TASK MIGRATION IN VIRTUALIZED MULTI-CORE REAL-TIME SYSTEMS

Simon Holmbacka

Master of Science Thesis
Supervisor: Johan Lilius
Department of Information Technologies
Åbo Akademi University
September 2010
Virtualization of embedded systems is growing more popular due to the increasing complexity of hardware and the benefits of using the technology. A virtualization layer can provide isolation of hardware and core-to-core communication support in a multi-core embedded system. One of the challenges of developing a virtualization layer for embedded systems origin from the fact that a generic embedded multi-core platform cannot be expected to serve all functionality concerning both hardware architecture and scope of use. This thesis suggests an approach to embedded system virtualization based on the FreeRTOS kernel. Virtualization could help the embedded systems to optimize the energy consumption and safety by having the ability of moving running tasks between CPU cores. This thesis shows ways of optimizing multi-core embedded system by using the aforementioned technique called task migration. In order to achieve task migration the different platform models must be studied deeply before an implementation strategy can be established. Different memory models, communication structures and hierarchies are essential parts of the modeling when deciding on a design approach. In a real-time embedded system the task migration mechanism is based partly on a discussed energy policy that states the amount of CPU power currently in use, and partly on an additional policy to preserve the real-time requirements. The weights of these policies are presented together with simulations and experiments regarding task migration, to give a foundation for a future implementation of a virtualization layer for multi-core embedded systems.

**Keywords:** virtualization, multi-core embedded systems, hypervisor, task migration, energy policy, mixed-critical systems
CONTENTS

Abstract i

Contents ii

List of Figures v

Glossary vii

1 Introduction 1
   1.1 RECOMP project ........................................... 2
   1.2 Thesis structure ........................................... 2

2 Virtualization using hypervisor 3
   2.1 Introduction to virtualization ............................ 3
   2.2 Virtualization techniques .................................. 4
      2.2.1 The hypervisor ...................................... 5
      2.2.2 Hardware support for virtualization ................. 6
   2.3 FreeRTOS ...................................................... 6
      2.3.1 The kernel ........................................... 7
      2.3.2 A hypervisor for FreeRTOS .......................... 8
      2.3.3 Memory protection .................................. 9
   2.4 Task creation in FreeRTOS ................................ 9
      2.4.1 Task list ............................................. 11
      2.4.2 Task creation in a Multi-Core system .............. 12
      2.4.3 Interfaces for creating tasks ...................... 14
   2.5 Context switch using hypervisor ....................... 14
   2.6 Summary ..................................................... 18

3 Task migration 19
   3.1 Introduction .............................................. 19
   3.2 Overview .................................................. 20
      3.2.1 Starting a task migration .......................... 20
      3.2.2 Reasons for task migration ......................... 21
      3.2.3 Load balancing monitor ............................. 23
3.2.4 Stages of task migration .................................. 23
3.3 Task migration models ........................................ 25
  3.3.1 Re-creation .................................................. 25
  3.3.2 Replica ...................................................... 26
3.4 Task migration communication ................................. 28
  3.4.1 Inter-Core communication .................................. 28
  3.4.2 Shared memory .............................................. 29
  3.4.3 Crossbar .................................................... 30
  3.4.4 Point-to-point communication ............................ 31
  3.4.5 Summary .................................................... 32
3.5 Task migration mechanism ................................. 33
  3.5.1 Checkpoints ............................................... 33
  3.5.2 Debug registers .......................................... 36
  3.5.3 Incoming migrations ....................................... 37
  3.5.4 Accepting task migrations by polling .................. 37
  3.5.5 Accepting task migrations by interrupts ............... 39
  3.5.6 Summary .................................................... 41
3.6 Task migration initiative .................................... 42
  3.6.1 Master-Slave ................................................. 42
  3.6.2 Decentralized monitors ................................... 43
  3.6.3 Summary .................................................... 45

4 Load balancing policy 46
  4.1 Introduction .................................................. 46
  4.2 Measurement techniques .................................... 47
    4.2.1 Temperature measurement ................................ 47
    4.2.2 Discrete control blocks ................................ 48
  4.3 Migration policies ........................................... 49
    4.3.1 Performance ............................................. 49
    4.3.2 Energy consumption ..................................... 51
  4.4 Implementation of monitor .................................. 53
    4.4.1 Monitor latency .......................................... 54
  4.5 Summary .................................................... 54

5 Mixed-critical systems 55
  5.1 Overview of a mixed-critical system ....................... 55
  5.2 Task migration in a real-time system ..................... 56
    5.2.1 Migration of soft real-time tasks .................. 57
    5.2.2 Migration of hard real-time tasks .................. 62
    5.2.3 Task migration hazards in a hard real-time system .. 63
    5.2.4 Weighted policies ...................................... 65
    5.2.5 Emergency task migration .............................. 66
    5.2.6 Elevator example ........................................ 67
5.3 Simulations ................................................. 69
  5.3.1 Safety-critical tasks on RT cores .................... 69
  5.3.2 Non-safety-critical tasks on RT cores ............... 71
5.4 Summary .................................................. 72

6 Future work and conclusions .......................... 74
  6.1 Future work ............................................. 74
  6.2 Conclusions ............................................. 74

Bibliography .................................................. 76

Swedish summary ............................................ 81

A Appendix .................................................... 87
  A.1 Simulations ............................................. 87
    A.1.1 Task migration to maximize overall performance .. 87
    A.1.2 Task migration to minimize overall energy consumption ... 89
    A.1.3 Migration of hard real-time tasks .................. 92
    A.1.4 Migration of soft real-time tasks .................. 95
LIST OF FIGURES

2.1 Insertion of virtualization layer in form of a hypervisor . . . . . . . . 5
2.2 Paravirtualized system using hypervisor . . . . . . . . . . . . . . . . 8
2.3 Task structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.4 Task lists on two CPU cores . . . . . . . . . . . . . . . . . . . . . . 11
2.5 Priority lists . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.6 Procedure how to allocate a task in a multi-core environment . . . . 13
2.7 Context switch with hypervisor . . . . . . . . . . . . . . . . . . . . . 15
2.8 Privileged call structure from application to execution on hardware . . 16

3.1 Task lists on two cores . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.2 Load representation on two CPU cores . . . . . . . . . . . . . . . . . 22
3.3 Load representation on two CPU cores . . . . . . . . . . . . . . . . . 22
3.4 Migration using re-creation . . . . . . . . . . . . . . . . . . . . . . . 26
3.5 Migration using replica . . . . . . . . . . . . . . . . . . . . . . . . . 26
3.6 Task re-creation vs replication [1] . . . . . . . . . . . . . . . . . . . . 27
3.7 Communication using shared memory . . . . . . . . . . . . . . . . . . 29
3.8 Communication using crossbar . . . . . . . . . . . . . . . . . . . . . . 30
3.9 Communication using point-to-point connection . . . . . . . . . . . . . 31
3.10 Communication measurements [2] . . . . . . . . . . . . . . . . . . . . 32
3.11 Comparison of migrations [3] . . . . . . . . . . . . . . . . . . . . . . 36
3.12 Polling mechanism for incoming tasks . . . . . . . . . . . . . . . . . 38
3.13 Communication during migration . . . . . . . . . . . . . . . . . . . . 38
3.14 Interrupt mechanism for incoming tasks . . . . . . . . . . . . . . . . 39
3.15 Master-slave model . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
3.16 Decentralized model . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
3.17 Measurement of overhead [4] . . . . . . . . . . . . . . . . . . . . . . 45

4.1 Thermal curve in component . . . . . . . . . . . . . . . . . . . . . . . 47
4.2 Model to calculate migration points . . . . . . . . . . . . . . . . . . . 48
4.3 Load generator . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
4.4 Measurement of migration scores . . . . . . . . . . . . . . . . . . . . . 50
4.5 Load generators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
4.6 Result scores . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
5.1 Power state machine .................................................. 58
5.2 Extended power state machine ...................................... 59
5.3 Migration overhead [5] .................................................. 60
5.4 Overhead measurements [5] .......................................... 61
5.5 Model of the states in an elevator ................................... 68
5.6 Load generators of hard real-time tasks ......................... 70
5.7 Two hard real-time tasks running on one CPU core ........... 70
5.8 Load generators of soft real-time tasks ........................... 71
5.9 Two soft real-time tasks running on one CPU core ............. 72

A.1 Block scheme to simulate maximum performance ............... 87
A.2 Subsystem defining the derivative of the curve ................. 88
A.3 Subsystem defining a cut-limit ...................................... 88
A.4 Subsystem defining a critical-limit ................................ 89
A.5 Block scheme to simulate minimum energy consumption ....... 90
A.6 Subsystem defining the derivative of the curve ................. 90
A.7 Subsystem defining the integral of the curve .................... 91
A.8 Subsystem running logical operations ............................. 92
A.9 Block scheme to simulate migration of hard real-time tasks ... 93
A.10 Subsystem used to detect overlap situations .................... 93
A.11 Subsystem used to detect overlap situations .................... 93
A.12 Subsystem used to delay task ..................................... 94
A.13 Subsystem used to delay task ..................................... 94
A.14 Block scheme to simulate migration of soft real-time tasks ... 95
A.15 Subsystem used to detect overlap situations .................... 95
A.16 Subsystem used to detect overlap situations .................... 96
Glossary

Atomic operation
An operation that will not be interrupted in the middle of the execution.

Co-routine task
A task that will always complete its execution before being switched out by the scheduler.

Context switch
The computing process of storing and restoring state (context) of a CPU so that the execution can be resumed from the same point at a later time.

Control feedback
To control a process by measuring the output of the system.

Critical region
Section of code where multiple accesses by threads is not allowed.

Debug register
Register used for real-time debugging for example to execute breaks in a debugging unit.

Deadline (real-time systems)
The instant of time by which a job is required to have completed its execution.

Granularity
The extent into which a system is broken down into small parts.

Hard real-time system
A real-time system where a deadline miss is not allowed.
Heterogeneous multi-core system
Multiprocessing computer system where all the processing units or operating systems do not necessarily need to be identical or of the same architecture.

HRT
Hard real-time task.

Instruction set architecture
A list of all the instructions with all their variations which a processor can execute.

Inter-core communication
Communication that is taking place between CPU cores in a multi-core system.

Mixed-critical system
Real-time environment with both a safety-critical part and a non-safety-critical part.

MMU
Memory Management Unit. A computer hardware component responsible for handling accesses to memory requested by the CPU. Its functions include translation of virtual addresses to physical addresses.

MPOS
Multi-Processor Operating System.

MPSoC
Multiprocessor configuration integrating all components of a computer or other electronic system into a single integrated circuit.

NoC
An approach to create a network of hardware nodes in a System-on-a-Chip (SoC).

