2 was found to be 4100036 cycles with a standard deviation of 0. It is now established that without contention for the mutex, the execution time is highly predictable.

3. Now, timers on both CPUs are activated. Since both the tasks are now activated once in 10 seconds simultaneously, during the execution of the code guarded by the mutex, there is a contention between the two tasks. The results are summarized in Table 9.1.

<table>
<thead>
<tr>
<th></th>
<th>No contention</th>
<th>Under contention</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU 1</td>
<td>CPU 2</td>
</tr>
<tr>
<td>Average execution</td>
<td>4100038</td>
<td>4100036</td>
</tr>
<tr>
<td>time (20 measurements)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>for $10^5$ iterations (in cycles)</td>
<td>6500025</td>
<td>6500028</td>
</tr>
<tr>
<td>Standard deviation</td>
<td>0</td>
<td>14.07</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>5.55</td>
</tr>
</tbody>
</table>

TABLE 9.1: Execution times for $10^5$ iterations of mutex acquisition and release for CPU 1 and CPU 2 under contention and no contention.

Since the mutex access is unordered, it can be seen from the standard deviation values in Table 9.1, under contention, that there is unpredictability induced in the otherwise stable overheads.

The graph in Figure 9.6 illustrates the increase in execution time caused due to contention between the two cores for acquiring the same hardware mutex. The increase in execution times between no contention and under contention for $10^5$ iterations is approximately 58.53%. Although theoretically, it is expected for the cumulative execution time of CPUs 1 and 2 under contention to be twice the cumulative execution time under no contention, due to inaccuracies in synchronization between the cores and the lack of guarantee that both cores begin the execution of the code sections at the same synchronized clock tick, the experimental values deviate from the expected theoretical values.

Thus, from the experimental results on the Altera DE2-115 platform with the chosen configuration, it can be seen that for a typical section guarded by a mutex, the execution time increases...
from 41 cycles per iteration to approximately 65 cycles per iteration under contention from one more core. Since the probability of acquiring the mutex under contention between two cores under the considered experimental conditions is 0.5, we do not see starvation of one core by the other core as evidenced by almost equal overall execution times of CPUs 1 and 2.

The number of CPUs contending simultaneously for the same mutex device is now increased to three. Table 9.2 shows the values obtained without and with contention during the experiments for 20 measurements each. The following observations can be made from the table:

<table>
<thead>
<tr>
<th></th>
<th>No contention</th>
<th>Under contention</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU 1</td>
<td>CPU 2</td>
</tr>
<tr>
<td>Average execution time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(20 measurements)</td>
<td>4100038</td>
<td>4100036</td>
</tr>
<tr>
<td>for $10^5$ iterations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(in cycles)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standard deviation</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 9.2**: Execution times for $10^5$ iterations of mutex acquisition and release for CPUs 1, 2 and 3 under contention and no contention.

i) Under no contention, the average execution times for $10^5$ iterations on CPU 3 is found to be 4000038, which is almost $10^5$ cycles less than the average values on CPUs 1 and 2. This indicates that every iteration on CPU 3 is 1 cycle faster than on CPUs 1 and 2. This could be attributed to the nature of the hardware platform.

ii) Under contention, the average execution times for $10^5$ iterations on all 3 CPUs increases to almost 9 million cycles, indicating an increase of between 117% and 124% with highest percentage increase for CPU 3. This increase is also illustrated by the graph in Figure 9.6.

iii) The variation in the measured values are indicated by the standard deviation in the table. It can be observed that the standard deviation for 3 CPUs has significantly increased when compared to the scenario with 2 CPUs presented in Table 9.1. This indicates that with an increase in the number of CPUs contending for the mutex, not only the execution time increases, but it also becomes less predictable.

Thus, from the experiments, when there is a contention between three cores for a section guarded by the same hardware mutex, the average execution time per iteration increases from 41 cycles to 90 cycles on CPUs 1 and 2 while it increases from 40 cycles to 92 cycles on CPU 3.