NRC core
Non-real-time core.

NSC core
Non-safety-critical core.
Open-loop control
A type of controller which computes its output only based on its current state and its model of the system.

Position independent code
Machine instruction code that executes properly regardless of where in memory it resides.

Preemptive task
The act of temporarily interrupting a task being carried out by a computer system, without requiring its cooperation.

Response time (real-time-systems)
The time from where a job is released until it is completed.

RC core
Real-time core.

SC core
Safety-critical core.

Soft real-time system
A real-time system where a deadline miss is not desired, though possible.

SRT
Soft real-time task.

Task migration
Moving a task and its associated data to another location for further execution.

Time slice
The period of time for which a process is allowed to run in a preemptive multi-tasking system.

Wrapper function
A function in a computer program which main purpose is to call a second function with little or no additional computation.
1 Introduction

Multi-core systems have recently made their entrance into the embedded computing world. This new type of system requires software that supports its architecture. The problem of creating generic programs to multi-core embedded systems arises from the fact that different architectures require different implementation. The differences can appear when studying the communications channels between the cores, the connection to the memory and various hardware support, etc. The need to generalize hardware for the operating system is of the essence.

A virtualization layer can be inserted between the operating system and the hardware in order to simplify programming, improve the safety, enhance possibilities for minimizing energy consumption and preserve eventual real-time requirements. System virtualization has long been used in desktop and server systems, but is now also introduced into embedded systems. Virtualization for desktop and server systems is often used in order to provide the use of multiple operating systems on a single CPU, while virtualization for embedded systems is used to enable multi-processor support for a single operating system. Virtualization enhances hardware utilization and the adaptability of separating the operating system domain from the outer world. Virtualization of embedded systems can be used to help solve the aforementioned problem.

Lowering the energy consumption is becoming more and more important, and methods for achieving this goal are a part of this thesis. The method that has been focused on is a way of moving running tasks between cores in order to shut down idle core. This method, called task migration, is used to decrease or increase the energy consumption of a system by shutting down more or less CPU cores depending on the situation. The technology has existed in distributed computer systems for decades, and is now introduced into the era of embedded devices. A task migration mechanism depends highly on the hardware configuration, but virtualization of the system could help make the mechanism more generic.
1.1 RECOMP project

The RECOMP [6] (Reduced certification cost for trusted multi-core platforms) research project will establish methods, tools and platforms for enabling cost-efficient certification and re-certification of mixed-critical systems. RECOMP will provide reference design and platform architectures together with the required design method and tools for achieving the stated goal. This thesis reviews material that will lay the foundation of implementing operating system support for safe multi-core integration and core-to-core communication. The project is an international ARTEMIS joint undertaking project and is active in the years 2010 to 2013.

1.2 Thesis structure

The thesis reviews existing approaches which bring the reader closer to understanding the subject of virtualization of multi-core embedded systems. Chapter 2 brings the details about virtualization of embedded systems using a hypervisor. Virtualization techniques, hardware support and memory protection is, in this thesis, briefly explained. Chapter 2 also describes the real-time operating system FreeRTOS and its way of managing tasks and context switches, which is a first step in understanding task migration and its potential. Task migration, which encompasses moving a running task to another processing element for continual execution is an essential part of this thesis, and will presented in chapter 3. Task migration models, memory models and core-to-core communication are also described in this chapter; furthermore an overview of task migration and how it is achieved on different platforms, how the tasks are transferred and how migration decisions are made is also given. Chapters 4 and 5 presents ways of migrating tasks in order to reach high performance, low energy consumption and preserve real-time requirements. A crucial point regarding these chapters refers to not be able to reach all of these three objectives, but to configure a reasonable trade-off for all the constrains.
2 VIRTUALIZATION USING HYPervisor

2.1 Introduction to virtualization

Virtualization abstracts the physical resources for a user application or kernel in a system platform. The hidden resources are replaced with a virtualization layer that represents the hardware using an abstraction level. The concept of virtualization has been known for several decades. Virtualization simplifies programming because instead of manually dealing with the hardware resources directly, the program can be created for a generic platform. The virtualization technique multiplexes the underlying resources to all the applications in a transparent fashion. The virtualized resources acts identical to the corresponding resources which leads to no limitation in the programming itself.

Virtualization is possible on both operating system level and hardware level. Software like VmWare and QEMU run a complete operating system as an application on the host operating system. Virtualization of hardware such as the MMU eliminates fragmentation by mapping a contiguous range of virtual addresses to the physical memory. The blocks in the physical memory do not have to be in a contiguous range as long as the pointers from the virtual memory are. This makes memory access in programming simpler. The program language Java is an example of complete machine virtualization. The language itself operates on the Java virtual machine that makes the calls to the physical hardware. Furthermore, modern research regarding virtualization is referenced from [7].

The client using virtualization for making access to underlying resources is called a guest. The guest is requesting resources from the host that also could be acting as a guest for the next level of virtualization.

This thesis focuses on system virtualization, which is a method of complete hardware virtualization. Virtualization of this kind makes it possible to run multiple operating systems in parallel. The operating system itself is not aware of the other operating systems running in parallel, but an underlying virtualization layer handles a communica-
ations channel between the operating systems. The purpose of this is to run instances of an operating system on each core in a multi-core embedded environment and make resource sharing and communication between these cores possible.

System virtualization provides isolation of the operating systems which leads to robustness and security since the kernels and the applications on different cores are not able to freely interact with each other. A system virtualization also prevents external attacks on the system because all outside communication with the applications must pass the virtualization layer that runs on the highest privilege level. The use of different privilege levels is a way of determining the permissions of the different software parts of the system. Virtual machines may also encourage a mixture of different architectures on the same platform [7], which can be done by translating all communication into a generic language which all devices are capable to understand. The single most important issue in this thesis is to address the possibility of task migration in mixed-critical systems by the use of virtualization.

2.2 Virtualization techniques

Virtualization methods can be divided into several approaches, but the end result is always a virtual system on which software can run. This section gives an introduction into several techniques to achieve virtualization. Initially different methods are briefly presented, followed by the construction of a virtualization layer called the hypervisor and lastly hardware support for virtualization.

The ISA translation method [8] is a way of translating the generated instructions into the hardware supported instruction on-the-fly. This means that a generic instruction can be run on several hardware architectures as long as the translation layer supports the selected architecture. The translation is only necessary if the guest and the host utilizes different instruction set architectures.

Paravirtualization [9], which is our approach to virtualization, is a way to modify the guest operating system interface in order to pass privileged call to the underlying virtualization layer. The modified interface will be used for critical and privileged instructions, such as core-to-core communication, critical regions access and important setups. Paravirtualization is only possible if the operating system allows modification and if it is continuously maintained. A third method called pre-virtualization [10] aims to combine the performance of paravirtualization and the flexibility of ISA translation.
2.2.1 The hypervisor

A complete system virtualization can be achieved by using a software layer between the host operating system and the hardware. This layer, which is called the hypervisor, handles privileged calls from the operating system to the hardware. The hypervisor is a layer of software that performs system virtualization, and thus using the virtual machine as a system abstraction. The hypervisor could be seen as a virtualization layer for the whole system, as seen in figure 2.1 although it is running on all CPU cores. In a paravirtualized system the operating system is aware of the underlying software layer, and will therefore be modified in order to support hypercalls, which are special defined calls to the hypervisor. The hypervisor is running on a higher privilege level than the operating system, because the aim is to have the hypervisor take control from the operating system at certain points. This system structure would be easier to achieve with certain hardware support, such as different privilege levels in the memory.

![Figure 2.1: Insertion of virtualization layer in form of a hypervisor](image)

The objective of our research is to create a hypervisor layer for a multi-core embedded system such that the system looks to the user like one operating system with tasks distributed internally, although all CPU cores actually run one instance of an operating system and a hypervisor. This implies that the hypervisor needs to support communication with other hypervisors on other CPU cores. The following goals should be reached in order to create an efficient hypervisor:

- The hypervisor will provide a mechanism that allocates tasks, created by the operating system, on the appropriate target core. The possibility of choosing core may vary depending on the user and the functionality of the task.
• The hypervisor will offer the possibility of *migrating* tasks between multiple CPU cores.

• The hypervisor will enable load monitoring and sending load statistics between cores.

• The hypervisor will not affect the functionality of the tasks and the system regarding their correctness.

• The hypervisor virtualizes the CPU cores at the cost of an acceptable performance.

### 2.2.2 Hardware support for virtualization

The implementation of a virtualization technique may become easier and more robust with certain hardware support. The grade of hardware support depends on the selected architecture — and should be considered before the virtualization design begins.

Arm TrustZone [11] found in ARM 11 and ARM cortex-A processors is a technique to make two separate *worlds* in a single core processor. The worlds are a virtual representation of the CPU core, which are able to run in parallel. One of the worlds is the *secure world* while the other world is the *normal world*. The worlds act independently as they where two different CPU cores; one core for the normal functionality and one for the secure functionality. Resources such as interrupts and peripherals can be assigned to a selected world. The secure world can access the resources assigned to the normal world, but not vice versa [10].

Both Intel and AMD also provide their own extension set Intel-Vt [12] and AMD-V [13], to support virtualization on the aforementioned architectures.

### 2.3 FreeRTOS

FreeRTOS [14] is a small real-time kernel ported to many popular architectures. The kernel supports a real-time scheduler on top of which applications can be scheduled with *hard real-time* requirements. The tasks are scheduled and selected for processing depending on their priority. Configurable task stack size and system tick interval is also supported in FreeRTOS. An API is provided to the user for task allocation, message passing, load measurement, etc. There are also features like semaphores, priority
assignment, hook functions and security checking implemented in FreeRTOS [15]. FreeRTOS is free to use in commercial products as long as the documentation states that the software is running on FreeRTOS. The fully commercial version of the operating system is called OpenRTOS and includes commercial support from the providers. In this thesis the operating system references will be related to FreeRTOS.

### 2.3.1 The kernel

The FreeRTOS kernel is a real-time scheduler that switches in and out tasks depending on their properties. The scheduler is driven by a timer interrupt that is set-up according to the platform specific settings. The time slice frequency as well as other functionality are set in a platform specific configuration file. A task in FreeRTOS can run in both preemptive mode and co-routine mode. The preemptive tasks are allowed to be interrupted by a task of higher priority while a co-routine only will make a context switch when a task finishes its execution. The timer interrupt makes a port specific yield call which switches in the next task, and then returns. The context switch inside yield-method works as follows:

- Save the platform specific content
- Increment the tick
- Select a new task from the task lists
- Restore the new platform specific content
- Return to the new task

The platform specific part of the code must support the platform on which FreeRTOS is run because this part usually includes assembler instructions executed directly on the CPU. Using the FreeRTOS scheduler begins with creating a task. The task is a function that executes user defined code, and is then linked to the scheduler with a handle, task name, priority and stack size. When the selected tasks are created, the scheduler starts. The scheduler will automatically create an idle task when the first context switch is taken place. The idle task is a task with the lowest priority that will run when no other task is in the ready state, i.e. when no other task requests the CPU.

The FreeRTOS kernel consists of three generic files: list.c, queue.c and tasks.c. These files describe the task list structure, inter-task communication and
the tasks themselves. Alternatively a file `coroutine.c` can be used instead of preemptive tasks if the platform supports very limited amount of memory. Any one is allowed to create the portable part of FreeRTOS, which is platform specific code that the generic code occasionally will call.

### 2.3.2 A hypervisor for FreeRTOS

The virtualization layer called the hypervisor will be inserted between the FreeRTOS and the hardware. The code in FreeRTOS should best not be tied to the hypervisor in order to be as generic as possible. However this is not a feasible achievement. The operating system must explicitly make the hypercalls to the hypervisor and thus make use of its particular interface. The FreeRTOS must therefore adapt to the underlying hypervisor by the use of a slight modification. As seen in figure 2.2, the hypervisor is inserted on all cores and thus functions as a part of the operating system. The structure in figure 2.2 is the one used in a paravirtualized system.

![Figure 2.2: Paravirtualized system using hypervisor](image)

When using a hypervisor, the privileged code in the platform dependent part of FreeRTOS has to be executed via a set of hypercalls [7]. Privileged code and privileged functions are certain specially defined code that can be run only with the highest permissions. The hypercalls call platform dependent code that FreeRTOS uses for privileged functions e.g. enter/exit critical part, setting up system timer, manual context switch etc. These calls will be handled by the hypervisor and executed only on request from the operating system or the user. A separate part of FreeRTOS contains all the platform dependent code. This code cannot be made generic because of the architec-
tural specific instructions that are part of the code. When a hypercall is made from the operating system, the control is taken by the hypervisor. The hypervisor is a virtualization layer, which will for example handle the task migration and the transfer of load statistics.

2.3.3 Memory protection

A hypervisor must run in a privileged mode while the guest OS runs in user mode, which is a mode that has been granted lower permissions than the privileged mode. This has been done in order to protect hypervisor memory from being accessed by the OS kernel. The memory is divided into application, kernel and hypervisor memory; parts that could be referred to as memory domains. Hypervisor domain is only accessible in privileged mode which is best provided by hardware support on the platform. ARM TrustZone [11] hardware gives for example the system two distinct memory privilege levels. One level is used for user and kernel domain, and the other is used by the hypervisor. In FreeRTOS there are no system calls, but only library calls to the functions; therefore tasks and the kernel run on the same privilege level [7]. In a hypervisor implementation the tasks should be able to call the other domains, but they should not be able to tamper with the kernel memory nor the hypervisor memory domain. FreeRTOS provides a memory protection unit (MPU) for some of its ports. The MPU-defined code is reached by a call to a wrapper unit. This wrapper unit disables the functions from altering the kernel memory, while calling kernel functions is still possible. The tasks should not access each other’s stacks and neither should non-kernel code execute in kernel mode. Peripheral devices such as UART interfaces etc., can be set to be used only in privileged mode [14]. The user defines a memory region for a device, which makes the device either a standard peripheral or a system peripheral. System peripherals run in privileged mode, and are thus only accessible inside the hypervisor. By using separate hardware memory levels and the MPU, a basic memory domain structure can be obtained.

2.4 Task creation in FreeRTOS

A task in FreeRTOS is a piece of software that holds a collection of methods which can be scheduled on the CPU. Figure 2.3 shows how a task is divided into two separate parts that are allocated in the memory heap. Each task can be seen as a small program
that is running independently of other tasks. A task consists of a Task Control Block (TCB) and a task stack. The TCB is a structure that holds vital information about the task. The information such as a function pointer to the task (called task handle), name of task, size of the task stack, priority and various parameters is stored in the TCB. The TCB is constant in size for all tasks and has to be allocated for all tasks.

The stack can be allocated in different sizes depending on how much memory a certain task needs. The stack is used to store variables used by the running task. Tools in FreeRTOS could determine if a task uses more memory than it has been allocated for, which could happen if the stack grows too large. Tasks are typically created when the system is initializing, after which the system goes into an infinite loop and tasks are being switched in for processing by the scheduler. A task can either be running or not-running [14],[15]. A running task is being executed by the CPU while all the other task are not-running i.e. not executed. The not-running-state can be divided into sub states which describe the reason a task is not running at the moment. A blocked task is a task that either has called a wait function or is waiting for data to arrive from another task. A task in this kind of state will try continuing to execute as soon as its waiting period has ended. A suspended task will not run until the user has resumed the task. The suspension is only done by the vTaskSuspend() function which is intended for tasks that do not need to be used for the moment. Finally the ready task is a task that is not blocked or suspended.

A ready task will be switched in for processing as soon as the current priority will exceed the other tasks in the ready state.

The idle task is always ready in FreeRTOS after the scheduler has been started. An idletaskhook, which is a defined function, can be used to make the idle task execute instructions. These instructions are only executed when no other task is being executed because of the lowest priority of the idle task. Priorities can be updated on-the-fly for all tasks except the idle task.
2.4.1 Task list

Tasks in FreeRTOS are stored in lists from where they are selected for execution. Every instance of FreeRTOS holds one list of tasks that currently are allocated on the CPU core (figure 2.4). This means that in a multi-core environment there will be one task list per CPU core.

As a task is created, the properties of the task are stored in a Task Control Block (TCB). This block is inserted into an array of pointers that could be seen as a list with different task priorities as shown in figure 2.5a and 2.5b. A task with the highest priority is stored in the first list while a task with the lowest priority, for example the idle task, is stored in the last list. When the scheduler switches in a task for processing, it will always start checking the first list to see if there are any ready tasks present. The scheduler scans the lists downwards until a ready task is found; this task is then switched in. The scheduler will only accept tasks that are in the ready state since they are the only ones that are ready for execution i.e. not blocked or suspended. Deletion of a task means that a task is completely removed from the task list, and the memory associated with the task stack is freed. Tasks are only deleted upon request by user.
2.4.2 Task creation in a Multi-Core system

In a multi-core system running FreeRTOS, every core runs one instance of the operating system. Task creation in a multi-core environment can be described as allocating tasks on any core on the platform. The reason for making the task allocation possible in a multi-core system is e.g. to be able to distribute tasks between the cores. The tasks are moved between cores by the use of a task migration mechanism. This matter will be discussed in the forthcoming chapters.

Creation of a task suitable for a multi-core environment requires knowledge of the surrounding system. The task can be selected for allocation on another core, which means that the task is being migrated to the other core after creation. The task also needs the correct structure and permissions to be able to perform such an action. If the task is created completely unaware of the other processing units, the scheduler will assume that it is being run only on the native core. The task can however be migrated onto another core later if migration permissions are set and if the nature of the task makes such an action feasible. When a task is created for a multi-core environment, the creator of the task uses a set of defines to create a task suited for this platform. The system can use a wrapper unit (listing 2.1) to create a generic task from the multi-core specific demands.

```
#define createTaskMultiCore(param...) createTaskGeneric("
#define createTask(...)
```

Listing 2.1: Definition of wrapper unit

This wrapper can similarly be included in the MPU wrapper which is intended to create a generic task independent of MPU or multi-core settings. As a task is created, different
situations occur depending on the behavior of the task. A parameter can describe what kind of multi-core configuration is present. Different situations can occur as follows:

- If the parameter is NULL then the task is inserted into the native task list.
- Task with parameter = core1 (native). If the parameter is the name of the native core then the task is inserted into the native task list
- Task with parameter = core2 (not native). If the parameter is the name of a non-native core then the task is migrated.
- Task with parameter = chooseBestCore. If the parameter is to choose the best suitable core for the task, then a check is run and the task is migrated if the result is not the native core. Else the task is inserted into the native task list.

The check needs to determine which core is best suited for the task. The result depends on what task and what properties are currently prioritized by the system. The check must gather information from other cores in the system, which means that it must be run in the hypervisor (figure 2.6). The hypervisor can communicate with the other cores since it handles the core-to-core communication.

If a task is manually selected to be run on a certain core, the task is considered to be user migrated. User migrated tasks only take place if this function is supported; otherwise the system takes care of the migration. As explained, the task that has been created on a specific core can either be migrated directly by the migration mechanism or be allocated on the native core. If it is not allocated on the native core then the
migration mechanism will migrate the task after a certain interval (see chapter 3.5). This interval is dependent on the priority of the task, communication delays and the mechanism that migrates the tasks.

Task creation for multi-core systems can be similar to single-core systems by the use of a wrapper unit which automatically creates the task for the selected system. Task allocation in a multi-core environment is not trivial because the tasks must be able to migrate to other cores in the system on demand. Task migration is discussed in chapter 3.

2.4.3 Interfaces for creating tasks

The interface for creating a task for multi-core systems must also support the original way of creating tasks i.e. creation of tasks for single-core systems. The user (the creator of the task) can modify the tasks using for example a set of defines to generate multi-core specific tasks. In single-core cases the user would run the normal createTask() method which is later wrapped into a createTaskGeneric() method. In the multi-core case the user would run the createTaskMultiCore(parameters) method which is later wrapped to the same createTaskGeneric() method. The difference is that the system is now aware of the multi-core environment, and can select an appropriate destination core depending on the input parameters. The system can in this case measure which core is best suited to allocate the task on. The second possibility is that the creator of the task states which CPU core he/she wants to use or prefers. Such approaches can only be used if the system allows migration by the user.

A task could be defined as static, meaning that it cannot be migrated at all. Eventually the kernel or super user with the right permissions could be able to migrate a static task. A task could by definition not be migrated because of interaction with the surroundings e.g. external interrupts. These types of tasks must be distinguished from other tasks with explicit parameters or automatically by the system.

2.5 Context switch using hypervisor

The context switch changes the currently executing task on the CPU core. A context switch interrupts the currently running task and stores its states. After this a new task is switched in from the task list to run in the operating system. The state of the task
changes to a not-running state while the new task changes to the running state. The choice of this new task depends on properties such as suspension, deletion, entrance of a blocking state or the priority in the task list.

When a context switch is taking place in FreeRTOS (figure 2.7) the scheduler starts checking the task lists for a task that is in the running state — following the assumption that the scheduler is not suspended. Whenever a possible task is found, the TCB information is loaded into the CPU and the operating system starts executing the task exactly from where it was previously interrupted.