Hence, with the addition of multiple cores, it can be concluded that the performance and predictability might become worse in case of contention, especially when subsequent mutex acquisitions by a core can potentially starve other cores due to the inherent unpredictability of mutex acquisition. However, since the probability of two or more cores executing calls to the mutex simultaneously is extremely small, in order to avoid more complexity in the hardware design and to maintain the independence between the hardware platform and the operating system, in most scenarios, it might be beneficial to still retain only one low-level mutex per system.
Chapter 10

Conclusions

In this thesis, a real-time multi-core system based on partitioned fixed-priority preemptive scheduling (P-FPPS) and a spin-based global resource access protocol (FSLM) for spin-priorities in [CP, HP] was considered. P-FPPS and spin-based protocols for multi-core systems are prescribed by standards such as AUTOSAR [1]. The initial implementation of FSLM [3][4] in the OSEK/VDX compliant RTOS Erika Enterprise on an Altera Nios II platform using 4 soft-core processors was optimized by making performance and memory related improvements. From a theoretical perspective, i.e. by excluding runtime overheads, preemptive spin-based protocols, such as those described by FSLM, outperform non-preemptive protocols, such as MSRP. To make FSLM suitable for the automotive domain, the runtime overheads introduced by FSLM were evaluated, dedicated terms were added to the schedulability analysis and the overheads were measured using the above platform. A comparison between the response times of tasks under FSLM and MSRP including runtime overheads were made and the results indicate that FSLM becomes particularly attractive compared to MSRP when the critical section length of tasks accessing global shared resources increases and the number of cores increases. Recommendations for further optimizing the performance by exploring low-level RTOS mutual exclusion primitives were also provided. A detailed summary of the outcomes of the thesis and directions for future work are presented as conclusions.

10.1 Outcomes of the thesis

The outcomes of the thesis compared against the initial thesis goals (Chapter 1.4) are presented in this section as a part of the conclusions.

1. Goal [I.1] was satisfied by creating the Dedicated Interrupt (DI) mechanism to replace the RN mechanism for the reduction of inter-core communication overheads as detailed in Chapter 4. There was a significant reduction achieved in the overall inter-core communication related overheads. This chapter also answers the research question 3 (Chapter 1.3) by showing that a dedicated interrupts mechanism is indeed feasible for spin-lock priorities in [CP, HP] while offering significantly better performance than the native RN mechanism.

2. Goal [I.2] involves the implementation of dual shared stacks per core for FSLM and to explore the memory savings offered by the resulting implementation. This implementation has been detailed in Chapter 5. The chapter answers research question 2 by showing that the implementation is feasible and provides a general formula for calculating the memory saving by adopting the dual shared stacks method.

3. Goal [I.3] requires exploring the low-level hardware spin-lock usage in the RTOS. Chapter 9 explores the Erika Enterprise RTOS hardware mutex usage and by means of experiments, establishes that guarding unrelated shared data structures using the same hardware mutex leads to bottlenecks and uncertainty in the RTOS overheads. The findings of the chapter suggest that using a dedicated low-level hardware lock per resource can potentially reduce such overheads, thereby answering research question 4.

4. Goals [M.1] and [M.2] have been satisfied as detailed in Chapter 8 by creating an elaborate system for measuring various sections of code in the platform. The overheads were then identified and measured using the measurement system.
Chapter 10. Conclusions

5. Goal [A.1] pertaining to the inclusion of overhead terms in the schedulability analysis of FSLM has been addressed in Chapter 7. Overheads have been included in the analysis of MSRP as well as the two main spin-lock approaches of FSLM namely CP and HP. A generic method for classifying the RAP related overheads has been presented which can be applied with minor modifications to any RAP implementation. Analytical comparisons were then presented showing that with an increase in number of processors, in-spite of overheads, FSLM-CP outperforms MSRP with respect to the response times of high priority tasks in the system, thereby answering research question 1.