![Figure 2.7: Context switch with hypervisor](image)

A system with a hypervisor will have to execute the platform dependent code inside the hypervisor instead of inside the kernel (figure 2.7), because as the hypervisor is running on the highest privileged mode it is able to take the control from the operating system. The control need to be taken in cases of executing CPU specific instructions such as entering a critical region or disabling user mode interrupts [7]. Figure 2.8 shows an example of a privileged call that explicitly uses the underlying CPU. As soon as the privileged code is executed the control is returned to the operating system that handles the platform independent code. Depending on the implementation, a hypercall could also be made from within a task. Such a hypercall is made only if no kernel code is needed to be run, for example if a task needs to enter a critical region in its code.

A task can be defined to run in user or privileged mode [14]. A user mode task in FreeRTOS has only access to its own memory stack and up to three defined memory
regions, which are defined upon creation of the task. Different permissions can be set to these distinct regions, such as read-only or read/write etc. No data memory is shared between user mode tasks, instead all user mode tasks communicate via message passing using message queues. Privileged tasks have nevertheless access to the whole memory map of the system. System calls to the FreeRTOS API is located in such a region that only a privileged call will access it. A privileged task in FreeRTOS is able to set itself to user mode, though not the other way around once entering the user mode.

When the privileged code has been run, the hypervisor gives control back to the kernel which runs the platform independent code. The platform independent code consists in this case of the traversal of the priority list that selects the next task. When the task is found, a hypercall is made to the hypervisor which restores the CPU states. The hypervisor will then return control to the task. An example of this procedure is shown in the following code listing:

```c
while(1){...}
```

Listing 2.2: Task1
ISR_timer_int()
    //execute platform dependent code
    saveContext();
    disableUserModeInterrupts();
    //return to kernel handler function
    ....
    DoYield();
    asm volatile ("reti"); //return from interrupt
}

Listing 2.3: Hypervisor interrupt handler:

void vDoYield()
{
    storeContextTCB();
    //read-only access to saved context
    switchContext();
    vRCON(parameters); //restore context
    asm volatile ("ret"); //return to task
}

Listing 2.4: OS interrupt handler:

void vRCON(_param_p)
{
    //execute platform dependent code to restore content
    lds r27, currentTCB
    ld r28, x+  
    out SPL, r28
    pop r31
    ...
    pop r0
    out SREG, r0 //CPU state restored
}

Listing 2.5: Hypervisor restoring task

A context switch using a hypervisor will run all privileged code inside the hypervisor. The hypervisor is called when the system tick interrupts the currently running task. After the new task has been switched in for execution, the hypervisor returns control to the task.
2.6 Summary

Virtualization of embedded multi-core systems can increase safety because of the isolation of different memory domains. Privileged code can be run in separate domains which cannot be altered by the user domain code. Virtualization also simplifies the programming and enhances the use of core-to-core communication. Communication in a multi-core embedded system is needed for the CPU cores to cooperate and distribute tasks internally. The virtualization in form of a hypervisor is inserted into the already existing FreeRTOS operating system by re-routing privileged calls to the hypervisor level instead of the operating system level. The hypervisor should primarily be able to allocate tasks in a multi-core environment, control the context switch and handle communication with hypervisors on other CPU cores.
3 Task Migration

3.1 Introduction

Multi-core embedded systems are getting more and more popular on the market. In contrast to the popular SMP structure in the desktop computers, the embedded multi-core platforms tend to have a more heterogeneous and asymmetric approach [7],[16]. This means that instead of all CPU cores being identical and use a shared memory, the CPU cores could have a different architecture and use a single local memory from which the software is run. An asymmetric system runs one instance of the operating system on each core. This increases the scalability since all cores can access the memory completely in parallel.

Task migration in asymmetric multi-core systems is a method of moving running tasks between CPU cores, and allocating the task in the target core’s local memory. The task that is being migrated will shut down its execution, and continue executing after the migration has been performed. The execution will start exactly from where it was interrupted before the migration. The support for task migration between CPU cores is vital when attempting to optimize performance, energy consumption, reliability and cost. In order to make the task migration effective, the need to minimize involved overhead is essential. Overhead is introduced when the task is physically moved, when decisions are made and monitors are run etc.

To be able to design a task migration mechanism, a decision on a model overview is clearly of importance. The model must be adapted to the real world situation depending on the platform and memory requirements, but also the way of handing communication between cores is a non-trivial decision that needs to be handled in the planning stage.
3.2 Overview

A running task is a piece of a program code that is currently being executed on one processing element of the system. A system with one processing element can schedule multiple tasks, but only process one task at the time. The migration of a task is stated as moving an allocated task from one processing element to another, which could be a CPU core connected to other cores using some kind of communications bus.

Besides task state, the CPU state, task stack, heap variables and program code must also be completely moved to the target core [17]. Figure 3.1 shows a task being migrated from one core to another. The migration can be modeled as moving the task from one task list to another, although the real transfer is more complicated. There are different ways to model task migration. In order to cope with all these different cases, the next section will show a generic overview of the migration mechanism as well as when, and why task migration is needed.

![Figure 3.1: Task lists on two cores](image)

### 3.2.1 Starting a task migration

Task migration is a call to the operating system (or the virtualization layer) during run-time. The call can be made from both the user layer and the kernel layer, but all migration callers will be treated as a generic source regardless of the layer. Task migration is usually started at selected checkpoints in the program (see section 3.5). When a call is made and the migration is granted by the target, the operating system stops the
currently running task by suspending it. The data associated with the task is stored and moved onto the target core by the migration mechanism. The operating system can, if the migration was successful, manually switch in a new task for processing. If the manual context switch was not made, the operating system will switch in the new task after the next system tick.

Task migration must be supported by a library containing essential functions that makes a task migratable. Enabling migration for a task means using the provided task migration library and the related paradigms as a programmer. An arbitrary created task will probably not be used by the migration mechanism efficiently, in the same fashion as a single threaded program will not make use of multiple processing elements. This means that the developer should be aware of the resources and regulations at hand in order to create a program suitable for task migration.

In case migration is not wanted, the tasks are declared static and cannot be migrated by any user or kernel [3]. The static declaration can be stated at the creation of the task, or dynamically during run-time depending on the implementation. Tasks that have dedicated hardware drivers or specific user manager programs etc., should by definition not be migrated — and are thus declared static by the programmer or automatically. Tasks with strong communication dependencies or dynamically changing dependencies can pose severe problems using task migration due to the re-routing of resources [18],[19]. Hazards like these should be investigated before the creation of a task to prevent the system from misbehaving or having ineffective execution.

In our model, the migration mechanism is located in the hypervisor. The hypervisor has full control of the operating system and can communicate with other cores using a communications channel. This means that a migration call will always be handled by the hypervisor. After a migration is performed, the hypervisor gives the control back to the operating system, which continues to run as normal.

### 3.2.2 Reasons for task migration

Tasks are migrated between cores in order to achieve dynamic load distribution, ensure fault resilience, provide system administration, minimize energy consumption and thermal chip management [3],[4]. A fully loaded CPU core can migrate tasks to a lighter loaded core to increase the overall system performance. Performance improvements will show clearly in situations when a single core is loaded over its capacity as in figure 3.2. A less loaded core can be handed some of the first core’s tasks in order
to increase the overall performance in the system.

![Figure 3.2: Load representation on two CPU cores](image)

Task migration can also be used to migrate all tasks to one single core in order to decrease the energy consumption, which is shown in figure 3.3. Energy consumption is a very important concern both in embedded systems, workstations and servers. Less energy consumption means longer battery time, less generated heat, lower electric bills and is beneficial for the environment. A CPU core consumes more energy fully loaded than it does idle. An energy saving strategy comprises shutting down idle cores, and keeping them shut-down until they are needed. When a core is shut down it consumes the least energy [20]. Two cores with light load could, combine the load into a single core with all the tasks running on it, while the other core could be shut down.

![Figure 3.3: Load representation on two CPU cores](image)

The objective is to control the task distribution according to an applied policy. The main advantages of implementing this feature is to have a good utilization of CPU cores and to improve the systems’ efficiency [16]. Energy policies involve latency, throughput, communication delay, wake-up times etc. [20] This matter will be discussed in chapter 4.
Task migration in a real-time system can prevent a low-prioritized task from being preempted by a high-prioritized task, which will result in a faster execution time for the low-prioritized task if it is migrated to a core without high-prioritized real-time tasks. Tasks that overlap each other’s cache lines will also benefit from one migrating to another core [21], because cache misses could eventually be reduced which leads to less memory operations.

Task migration is in summary primarily used to increase performance and decrease energy consumption.

3.2.3 Load balancing monitor

A load balancing monitor is a piece of software that, on basis of the CPU statistics from one or many cores and an energy policy, makes the decision of migrating tasks from one core to another. The load balancing monitor is a process that runs on at least one CPU core in a multi-core environment. The load balancing monitor considers all trade-offs and overheads provided by the nodes when making a migration decision. The heavy overhead of moving the tasks physically is one of the main concerns when making a decision. The overhead regarding the delay of communication between tasks is also significant. A task that communicates highly with other tasks is less likely to be migrated because it is either declared static, or has a low migration priority. As illustrated by table 3.1 the extra overhead introduced when communicating over inter-core channels is significantly higher than communication inside one single core.

The monitor is the process that checks the statistics from all the cores, and makes a migration decision. The monitor could either be located globally in one core or locally in every core in the system (chapter 3.5.6). The complexity of the monitor as a decision making mechanism depends on the available resources. A too complex monitor running on a system with very restricted resources will not be beneficial, since the monitor itself allocates most of the resources.

3.2.4 Stages of task migration

Assuming a migration is taking place between two CPU cores. The target core must have all the necessary information needed for resuming the migrated task, and all the information associated with the task e.g. communication between tasks. This information is sent to the target core either by means of temporal storage in shared memory.
or some other way of direct communication. The following information is needed to resume a migrated task:

**CPU state**

A task needs to know its CPU state, in both a context switch and in a task migration. The CPU state is easily read from the CPU status register and is stored in a temporary variable. This variable can be migrated onto the other CPU core.

**Task stack**

The task stack holds the variables used in the task and the values of these. The task stack has a similar structure as the system stack and is allocated in the memory heap. The task stack is fairly easy to migrate because the system knows where the stack begins and its size. A similar stack could then be allocated on the other core, and the variables could then be restored.

**Heap data**

Data allocated in the heap is difficult to migrate since the whole core uses the same heap. Variables associated with the relevant task must be somehow marked to inform the migration mechanism which data to migrate. Problems could eventually occur when determining the size of dynamically allocated arrays in the heap. Assuming there is a mechanism for linking heap data to tasks, the data could be migrated and re-allocated on the other core. The pointers to the data must be translated upon such a re-allocation.