6. Goals [D.1] and [D.2] have been satisfied in the form of this thesis document and the included Appendices.

7. Goal [D.3] was to submit parts of the findings from the thesis at appropriate outlets. The following papers were written based the findings of the thesis:

   i) A paper titled “Incorporating implementation overheads in the analysis for the Flexible Spin-Lock Model” by Sri Muthu Narayanan Balasubramanian, Sara Afshar, Paolo Gai, Moris Behnam and Reinder J. Bril has been submitted to the 43rd Annual Conference of the IEEE Industrial Electronics Society (IECON 2017). The paper is under review as on the date of writing this thesis.

   ii) A Work-in-Progress poster paper titled “A dual shared stack for FSLM in Erika Enterprise” by Sri Muthu Narayanan Balasubramanian, Sara Afshar, Paolo Gai, Moris Behnam and Reinder J. Bril was submitted to the 23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2017). The paper was accepted by the review committee to appear at the conference as a poster presentation.

10.2 Future work

A few avenues for future work were identified during the thesis. A brief summary of the ideas are presented below:

1. Unified implementation. The implementations of MSRP and FSLM used for this study were not unified, i.e., they were separate implementations. During the thesis, it has been identified that even though FSLM-HP and MSRP approaches are theoretically identical, the FSLM-HP implementation suffers from additional runtime overheads which are characteristic of the implementation of FSLM. Also, the Dedicated Interrupts (DI) mechanism created for the new implementation of FSLM is suited only for spin-lock priorities in [CP, HP] for spin-lock priorities in [LP, CP), the use of Remote Notifications (RN) is required. Thus, it might be beneficial to have a unified implementation of FSLM and MSRP into a single framework which can leverage the advantages of MSRP while maintaining the flexibility of FSLM to further reduce the overheads in the system.

2. Built-in support for dual shared stack. It has been shown in the thesis that the existing infrastructure offered by Erika Enterprise and RT-Druid allows for the configuration of dual shared stacks in FSLM for spin-lock priorities in [CP, HP] by manually modifying the properties in the compiled files. Therefore, it might be desirable to make minor changes to RT-Druid and Erika such that dual stack configuration can be achieved automatically from the OIL file itself rather than the user having to modify the compiled files.

3. Per resource low-level hardware spin-lock. The drawbacks of using the same low-level hardware spin-lock (mutex) for guarding unrelated shared data structures has clearly been demonstrated in the thesis. Hence, as future work, it is desirable to have support for using multiple hardware mutexes, preferably one mutex per global shared resource in the Erika Enterprise implementations of FSLM and MSRP to reduce the unpredictability of RTOS overheads.
Appendix A

User manual - FPGA hardware and software design

A.1 Hardware manual

The user manual for re-creating the hardware design used for the thesis is provided in this appendix. It is to be noted that this appendix merely provides one approach of creating the hardware design. However, the user is free to use the tooling to create compatible hardware designs in any other method.

The following hardware and software tools are required:

**Hardware**

1. The Altera DE2-115 FPGA board was used to perform the experiments in this study. Altera DE2-115 purchase link - [http://www.terasic.com.tw/_sub/de2-115/](http://www.terasic.com.tw/_sub/de2-115/)
2. 1x USB A to B cable
3. 4x standard female-female breadboard berg connectors

**Software tooling**

1. Quartus II version 9.1 Web Edition
2. Nios II embedded design suite version 9.1
3. DE2-115 System Builder

**Installation**

The instructions for installation for Quartus II, Nios II embedded design suite and the device drivers for the FPGA board can be followed from Appendix B.1 of [4].

A.2 Creating the hardware design

A.2.1 DE2-115 system builder

The DE2-115 system builder software is used to create the basic framework for the hardware system. It automatically generates a Quartus II project file and a Verilog file for the project.

The DE2-115 system builder can be opened by running the **DE2_115_SystemBuilder.exe**
from the installation directory. In the resulting window, the settings for the project can now be configured. For the demo hardware project, the following settings are made as shown in Figure A.1.

![DE2-115 System Builder Screen](image)

**Figure A.1:** DE2-115 system builder screen

1. Provide a name for the project under “Project Name” field. In this example, we use the name “four_core_demo”. Ensure that there are no spaces in any of the file paths!

2. For the demo, the hardware system includes clock, LEDs and buttons.

3. The GPIO header is enabled. The inter-core communication network will be set up using the GPIO pins.

Upon completing the above settings, click “Generate” and choose a path for the system builder to generate the files. For purposes of this demo, the following file path is used: `C:/altera/Projects/four_core_demo`

The contents of the `C:/altera/Projects/four_core_demo` are shown in Figure A.2. It can be seen that the output folder contains a Quartus II project file named `four_core_demo.qpf`. This project file will be used to build the rest of the system.

![Output Files](image)

**Figure A.2:** Output files generated by the system builder

### A.2.2 Create the project in Quartus II

Open the Quartus II web edition v9.1 software as an **administrator** in Windows to ensure that the tool suite has permission to create new files in the project directory.
In the “Welcome” page of the tool, click “Open Existing Project”. Alternatively click File → Open Project … Navigate to the folder with output files from the system builder. In this demo: C:/altera/Projects/four_core_demo. Select the four_core_demo.qpf file to open it in the Quartus II tool. The steps are shown in Figure A.3.

We now have to add the individual components to the system design using the SOPC builder tool.

To open the SOPC builder tool click Tools → SOPC Builder

In the opening screen of the SOPC builder, a prompt automatically appears for the creating a new system as shown in Figure A.4.

For purposes of this demo, we create a new system with the name four_core_SOPC

A.2.3 Adding 4 Nios II cores

To add a Nios II processor to the system, from the System Contents menu on the left view the Component library.
In the Component library select Processors → Nios II processor as shown in Figure A.5.
A new window named “Nios II processor - cpu_0” appears. The wizard contains options for

choosing between three types of Nios II cores. For the purpose of this demo, we select the Nios II/f core as shown in Figure A.6
Select “finish” to add the Nios II core to the system. Repeat the same steps three more times to

get a total of 4 Nios II cores in the system. The newly added cores are automatically named from cpu_0 to cpu_3 as shown in the Figure A.7
A.2.4 Adding internal memory to the system

As a next step, we add internal memory to the system for all the four cores in the system. In the Component library select Memories and Memory Controllers \(\rightarrow\) On-chip \(\rightarrow\) On-Chip Memory (RAM or ROM) as shown in Figure A.8.

A new window named “On-Chip Memory (RAM or ROM) - onchip_memory2_0” opens with configuration options for the on-chip memory.
Select “RAM memory”
Set the “Data Width” to 32
We intend to allocate 48 KBytes of memory for each core in the system. Hence the “Total memory size” for the system is 192 KBytes or 196608 Bytes.
The steps for configuring the internal memory is shown in Figure A.9.

A.2.5 Adding a Mutex to the design

For enabling mutual exclusive access to shared data structures in the operating system kernel, hardware mutex is required in the system.
To add a new mutex in the system (Figure A.10), on the Component library select
Peripherals \(\rightarrow\) Multiprocessor Coordination \(\rightarrow\) Mutex
Since we intend to use Erika Enterprise as the RTOS, it requires the mutex to be created with the master CPU as the initial owner of the mutex. Considering that cpu_0 in our design is the master CPU, we assign the value of the mutex as 0x01 and specify cpu_0 as the initial owner of the mutex as shown in Figure A.11.

A.2.6 Inter-core interrupt input lines

To enable inter-core communication, every core must have an incoming interrupt line. To add an input interrupt the following actions are performed:
Select Peripherals \(\rightarrow\) Microcontroller Peripherals \(\rightarrow\) PIO (Parallel I/O)
A new window named “PIO (Parallel I/O) - pin_0” appears.
In the Basic Setting tab: (Figure A.12a)
Set “Width” to 1.
Set “Direction” to Input ports only.
In the Input Options tab: (Figure A.12b)
Check “Synchronously capture”
Set to “Raising edge”
Check “Generate IRQ”
Set to “Edge”
Click “Finish”
Repeat the steps 4 times to create 4 input lines. Rename the lines as “ipic_input_0” … “ipic_input_3”.
(Figure A.13)
A.2. Creating the hardware design

A.2.7 Inter-core interrupt output lines

To enable inter-core communication, every core must have an outgoing interrupt line to every other core. To add output interrupt lines the following actions are performed:

Select Peripherals → Microcontroller Peripherals → PIO (Parallel I/O)

A new window named “PIO (Parallel I/O) - pin_0” appears.