**Program code**

Another difficult part to migrate is the executing code in the program memory. The program code is statically stored, and must also be marked with associations to the relevant task. The program counter in the system must continue exactly from the migration point relative to the program. The migration of the program code is also substantive when considering the run-time update, which would switch in-and-out program code while a task is running. The migrated code should, in summary, continue its process where the first core was stopped.
Task associations

Various associations exist between tasks, such as information exchange, calls, child tasks etc. A task is able to communicate with other tasks using message queues [14], which can only be used if the tasks are executing on the same core. Otherwise the interface of communication must change, so that the tasks communicate over some kind of inter-core communication. Task calls between cores must be handled either in a way that the tasks pass values between cores, or so that called tasks also migrate to the same core as the caller. This could of course pose problems with other tasks that call the same task.

Cache memory

To speed up cache searches and to eliminate the cold start of cache memory, possible to migrate the whole cache memory or cache areas is needed. The assumption is that the system has private L1 and L2 cache memory for all CPU cores.

Global variables

Several tasks could use the same global variables, which leads to a synchronization problem. This could possibly be a part of task associations, where the variable itself is not difficult to migrate. For security reasons tasks that are selected for migration should preferably not share global variables with other tasks, since the communication using global variables could be broken or slowed down if one of the tasks is moved to another core.

3.3 Task migration models

A task migration model is a way of showing what actually happens during a migration on a higher level of abstraction. The model shows which actions need to be taken, and what platforms support such actions.

3.3.1 Re-creation

The re-creation model [3],[16],[22],[17] is a strategy where a task is initially only created on the native core. The task is completely copied to the other core’s memory,
and started from the point where it stopped. Figure 3.4 shows the migration taking place. After the migration is completed, the task is killed and deleted on the first core. Supporting task re-creation on a platform with no MMU poses problems [5] because of possible errors in the function pointers, which must be taken into consideration when migrating a task. To support task re-creation on a platform with no MMU, a *position independent code* [1] must be used, as the starting address of the process memory can change upon a migration. The selected architecture must also support position independent code.

![Figure 3.4: Migration using re-creation](image)

### 3.3.2 Replica

The replica strategy [4],[5] solves the problem encountered when having no MMU. When a task is created on one core there is also a replica of that task created on the other cores. The replica tasks are stopped whilst the original is in the running state.

![Figure 3.5: Migration using replica](image)
When migrating the task, the program memory does not need to be moved to the other core, instead the replica task gets its starting point and starts running exactly from where the original task was suspended (figure 3.5). A migratable task uses more program memory since it is created on all cores in contrast to the re-creation model. This memory requirement does not apply for non-migratable tasks since they are only created on the native core.

In the article [1], the authors conducted an experiment in order to compare the overhead between task replication and task re-creation during a migration. In the test, various tasks of different size were migrated using the two strategies. The configuration of this test was two RISC based processors with a private memory for running a private instance of uCLinux and a shared memory for communication. The migration decisions were made based on thermal measurements inside the CPU cores. The goal was to even out the thermal dissipation of the whole system by migrating tasks between CPU cores. The amount of clock cycles a migration needs was measured with different tasks sizes.

The result shows in figure 3.6 that small tasks have about the same migration overhead regardless of the migration method that is used. The black line represents the re-creation of tasks and the gray line represents the replication. As the task size increases the replication method’s overhead as a function of task size increases slower than the re-creation method’s. The result is rather obvious, when comparing the models, due to the fact that the replication method does not need to transfer any program code.

Figure 3.6: Task re-creation vs replication [1]
3.4 Task migration communication

One of the main concerns in task migration is the communication between CPU cores. Core-to-core communication is needed to set-up a migration, to send statistics on which core to schedule, to accept or reject a migration and for various other signals to be sent between the cores. The means of communication depends to a large extent on the structure of the platform, such as having common or local memory. The difference between a common or local memory location reflects largely on the choice of communication method. A system with a common memory location is not in the need of having a dedicated communications line between the cores, while a system with only local memory is. The performance of a MPSoC is largely dependent on the communication between components [23]. This issue will be reviewed in the following section, after which examples of different means of communication are presented.

3.4.1 Inter-Core communication

Inter-core communication is the communication taking place between CPU cores inside the very chip. The way the communication is taking place depends on hardware, software and protocols available. Inter-core communication is a vital part in task migration.

When considering task migration on a heterogeneous MPSoC platform, the migrating tasks are sent, in theory, to any core. In order to ensure success, the cores could either use some kind of point-to-point communication, share a communications bus or share a memory location [5]. The communication is used for sending messages concerning the migration between the CPU cores. When a task is about to migrate, a core has to decide where to migrate the task. This decision must be made based on some statistics from all or a part of the CPU cores in the system. The cores need to communicate in order to make such a decision [4],[3]. A protocol has to be set-up to support inter-core communication, which will be used when CPU cores are exchanging information such as transfer of statistics, acceptance and rejection of tasks, package transmissions and various other signals.

Different methods must be implemented depending on what hardware support is present. The task migration mechanism itself is also highly dependent on which communication model is in use, because the algorithms handling the messages and statistics must match the selected model.
3.4.2 Shared memory

A shared memory is a memory located so that all CPU cores can access it (figure 3.7). This shared memory is not intended for storing program code, but to handle common data used as communication between CPU cores. The applications containing the program code running on top of the operating systems are located in the local memories of the CPU cores.

S. Bertozzi & Co. suggests in [3] a shared memory containing a migration table, in which information about migration requests is stored. The separate CPU cores read this table regularly to check if the current task has got a migration request, which is made by a separate monitor process that read statistics from this same memory location.

Acquaviva & Co. explain in [5] a method of accessing the shared memory location. “To use shared memory paradigm, two or more tasks are enabled to access a memory segment through a shared malloc”. This shared memory allocation routine returns a pointer to the shared memory area. The stated mechanism is needed because the default OS is not otherwise aware of the shared resource. Besides this memory allocation extension, a synchronization mechanism must be included to prevent resource conflicts in the shared memory. The communication needed to access the memory must nevertheless be kept light because of the shared memory bottle neck [4]. The same observation must be done for a completely shared bus without any shared memory.

Experiments in [4] reviews a decentralized model which uses shared memory and performs task migration with a replication mechanism. The test regarding the communication overhead shows the difference between a local call to a OS (single core), a local call to a MPOS (Multi-Processor Operating System), and a remote call from another CPU core (in a MPOS). The calls concerned the usage of mu-

<table>
<thead>
<tr>
<th></th>
<th>local OS call</th>
<th>local MPOS call</th>
<th>remote MPOS call</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>31.38 µs</td>
<td>51.19 µs</td>
<td>1296.9 µs</td>
</tr>
<tr>
<td>PEND</td>
<td>32.85 µs</td>
<td>53.02 µs</td>
<td>1297.41 µs</td>
</tr>
<tr>
<td>CREATE</td>
<td>30.74 µs</td>
<td>104 µs</td>
<td>1295.27 µs</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>30.65 µs</td>
<td>51.32 µs</td>
<td>1295.27 µs</td>
</tr>
</tbody>
</table>

Table 3.1: Overhead measurements [5]
flags and mailboxes. Table 3.1 shows the communication delay of different calls.

By using the local OS call as reference, one can state that a remote call to another CPU core uses significantly more time due to the communication delay. Even the local call in a multi processor environment (MPOS) will delay the communication internally because of some extra functionality needed for multi processor support. The specification for the occurrence of this delay was not stated in the article.

### 3.4.3 Crossbar

A crossbar is a unit which is able to handle multiple connections inside a shared bus. Figure 3.8 shows a crossbar-type of communication. Using this configuration the CPU cores pass messages directly to the target core without using a shared memory or completely shared bus.

![Crossbar Diagram](image)

Figure 3.8: Communication using crossbar

A crossbar [24] connection is able to handle multiple messages simultaneously by switching on and off the corresponding connection inside the crossbar. The sender of a message has a certain target destination associated with the message. The message is sent to the target via a bus called the crossbar. A *switch* unit is associated with the crossbar, which is a uni-directional connection between nodes. The switch controls the package flow with simultaneous connections inside the crossbar. The crossbar also has a *controller*, which chooses, depending on its input, which crossbar channel to switch on [2]. This makes sure that the packages will reach the correct destination.

A way of routing packages in a larger network is communication using a Network-on-Chip (NoC). A NoC controls packets by a routing algorithm that routes the packets to the determined destination in an Ethernet-like network. Routers are implemented on the chip that either send the package to the respective CPU core, or to the next router.

Task migration messages, variables and program code is sent using the crossbar to the respective destination. Additional logic must be implemented to support the
crossbar controller. The implementation of a crossbar will not be a part of this thesis.

3.4.4 Point-to-point communication

In point-to-point networks the number of nodes and connections between them are the same as seen in figure 3.9. The assumption to make messages available to all cores is not feasible here, since there is no common communications component and also no resource sharing. This strategy reaches the highest possible communication performance according to the experiments performed in [2].

![Figure 3.9: Communication using point-to-point connection](image)

Using this model some task migration strategies cannot be implemented in the same way as in the shared memory case. Statistics from each CPU core cannot be written into a single table in a common location, but has to be sent from all the separate cores individually. Sending signals and messages between cores is straightforward in the way that the transmitter always knows where the destination core is. Semaphores and similar mechanisms that were used in the shared memory location are not essential in the same way.

Nikolov & Co show in [2] a comparison between different communications techniques using an asymmetric MPSoC platform with 8 processors. The first test took place on a MicroBlaze processor platform. The algorithm processed a 128x128 pixel image, after which the elapsed time was measured. The two first configurations were: communication by crossbar and by point-to-point communication.

The work load was intended to distribute over the whole system as efficiently as possible. Figure 3.10a shows the amount of clock cycles required for the calculation, with a hardwired timer on each platform used to measure the time a test used. This test shows a clear advantage of using point-to-point communication. The point-to-point communication method does not interact at all with the rest of the nodes in the system, and can thus be made simpler and faster. The hardware support for point-to-point in a multi-core embedded system could on the other hand be difficult to support. The
completely shared bus shows a lack of speed-up because all communication must go through one single channel. A shared bus might be simple for a platform to support but it does not scale in a large system.

The second test in [2] was a quad-processor test with a single processor test as reference point. This test included one benchmark using shared bus, one using a crossbar and one using point-to-point communication. The result, in figure 3.10b, shows the performance speedup of the configuration. The completely shared bus shows again a lack of speed-up because all communication must go through the same single channel. According to this test, a clear speed-up is reached by using point-to-point or a crossbar for communication.