In the Basic Setting tab: (Figure A.14)

Set “Width” to 4 (Since there are a total of 4 cores in the system).

Set “Direction” to Output ports only.

Click “Finish”

Rename the newly created output as “ipic_output”.

---

**Figure A.11: Adding a mutex to the design - step 2**

**Figure A.12: Adding an input interrupt line - step 1**

**Figure A.13: Adding an input interrupt line - step 2**

**Figure A.13: Adding an input interrupt line - step 3**
A.2.8 Adding a button input for each core

To add button inputs for each core, perform the following steps once for each button input:
Select Peripherals → Microcontroller Peripherals → PIO (Parallel I/O)
In the Basic Setting tab:
Set “Width” to 1.
Set “Direction” to Input ports only.
In the Input Options tab:
Check “Synchronously capture”
Set to “Raising edge”
Check “Generate IRQ”
Set to “Edge”
Click “Finish”
Repeat the steps 4 times to create 4 input buttons. Rename the buttons as “button_pio_0”

A.2.9 Adding 4 LEDs for each core in the system

To add four LEDs each to every core in the system, the following steps are performed:
Select Peripherals → Microcontroller Peripherals → PIO (Parallel I/O)
A new window named “PIO (Parallel I/O) - pin_0” appears.
In the Basic Setting tab: (Figure A.14)
Set “Width” to 4 (Since we connect 4 LEDs to each core).
Set “Direction” to Output ports only.
Click “Finish”
Repeat the above steps 4 times to create four such outputs. Rename the outputs as “led_pio_0”

…“led_pio_3”. (Figure A.15)
A.2.10 Modifying the Nios II core properties

The properties of the Nios II cores must be modified to include information about memory allocation.
Each Nios II core has the following properties:

1. Reset Vector - Denotes the physical memory used for reset vector. In our demo it is “onchip_memory_2_0”
2. Exception Vector - Denotes the physical memory used for exception vector. In our demo it is “onchip_memory_2_0”
3. Reset Vector Offset - Denotes the starting offset address for reset vector.
4. Exception Vector Offset - Denotes the starting offset address for exception vector.

Since we plan to allocate 48KBytes for each core, the following values are determined for Reset and Exception vector offsets of the four cores:

- cpu_0  reset vector offset 0x00  exception vector offset 0x020
- cpu_1  reset vector offset 0xc000  exception vector offset 0xc020
- cpu_2  reset vector offset 0x18000  exception vector offset 0x18020
- cpu_3  reset vector offset 0x24000  exception vector offset 0x24020

The above settings have to be made for all the cores as shown in Figure A.16a. The CPU ID is assigned manually for each cpu from 0x01 to 0x03 respectively as shown in Figure A.16b.

A.2.11 Adding JTAG UART and performance counters

To add a JTAG UART port select Interface Protocols → Serial → JTAG UART and clock “Finish”. Similarly, to add performance counter cores to the design, select Peripherals → Debug and Performance → Performance Counter Unit.

A.2.12 Connecting and rearranging the components

All components have now been added to the design. The components can now be rearranged and connected as shown in Figure A.17. The components are ordered such that in each core, the ipic input line appears before other input lines such as buttons. This ensures that when the system auto-assigns interrupt priority levels, the inter-core interrupt always gets a higher priority than the buttons. All components are connected to the Clock “clk_0” as indicated in the figure.

A.2.13 Generate the design

The components have now been connected. The base address for the components has to be assigned. In order to perform this automatically, select System → Auto Assign Base Addresses.
Similarly to auto-assign the IRQ priority levels, select System → Auto Assign IRQs. Ensure that there are no errors in the information window at the bottom of the page. Save the design and click “Generate” to start the system generation. Upon successful completion of the system generation, the message shown in Figure A.18 is displayed in the window.

**Figure A.18: Successful completion of system generation**

### A.3 Synthesis of the design

After the completion of system generation, exit the SOPC builder tool and return to the Quartus II main window.
A.3. Synthesis of the design

A.3.1 Pin assignment

We now have to assign the pins for components such as buttons, LEDs and inter-core interrupt lines on the FPGA board.

In the Project Navigator pane, double-click the top entity “four_core_demo” as shown in Figure A.19.

![Figure A.19: Top entity in project navigator](image)

This opens the `four_core_demo.v` file which is the main verilog file defining the connections in the system. We now have to add the pin assignments to end of the file flagged by the comment “Structural coding”.

The component information is available in the output folder of the SOPC builder.

Click File → Open → `four_core_SOPC.v`

This opens the `four_core_SOPC.v` file which contains all the module informations of the generated system design. We now have to find the main module in the file. The main module is named “four_core_SOPC”. Search for the module in the file and copy the entire module as shown in Figure A.20.

![Figure A.20: Copy the contents of the module from four_core_SOPC.v file](image)

Paste the copied contents under the “Structural coding” part of `four_core_demo.v` file before the “endmodule” keyword.

Modify the pasted contents as shown in Listing A.1.

The physical pin details for the board can be found in [48].

```vhdl
//=======================================
//Structural coding
//=======================================

four_core_SOPC u0(
    //glb_signals:
    clk_0(CLOCK_50),
```
Listing A.1: Contents specifying pin assignments that must be added to the `four_core_demo.v` file

A.3.2 Analysis and synthesis

To compile and synthesize the hardware, click on the “Compile Design” option in the Tasks pane on the left (Figure A.21)

![Figure A.21: Analysis and synthesis](image)

To program the hardware, connect the DE2-115 board via USB and click Tools → Programmer. In the programmer window, click “Start” to program the device.
A.4  Software creation

The software components required are listed below:

<table>
<thead>
<tr>
<th>Software</th>
<th>Version</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSLM Erika</td>
<td>evidence_ee</td>
<td><a href="https://github.com/srimuthu/Erika-FSLM">https://github.com/srimuthu/Erika-FSLM</a></td>
</tr>
</tbody>
</table>

Steps for the creation of software system are identical to the ones presented in Appendix B.3 of [4].
Appendix B

Flex-Tool guide

Flex-Tool is a utility tool developed for the Erika implementation of FSLM to accomplish the following tasks semi-autonomously:

1. Determine priorities from OIL file
2. Receive user input for spin-lock priorities
3. Add spin-lock priorities to Erika application
4. Modify stack to create dual shared stack

B.1 Tool usage

How to download

The Flex-Tool can be downloaded from github by cloning the following repository: https://github.com/srimuthu/Erika-FSLM.git

Path to Flex-Tool in the repository: Erika-FSLM/flex_prio_tool/FlexTool.py

Flex-Tool details

File name: FlexTool.py
Requires Python version: Python 3
Package dependencies:
- prettytable
  (for displaying output on the console in table format)

Usage instructions

This guide presents a step-by-step guide for using the Flex-Tool.

1. Place the file FlexTool.py in the same directory that contains the conf.oil file.

2. Flex-Tool assumes that the user has not modified the following default file names:
   - The OIL file: conf.oil
   - The CPU application source files: cpuX_main.c
     where X is replaced by the cpu ID. For instance, cpu0_main.c

3. For the Flex-Tool to work as expected, the following convention must be followed in the OIL file while mentioning the STACK attribute of the TASK object. Please ensure that the keyword STACK and the corresponding terminator (;) appear on the same line, i.e., without a new-line character (“\n”) in-between. The following snippet shows the correct and wrong methods:
Figure B.1: Flex-Tool example summary of application and prompt for user input

Listing B.1: Convention for mentioning the TASK STACK attribute in OIL file
B.1. Tool usage

4. The tool involves setting of spin-lock priorities in the application depending on the user-input. In order to do so, the CPU application source files must contain the following global variables:

```c
const int EE_th_spin_prio[] = {}; 
const int GlobalTaskID[] = {}; 
```

**Listing B.2: CPU application file variables**

Kindly note that it is sufficient for the `cpuX_main.c` to just contain these variables. Flex-Tool will automatically find and modify the values of the above variables during run-time.