3.4.5 Summary

The introduced overhead associated with a task migration and in other mechanisms using inter-core communication is as shown highly dependent on the communications channel. Inter-core communication can be established by different strategies such as: shared memory/bus, crossbar or point-to-point. Depending on the platform and the hardware in use, the strategy of inter-core communication must be chosen and implemented accordingly.
3.5 Task migration mechanism

As a migration is made at a certain place in the program, means that the value of the program counter (PC) must be known at this very place. As concluded earlier in chapter 3, the task migration is initiated by a process monitoring the load statistics. The migration is started after the migration monitor has decided on a migration. This decision will be delivered to the core which has to the chosen task allocated. Two different approaches to signal a task to migrate are discussed in this section. Lastly, incoming migration will be discussed from the target core’s point of view.

3.5.1 Checkpoints

A migration checkpoint is a physical point in the program where a task migration is possible. The point is often provided manually by the programmer and should be put in a adequate location. The location of this point must not violate any logic or misuse any variables if the task is migrated. At this migration point the program will, with some mechanism, check if the currently running task has got a migration request. This request is made by the migration monitor that runs either on the local core or on some distant core, as explained earlier in the chapter.

The migration checkpoints shown in the code listing 3.1 consist of two parts [3],[4]. The first part is the check which, depending on the memory model, checks some kind of table to determine if the task has got a migration request. If so, then the task is migrated to another core by the local migration mechanism; otherwise the task continue running as before. The second part of the migration checkpoint is a label to where the program counter should jump if the task starts on a new core i.e. a resume state so it does not start from the initial state of the task. The following code listing shows the procedure:

```c
int main()
{
    id=restore_context(); //have I been migrated?
    if(id==1) goto resume; //if so resume.
    ....
    while(1)
    {
        if(request_migration){
```
save_context();
execute_migration();
}
resume: heavy_calculation();
encryption();

....

Listing 3.1: Pseudo code for checkpoints

The task is only migrated on a request from either the native core or some other core. The request is made for example by setting a flag in a table that has the task associated with it. This leads to that an overhead is not avoidable because the task must regularly check the value of the flag. A reaction time is neither avoidable because after the migration flag has been set in the table, there is a time $t$ before the task reaches a migration checkpoint. The value of $t$ could increase or decrease depending on how many, and where the programmer sets the migration checkpoints. If the migration checkpoints are set more frequently the reaction time is short but the overhead of checking the table will increase [3]. If setting the migration checkpoints less frequently the overhead will decrease but reaction time will increase.

Tasks are only migrating themselves, from the system’s point of view, because migration checks are done only inside the tasks while they are running. After a task has been migrated, the operating system will switch in another task automatically after a system tick or with a manually made context switch. The responsibility of choosing the checkpoints is hereby put on the programmer. In order to make a task migratable the programmer must carefully follow the paradigms provided by the task migration library.

The Condor project [25] deals with transparent checkpoints, that do not have to be set by the programmer. The programmer inserts checkpoints by including the condor library, and the software sets the checkpoints automatically. The Condor project assumes a UNIX base from which certain system calls are made in order to build the checkpoints. The Condor project does not support migration of tasks that communicate via signals, sockets, pipes or files.

The authors in [3] carried out experiments related to minimizing overhead and maximizing performance. The system was an asymmetric multi-core platform with
a shared memory. The migration mechanism used a *master-slave* approach (see next section) and re-creation way of moving tasks on the target core. The first experiment determined the overhead of the checkpoint mechanism. This check has a certain overhead because the task must either load values from a shared memory or from some other table that contains the information. The experiment shows how much a checkpoint slows down the program as a function of check intervals. Table 3.2a shows that a task will slow down its execution when checks are inserted into the program. If the checks are set more frequently the program slows down in a linear fashion.

<table>
<thead>
<tr>
<th>checkpoint frequency</th>
<th>task slow down</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (1 call every ms)</td>
<td>0.2%</td>
</tr>
<tr>
<td>0.2 (1 call every 5 ms)</td>
<td>0.04%</td>
</tr>
<tr>
<td>0.1 (1 call every 10 ms)</td>
<td>0.02%</td>
</tr>
<tr>
<td>0.05 (1 call every 20 ms)</td>
<td>0.01%</td>
</tr>
</tbody>
</table>

(a) Measurement on slowing down task

<table>
<thead>
<tr>
<th>state size (Kb)</th>
<th>time (ms)</th>
<th>time/size (ms/Kb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0565</td>
<td>0.0565</td>
</tr>
<tr>
<td>8</td>
<td>0.4471</td>
<td>0.0559</td>
</tr>
<tr>
<td>16</td>
<td>0.9231</td>
<td>0.0577</td>
</tr>
<tr>
<td>32</td>
<td>1.8131</td>
<td>0.0567</td>
</tr>
<tr>
<td>64</td>
<td>3.6136</td>
<td>0.0565</td>
</tr>
</tbody>
</table>

(b) Measurement on slowing down task

Table 3.2: Overhead measurements [3]

Still in [3], the overhead of the migration mechanism itself was measured. The overhead, which is listed in table 3.2b, consists of two time intervals. The first called Tshutdown, is the time taken to save the migrated task to the memory. The second time interval called Treboot, measures the time from when the migrated task arrives to the target core until it is fully loaded and running. The sum of these two time intervals was measured. The table 3.2b shows clearly that a larger task will slow down the first part of the procedure significantly more than the second part.

The next experiment in [3] was used to determine how long time is needed for certain tasks to receive a certain amount of CPU time. The experiment used two CPU cores and different sets of loaded tasks. The points in figure 3.11 represent migration checkpoints where the task checks if it needs to migrate. Various configurations of migrations are compared against a local task single task, two local tasks, and three local tasks. The time needed to get a certain amount of CPU time for a migrated task will, as shown in the picture, increase as the task size increases. This happens due to the fact that the time to transfer program code to a target core increases as the size of the program code increases. The graph shows the CPU time versus the time elapsed.
3.5.2 Debug registers

Task migration using debug registers [26] is a poll-free strategy that uses hardware generated interrupts when a task has got a migration request. This approach is only supported if a platform has programmable debug registers. Debug registers are special registers used for example in real-time debugging for setting break points in the code.

Initially when a task starts, all migration points in the task are registered with the OS, i.e. the OS will know where every migration point is. The Instruction Address Compare (IAC) register is then loaded with the defined migration checkpoints for the relevant task. The migration checkpoints now consist of just the label that the program counter (PC) jumps to after a migration. Every time a context switch is made the IAC registers are updated with the relevant information for the task that is to be switched in.

When a resource decides to migrate a task it simply activates the IAC registers, which will now generate an interrupt when the PC reaches the value of IAC. Activating the IAC registers means turning on the compare match interrupts for IAC. The interrupt routine can later call the task migration handler and provide the information regarding the relevant checkpoint to the handler. Migration check by using debug registers could be seen in the following code:

```c
int main()
{
```

Figure 3.11: Comparison of migrations [3]
Task migration with debug registers, according to the code 3.2, uses no overhead checking the migration table like the manual checkpoint approach did. The reaction time $t$ is still present though, because of the time from where the IAC are activated until the PC reaches its value. The Hardware platform must however support these debug registers in order to provide this functionality. The debug registers are often limited to a rather small amount and therefore few migration points are available for the tasks.

### 3.5.3 Incoming migrations

Once a migrating task has been transferred to the target core, the receiver must have a mechanism capable of loading the task into the new CPU core. The two cores must handle both communication and data transfer in order to achieve a successful migration. A problem arises when a busy core is going to handle an incoming task without altering both its own and the transmitter’s performance or response time.

### 3.5.4 Accepting task migrations by polling

Polling is a method in which the target core regularly checks a certain location for information. The location storing the information could be shared or local.
The response time of accepting a task by polling is dependent on the polling interval. Due to this response time, the incoming tasks could be more than one — and should therefore be stored in a buffer. The buffer could be either located in a shared memory or in the local memory. The model shown in figure 3.12 places all the incoming tasks in the task list of the operating system until the buffer is empty. In the most cases there are no tasks in the input buffer, so the mechanism that checks the buffer must be made very light. Minimal overhead should be considered in the aforementioned default case.

After accepting a task, the cores need to exchange communication [27],[18],[4]. The communication is used between CPU cores to ensure a successful migration as seen in figure 3.13. As a task is accepted, the receiver core sends an ‘ok’ signal to flag that a certain task has successfully been migrated to the core. If the migration was not successful due to maximum load or other reasons the mechanism signals ‘not ok’ to flag that the migration could not take place. The migration mechanism, or some other
mechanism, must possibly send an interrupt signal to the other core before migration to ensure that it is woken from an eventual sleep state. A sleeping core might not react on external interrupts or other signals until it has been woken. Eventually there has to be some 'ok I am awake' signal before a migration can take place, which indicates that the core is not in a sleep state. The core can also be configured to wake up from a sleep state regularly and check for migrated tasks. The polling introduces both overhead when checking table and a reaction time from when the task arrives to the buffer until it is polled. The same trade-off is present here as was present in the checkpoint mechanism; namely if the overhead is decreased, the reaction time increase and vice versa.

3.5.5 Accepting task migrations by interrupts

An interrupt means that the CPU stops its current task and jumps to a selected interrupt routine when an event occurs. Interrupts can be used in order to eliminate the reaction time present in a regular polling strategy. The interrupt can be triggered by an external pin, incoming data transfer or some other interrupt based mechanism.

As shown in figure 3.14, the incoming task triggers an interrupt on the target core in order to signal that the core must handle this request. In this case the incoming migrated task is only one to the amount, because each task triggers exactly one interrupt. The task is stored right away in the target task list (or possibly rejected) by the

![Interrupt mechanism for incoming tasks](image)

Figure 3.14: Interrupt mechanism for incoming tasks

handler. Even by using interrupts, the core-to-core communication is important. The communication in this case must ensure that the receiver core is not in a critical state or sleep state; thus not have any interrupts enabled, which would lead to no response from the target core at all.
Locking

Since interrupts could be disabled on the core receiving the task, there must be a way to ensure the migration of the task. The first lock situation occurs when the operating system is in a critical section and the global interrupt flag is cleared. The system will now ignore all external events by completely masking them by hardware. The second locking situation occurs when the external interrupt regarding the data transfer is cleared for some reason e.g. in a sleep state.

The programmer himself could be responsible for unwanted prevention of an incoming migrated task by not setting up the interrupt handler correctly or disabling the interrupts by mistake. Interrupts are also disabled while running a task with hard real-time properties (see chapter 5). In all of these cases there will be no response from the system (compared to the 'not ok' signal when polling). The migration mechanism on the other core must have some kind of timeout clock associated with the migration request. The interrupt timeout will act as a 'not ok' signal when it is triggered, but the whole core could be stalled during this interval.

Communication during migration

During a task migration, the migrating task cannot receive any messages because of the migration process. This poses problems when other tasks expect information from the migrating task, or when the other tasks are about to send messages to the migrating task. A mechanism for halting the incoming messages to the task in question is therefore needed. J.Song & Co. presented in [28] several ways of halting messages and signals to the migrating task. The Following strategies were proposed:

- Tell all peers (tasks) to stop sending and then process all pending messages after the migration.
- Delay and redirect incoming messages from other tasks (as in Charlotte [29])
- Reject incoming messages to the migrating task (as in V [30]).

"We chose the first option to tell other processes not to send any messages to the migrating process and then let the migrating process receive all pending messages before it exits. “[28]
The migrating process collects therefore all pending incoming messages before it is switched out for migration. The user is responsible for handling all the messages in the receive buffer before the task exits.

### 3.5.6 Summary

The mechanism for starting a task migration is usually based on manually inserted checkpoints. These checkpoints are intended to help the migration mechanism to find a place in the code where task migration is possible. The programmer must therefore be trusted to set the checkpoints on adequate places for the migration to be effective. A core must also be told to accept (or reject) an incoming task. A core could either scan a table regularly using polling, or receive an interrupt as a task is migrated.

Both the check for migration request and the acceptance of the incoming task will introduce an overhead, which needs to be taken into account when designing a migration mechanism.
3.6 Task migration initiative

A model for the migration initiative is required in order to define and connect the cores on the platform. The hierarchy distributes different or equal responsibility to the cores. Besides carrying out calculations, defined cores must be responsible for monitoring load balancing, monitoring real-time properties (see chapter 5) and deciding on policies (see chapter 4). The system’s load monitor is, for example, needed to supervise the cores and their activities in the system. This monitor [16] could either be global, or local. A global monitor would supervise all cores, while a local supervises the native CPU core only. The following two models illustrate two different ways of distributing the responsibility among the cores.

3.6.1 Master-Slave

The master-slave principle [16] is based on a centralized monitor which makes the migration decisions for all the other cores. Figure 3.15 shows multiple cores that are connected in such a way that there is a single master core with many slave cores working under it.

![Diagram of Master-Slave model](image)

Figure 3.15: Master-slave model

The master monitor needs information about the current activity of the slaves to
make a decision. This information is, depending on the memory structure, either queried directly from the different slaves, or stored in a memory location by the slaves themselves. These statistics containing CPU load and other possible properties that needs to be stated in order to make a task migration. The statistics are read regularly by the master monitor after which decisions are made. When a migration decision is made, the master inserts the task to be migrated into the migration table. The slaves, on the other hand, notice the migration at the migration checkpoints in their tasks. The migration table, which is read by the slaves, can be reached either with a table check or by handing the information directly to the very cores.

The master-slave model is only suitable when using the checkpoint approach because in this model, the task itself is polling a data structure which is not on the native core. When a migration request is found the slave stops its task and stores the context and migration checkpoint, and sends all relevant data to the new core. This data might also either be sent to a shared memory for temporal storage or directly to the core.

The master can accept or reject migration requests by the slaves. The slaves could be aware of that a native task should be migrated to some target core. This operation could be made by a manual migration by the user or some defined logic in the structure. The slave core can request the migration by sending a migration request to the master core, which could be stored in the same statistics table as the monitoring results. The master, who is reading the table, can make a decision about migration depending on a set of rules and/or a dynamic policy, which states how willing the system is to make migrations.

This model has one CPU monitor per slave core and one decision making mechanism on the master core. The master core could eventually also be running as a slave core if the platform only supports a few cores. In this model all the decisions are made by the master. The slaves could still though provide the master with useful suggestions about migration and other information.

### 3.6.2 Decentralized monitors

The strategy of decentralized monitors considers all cores of equal value. The separate cores have all one decision making mechanism running that decides if this core at the moment is too heavily loaded, and which task it must migrate or which policy must be applied. Each core also runs one monitor. Once the load monitor triggers a migration decision, all other (or a part of) cores are asked to accept the incoming migration.
Depending on the designs regarding the decision making, the target core is chosen. The easiest way is to select the first core for target, but eventually a more complex solution such as a best fit solution could increase the efficiency of the migration. The policy of choosing a core could also be dynamically designed. In contrast to the master-slave model, this model is not aware of the statistics of the other cores before a query has been made. The migration is made to an appropriate core by asking one core at a time for their statistics. The task is, like in the previous model, moved either first to a shared memory or directly to the target core. If statistics is placed in a table in a shared location, all cores must update the table regularly to provide the task migration mechanism with useful information about the load statistics (figure 3.16). Otherwise, moving the migrating task directly to a core could possibly result in a worst/best case outcome since the core holding the migrating task does not know which core has the least load.

There is in this model no need to dedicate a whole core only to migrate tasks — instead all cores take part in the normal calculations. On the other hand all cores must run their own instance of the decision making mechanism and the monitors, which will cause some extra overhead. All cores must also have a migration mechanism each that communicates with the other cores.

Figure 3.16: Decentralized model
Ganthel & Co. show in [4] results from a decentralized model which uses checkpoints, shared memory and a replication mechanism. The CPU cores in this model are of equal value and decide themselves if task migration is to prefer. In this experiment the tasks are initially created on all CPU cores. The task migration mechanism migrates only the relevant states that are needed to resume calculations on the target core. The test shows the overhead when migrating tasks of different sizes. The overhead includes all hereby explained steps that are taken in order to perform a task migration. The results in figure 3.17 shows a rather slow rising slope which means that the task size does not affect the overhead drastically. The complexity of the monitor or the decision making mechanism was not stated in this experiment.

3.6.3 Summary

A task is migrated to another CPU core when all relevant data and program code is transferred to the target CPU core, and the program is able to resume execution from the same place where it was interrupted. Constructing a task migration mechanism depends highly on the chosen model and strategy. The model describes the hardware platform in use and its internal structure which, depending on architecture, could vary. Two different set-ups have hereby been reviewed that show different strengths in different situations. Experiments in articles have also given a picture of the overhead introduced.
4 LOAD BALANCING POLICY

4.1 Introduction

The load balancing policy in this thesis determines how much the system prioritizes energy saving compared to maintaining a high performance. By migrating tasks away from a core, the source core gets lighter load and it will thus spend more time in the idle task. As a consequence, the source core will consume less energy while the target core will consume more energy. The optimal result would be having low energy consumption and a high performance. However, the policies of performance and energy consumption contradict each other due to the fact that a system with high performance will also have high energy consumption because the cores must actively carry out calculations, and cannot be switched off. A system which has placed many cores in the sleep state will, on the contrary, not reach a high performance because the cores are not active.

The migration policy must ensure stability so that the migration mechanism does not make any false migrations (or as few as possible), nor must the migration interfere or modify the running application. The migration policy must be scalable in order to be applied to a large network of cores. Minimal overhead should be used by the run-time migration mechanism, as well as when loading new configuration rules or settings. When a task is selected for migration, the reaction time must be as short as possible for the mechanism to be effective. The reaction time measures from when a migration request is made, until the task is running on the target core. If the reaction time is too long, the need for migration might not exist anymore, or the selected task for migration might not be the most efficient choice. Different systems might value different requirements in the case shutting down or waking up cores[31].
4.2 Measurement techniques

Before using a policy in task migration, measurements must be made on the system. The policy will respond to the measurements — and provide the migration mechanism with suggestions about the current energy situation. The measurements themselves should not introduce a too heavy overhead which could lead to overall weak performance.

The granularity (tasks per node) is important when a migration is made [31], because in a network of cores with high granularity the migration decisions are more likely to be efficient. For example, a core with 10 tasks allocated on it will more likely make an efficient migration decision than a core with only 2 tasks (provided that the CPU load is the same). The core with only two tasks could have to migrate away 50% of its load while 10% could be the optimal. A high granularity gives more possibilities to choose the optimal task/tasks for migration. According to simulations in [31] the improvement of task migration will cease when the granularity reaches a certain lower threshold. In order to determine when and where to migrate a task a measurement system is required. The system should regularly measure the core load for all cores connected to the system, and provide the task migration mechanism with the statistics. Two different methods will be reviewed in this chapter.

4.2.1 Temperature measurement

As presented in [32], [33], [34], [1] and shown in figure 4.1, one way of achieving load balancing is to measure the heat generated in every single core. This is done by an integrated thermometer inside the core itself.

The measurement data is then stored in order to provide heat statistics. A hot core results in a heavy loaded core, and a cool core will be nearly idle. The primary goal in these articles is, besides migration policies, to balance the heat distribution in order to provide enhanced stability, reliability and minimize failures due to the heat itself. The heat dissipation in a core is not instantaneously high when the load

![Figure 4.1: Thermal curve in component](image-url)
gets high [35] because of internal heat resistance in the components. Likewise when the load goes down the heat will remain high for a certain time until the core has cooled down. This phenomenon could be seen as a kind of integral representation of the load in a CPU core over time. The present behavior eliminates the migration mechanism from making preconceived decisions when, for example, a load peak occurs. The rule of the policy [36] is basically to even out the so called hot tasks and cool tasks over all the cores to create a uniform heat distribution. The energy policy for task migration can strive to maintain a certain amount of cool or hot cores in the system.

### 4.2.2 Discrete control blocks

Instead of approximating the CPU load by measuring the temperature, the statistics about the load can be directly derived from the operating system’s scheduler. FreeRTOS provides a call that returns a table of the current processes running including their CPU time, which can be used to calculate the load on the native CPU core. This statistics is stored in a possibly global place, or on the native core. The statistics is loaded into a control block system as seen in figure 4.2. The system manipulates the results by using mathematical calculations on the data. The outcome of this system is a factor that will be used in order to decide on a migration. A task should be migrated only when there is an improvement in doing so.

![Figure 4.2: Model to calculate migration points](image)

The simplest model will only perform task migration depending on the current statistics, this kind of system is called open-loop. A more advanced model would also

48
remark previous migration by providing feedback to the migration mechanism, which could reduce the ping-pong effect of task migration. A task might be migrating frequently from core to core because of an open-loop migration system. This effect takes action when the task in question always seems to be the perfect candidate for migration. A system using feedback can prevent this from happening, by using background knowledge which is received from previous migrations. The information is used to determine the outcome before a migration is made. Furthermore, research for adaptive scheduling and resource allocation has been conducted regarding generic algorithms, neural networks and artificial intelligence [37]. Intelligence is a base for an effective migration policy as long as the intelligence not is too heavy for the system in question.

4.3 Migration policies

The migration policy is a set of rules which will help the migration mechanism to make a decision on migration. The rules set weights on which outcome to be preferred from the system. Two main aspects in this thesis are performance and energy consumption, which have individual rules that differ greatly when using a task migration.