5. Run the Flex-Tool by using the following command:

```
$ <path to source dir> python FlexTool.py 
```

6. The Flex-Tool parses the OIL file and determines the spin-lock priority ranges. The complete summary of the application are presented on the console for the user to verify the correctness as shown in Figure B.1. It then prompts the user to input a spin-lock priority for each core.
7. The “Derived Priority” table displayed on the console provides information about the priority levels on each core. The user can use the information to input the spin-lock priority for each core.

8. Upon completing the user input, Flex-Tool computes the spin-priorities, modifies the values of the variables listed in Listing B.2 for each cpu, computes the stack allocation for the application and accordingly edits the OIL file. The user is then prompted to refresh the project in eclipse and Clean and Build Erika.

9. On Nios II IDE, refresh the project (optionally clean) and then build the project as shown in Figure B.2.

10. When the build is complete, preyy “y” in the python console to resume Flex-Tool. The Flex-Tool will now edit the eecfg.c files of the CPUs to modify the stack data.

11. The user is prompted upon completion of the updates.

12. Finally, use the “Run” option in the Nios II IDE to generate the .elf file and run the program on the hardware as shown in Figure B.3

Warning! Do not re-build Erika after the last step as this will cause RT-DRUID to overwrite the changes made to eecfg.c files by the Flex-tool! Just run the configuration as this initializes only the makefiles necessary to generate executable file without initializing RT-DRUID.
B.2 Auto generated documentation

class FlexTool.FlexTool (oilPath)
FlexTool Class - Utility tool for FSLM implementation in Erika Enterprise

Comments []

1. Determine priorities from OIL file
2. Receive user input for spin-lock priorities
3. Add spin-lock priorities to Erika application
4. Modify configuration files to create dual shared stack

_FlexTool__calculateSpinPriorities ()
Convert user input spin-priorities to Erika format (2^i)

_FlexTool__calculateStackAllocation ()
Group tasks on a core to 2 stacks based on spin-lock priority

_FlexTool__createResourceList ()
Create Resource list data structure from OIL file

Comments: __resourceInfo dict consists of
{resID (int) : resourceName (str)}

_FlexTool__createTaskData (task_counter)
Create Task information data structure from OIL file

Comments: The __taskInfo dict consists of:
    taskID (int) : [ 0] taskName (str),
                 [1] cpuID (int),
                 [2] cpuName (str),
                 [3] taskPriority (int),
                 [4] resourceBool (bool),
                 [5] resources (list) = [resourceName (str)] ]

_FlexTool__displayParams ()
Output task, resource and CPU info to the console

_FlexTool__editSystemTos (block_data, cpu)
Edit “EE_nios2_system_tos” variable in “eecfg.c”

Arguments: block_data - Part of file buffer from eecfg.c containing EE_nios2_system_tos,
cpu - CPU ID (cpu which the eecfg.c file belongs to)

Returns: Success or Failed (bool),
Edited file buffer of eecfg.c containing EE_nios2_system_tos

_FlexTool__editThreadTos (block_data, cpu)
Edit “EE_hal_thread_tos” variable in “eecfg.c”

Arguments: block_data - Part of file buffer from eecfg.c containing EE_hal_thread_tos,
cpu - CPU ID (cpu which the eecfg.c file belongs to)

Returns: Success or Failed (bool),
Edited file buffer of eecfg.c containing EE_hal_thread_tos
_FlexTool__findBraceBlock (data, item)
   " Identify the brace enclosed block containing a given string
   Arguments:  data - file buffer ,
               item - item to be found (str)
   Returns:  Index of brace block start
               Index of brace block end
               Data enclosed between in the block containing the (item)

_FlexTool__findGlobalResources ()
   Identify global resources (classify local and global resources)

_FlexTool__parseCpuInfo ()
   Extract CPU information from OIL file
   Called by : parseOilFile()
   Comments  [] Iterate through the OIL file and save CPU info to __cpuInfo dict __cpuInfo: { cpuID (int) :
                     cpuName (str) }

_FlexTool__parseTaskInfo ()
   Extract task information from OIL file

_FlexTool__reducePriorities ()
   Transform priority levels within a core to consecutive values starting with 0 (lowest)

_FlexTool__spliceTextToFileBuffer (file_buffer, block_data, start_index, end_index)
   Inserts given text into a file at specified location
   Arguments:  file_buffer - Full file buffer
               block_data - Data to be inserted
               start_index - Start position to insert
               end_index - End position
   Returns:  splicedList - Full file buffer with block_data inserted

calculatePriorities ()
   Calculate CP, CP hat and HP priorities
   Calls  [] __findGlobalResources()
           __reducePriorities()
           __displayParams()
   Called by  [] main() using FlexTool object
   Comments  []
               1. Identify global variables
               2. Transform priority levels within a core to consecutive values starting with 0 (lowest)
               3. Use available data to find CP, CP hat and HP for every core

getUserInput ()
   Get input from the user
   Calls  [] __calculateSpinPriorities()
           __calculateStackAllocation()
Called by [] main() using FlexTool object

Comments []
1. Prompt user for spin-lock priority per core
2. Convert priorities to HEX
3. Determine stack allocation of tasks into dual stacks on each core

`initializeFlexSpinToolVars()`
Initialize the flexible spin-lock priority tool related variables

Called by [] main() using FlexTool object

Comments [] Uses variables __cpuInfo, __taskInfo, __resourceInfo to construct mapping between tasks, cores and resources

`parseOilFile()`
Parse and extract information from OIL file

Returns [] self.__cpuInfo, self.__taskInfo, self.__resourceInfo

Calls [] __parseCpuInfo(), __parseTaskInfo(), __createResourceList()

Called by [] main() using FlexTool object

`promptUser()`
Prompts user to rebuild Erika.

Comments: This prompt is followed by the editing of eecfg.c files for dual stack implementation

`returnFlexSpinInfo()`
Return __spinPrio, __tasks2stack, __cpuInfo, __tasks2cores

`updateCfgFiles()`
Function to update Erika configuration files “eecfg.c” on all cores to include stack information

Called by [] main() using FlexTool object

Comments [] For all cores, modifies the “eecfg.c” to have only 2 shared stacks per core as per dual stack configuration

`updateOilFile()`
Modifies task stack info in OIL file to accommodate dual stack

Called by [] main() using FlexTool object

Comments [] Update “conf.OIL” file with task STACK attribute
1. SHARED for tasks with priorities up to spin-lock priority
2. PRIVATE for other tasks (later modified to shared after code generation by updateCfgFiles() )

___________!!!WARNING!!!___________

Only works for single line STACK attribute. Please specify the STACK attribute in a single line in the OIL file i.e keyword “STACK” and terminator “;” must be on the same line.

`updateSourceFiles()`
Update the application source files based on user input for spin-lock priority

Called by [] main() using FlexTool object

Comments [] Update “cpuXX_main.c” files for all cores
1. update EE_th_spin_prio
2. update GlobalTaskID

Throws error if the above variables are not found in the file

Also throws error if the file “cpuXX_main.c” is not found (XX = cpu ID (integer))
Bibliography


[12] S. Afshar, Lock-based resource sharing in real-time multiprocessor platforms, Please contact the authors if you would like to have a higher quality electronic version., Nov. 2014. [Online]. Available: http://www.es.mdh.se/publications/3837-