Since maximal performance and minimal energy consumption cannot be accomplished at the same time the system needs knowledge about what is more important at the time [38]. If the policy is set to maximize the performance, the migration mechanism will make the migration decisions based on what will minimize the overall calculation time. If the policy is set to prioritize energy consumption, the policy will allow increasing calculation time but decreasing energy consumption. The two policies could possibly be divided into several steps that overlap each other, as well as dynamically change during run-time.

4.3.1 Performance

A high performance is reached when all CPU cores run tasks completely in parallel without having any cores fully loaded. The strength of super computers and graphics accelerators is to have many calculation units and running the jobs in parallel. The policy of high performance will enable all CPU cores at all time. The migrations are made when a certain core reaches a defined threshold where the load monitor considers it profitable to run a task migration. A task is migrated when a CPU core shows too
high load statistics, while other cores have the capacity for more tasks. In a performance policy, the task is migrated to a core which has the lowest (or a lower) load than the source core.

A simulated model was created that uses a PID-controlled-like system which creates migration scores based on the previous behavior and the predicted trend of the system. The randomly generated CPU load curve shown in figure 4.3 is given scores that could help making decisions on task migration. The scores (as a function of time) show when a migration of tasks would prove to be beneficial. A high score means that a task should be migrated, while a low score means that the system should not take any action. The block diagrams for the simulations can be found in appendix A. The result of the simulation is shown in figure 4.4. The model calculates the integral over a certain time window, which size is dynamic and will be reset once the CPU load fall below a defined cut limit. The load trend is predicted by calculating the derivative of the load curve with respect to the time. Because of the integral and derivative part considered, it will be easier to eliminate false migrations. The D-value and the I-value can be tuned in order to set different weights on the history and the trend, as well as

Figure 4.3: Load generator

Figure 4.4: Measurement of migration scores
the positive and the negative slope of the load curve.

The highest scores are received from situations where the overall history has a high load while the load trend is going upwards. Low scores are received from load peaks and low overall load. High performance is reached, in summary, when tasks run in parallel with no CPU cores overloaded.

### 4.3.2 Energy consumption

The policy of energy consumption differs from the maximum performance policy. To minimize energy consumption, one CPU core could be overloaded to a certain point while another is idle. Overloading a core and reducing the parallelism of the system results in lower performance, but a low loaded core or an idle core will consume less energy. The least energy is consumed when a core is completely shut off [5] — which can be done if a core is 100% idle.

The energy consumption policy aims towards putting as much load as possible on a certain amount of cores and shut the rest of the cores down. This policy will allow a single core to be loaded until a threshold for deadline misses is reached or until a load threshold is reached. This model differs from the performance model also in sense of making local decisions. The model strives to migrate tasks from the core that has the least load in order to be effective, but a single core does not have the information about how much load the other cores have. Because of this, the single core cannot by itself make a decision on migration; instead all cores have to be checked to find the minimal load value before a decision can be made. As additional cores are added to the system there will be an increase of checks, both in order to find a source core and to find a target core.

A simulation was inducted to illustrate the use of a global decision making mechanism. Figure 4.5 shows two CPU loads that each will get migration scores according to the current situation. The upper graph shows a randomly generated load curve, and the lower graph shows a periodic load curve. For simplicity both CPU cores only run one task each. The CPU core with the periodic task seems every period to be the better candidate for task migration, because the load goes momentarily down to 0%. As soon as the waiting period is over, the load will de facto go up to 100%, and declare this task as a bad candidate for migration. This phenomenon could be beforehand noticed, and the task selected as static (not migratable). Another solution would be to use another
sample period for measuring the load, and thus get an average value for the load. The result of the random task in figure 4.5 sets high scores when the task is predicted to have a low load. A CPU core with such a total load could benefit from migrating the task to another core and shut itself off. The final result is presented in the figure 4.6. The upper part of figure 4.6 shows scores (1 or 0) where the system would benefit from migrating the task away from CPU1, while the lower part shows the corresponding values for CPU2. The scores are, like predicted, high on the periodic task every period. The useful result is however shown in the upper part of figure 4.6. The scores are set

Figure 4.5: Load generators

Figure 4.6: Result scores
where the load is low and where the trend is going downward. In a real-time system the rules for this policy must observe the real-time properties stated. The deadline misses for a soft real-time system could be measured, and used as guiding point for the task migration. This matter is discussed in chapter 5.

As concluded, a policy for minimizing energy consumption and maximizing performance must regard a trade-off between energy and execution time. The stated policy for the system gives an insight into which matter is of more importance for the system, and how tasks should be migrated in order to reach the stated goal.

4.4 Implementation of monitor

In this section an implementation of a load monitor is discussed. In order to gather load statistics from the system, there must be a way of monitoring the different parts of the system. The monitor is responsible for collecting all the information from the cores into the central collection of statistics that controls the behavior of the rest of the system. One part of the monitor is collecting information about the events in the operating system, and another part is creating statistics of this provided information. In [39] the authors describe this monitoring system with three different parts as follows:

The so called event tap is part of the operating system, which records important events in the operating system. Event triggers on task level such as creation, wake-up, suspension, resuming, removal etc. signal all the events of the tasks. When the event tap records an event, the information is stored in a local temporary buffer.

After a number of events have been stored in the buffer the reporter-mechanism sends the information over a communications channel. The reporter is run periodically and empties the whole information buffer each period. The mechanism for sending the information is entirely depending on the communications channel in use.

The data is sent from the reporter to the Visualizer, which interprets the data and provides the user with useful statistics over the system. The visualizer could either be run on the native core or on some other core; the only requirement is that there is a feasible communications channel between the two nodes. When using the Master-slave model presented in the previous chapters the visualizer would be running on the master core collecting statistics from all the CPU cores. The monitor will collect in-
formation regarding deadlines, priorities and CPU load etc. Based on both policies and statistics the monitor provides suggestions to the task migration mechanism.

The task migration mechanism is provided with information provided by all instances of the operating system. Based on the policy and the information, a task migration is eventually made.

### 4.4.1 Monitor latency

The reaction from the monitor is not instant, which means that there is a probability that the migration decision was not correctly calculated. The latency of the monitor is measured as the time it takes to update a state after an event. The reporter is sending the data in the event buffer to the visualizer with a certain time period, which length will introduce a lag time during which the system will not yet notice the event. Additional lag time also adds up during the calculation of which state should be next in turn. For example if the cable in an elevator breaks the system might not allocate all the resources to the emergency system instantaneously. After the system notices the event the full power of the system will be used and tasks will be distributed to reach maximum calculation capacity.

### 4.5 Summary

An energy policy is needed because system that prioritizes both performance and energy consumption is not feasible. The strategy of increasing performance is to distribute tasks evenly on all CPU cores, whilst the strategy of minimizing energy consumption strives to put as much work on a single core as possible, and shut the rest down. A policy can be made to gradually shift from energy consumption to performance, with the assumption that the policy can be dynamically altered. The amount of steps between performance and energy consumption could in principle be arbitrary, but in practice they should not be too many because of the overhead of changing the policy.

The migration decisions are made by monitoring the CPU cores for load statistics. When monitoring a CPU core a more effective result will be reached if the monitor uses a more complex way of measuring than plainly checking the current CPU load. A more realistic conclusion about the system can be drawn if the monitor checks both the history and the trend of the load curve.
5 Mixed-critical systems

5.1 Overview of a mixed-critical system

A mixed-critical system is a configuration where CPU cores of different criticality co-exist. This thesis presents two different types of CPU cores — safety-critical cores and non-safety-critical cores. Tasks on a safety-critical core are defined as hard real-time tasks (HRT). These types of tasks have a strict deadline which is not allowed to be missed. Deadline misses on a safety-critical core could have catastrophic consequences that could lead to massive profit loss or even injuries. Non-safety-critical cores run more flexible tasks that are allowed to miss a few deadlines as long as the system performs in an acceptable manner. The CPU core units run in parallel and are able to interact with each other. The different levels of criticality allow different tolerance of failure, but complicate the certification of the system.

The different types of cores are allowed to exchange information with each other, which in a multi-core platform could take place through a virtualization layer which lies beneath the operating system. Migration and communication to-and-from a safety-critical core cannot alter the calculation time of a HRT or delay the release time in such a way that a deadline miss could occur. These precautions are planned beforehand in a hard real-time system. Furthermore, a HRT must not be dependent on results from a non-HRT since the deadline of this task is not guaranteed. The HRT can, if this is the case, ignore the needed information or else the system must be restructured so that the information is provided from another HRT.

In a real-time system certain created tasks may have real-time requirements, which means that missing deadlines is not acceptable, or not desired. When shutting down cores the whole system will have a lower performance and thus eventually not meet all its deadlines. In [40] the two distinct definitions of cores are proposed as: real-time cores (RC) and non-real-time cores (NRC). The difference is that a RC will guarantee real-time properties for its tasks while the NRC core will run primarily the preemptive
tasks with lower priority and less strict deadlines or no deadlines at all.

5.2 Task migration in a real-time system

Task migration in a real-time system is possible, but in contrast to having no real-time tasks, deadlines must be taken into account. When planning task migration in a real-time system, one must observe the different types of tasks and the different types of cores. As stated, the two different cores — RC and NRC — run different tasks depending on their level of criticality. Four different combinations of task migration are presented [41].

NRC-to-NRC  Load balancing by migrating tasks between two non-real-time cores poses no problems related to keeping real-time deadlines. The policy regarding this kind of migration could simply be done with the default set of rules stated in the energy policy (chapter 4), since a task with no real-time requirements have no deadline to keep.

RC-to-NRC  A non-real-time task could be scheduled on a real-time core if the real-time core is not fully loaded with the real-time tasks. The real-time task/tasks have the highest priority on the core and will not be preempted by a non-real-time task. As a non-real-time task tries to disable its own preemption, the system must move this task to a NRC in order to maintain the real-time properties of the RC. The preemption could be disabled if the task goes into a critical section or a similar atomic operation. This is extremely important if HRT:s are involved. In case of emergency, even HRT:s on real-time cores could benefit from migrating to a NRC if no other solution is available.

NRC-to-RC  If a RC has free CPU time, a non-real-time task on a NRC could migrate to the RC for energy saving purposes. As the migration costs time it is important not to move tasks back and forth on a RC frequently. A RC can only accept incoming migrations if the QoS is kept on a sufficient level, and if the real-time properties are kept for hard real-time tasks. The QoS property will be discussed in the section 5.2.1.

RC-to-RC  A RC guarantees a hard real-time task (RT0) to keep its deadline, which means that the scheduler will give the RT0 task CPU time as soon as it requires it.
Allocation of multiple hard real-time tasks on a CPU is possible, but it is up to the developer to ensure the correct scheduling [40]. Besides from hard real-time tasks, there could be many soft real-time tasks allocated on a core, but the lower prioritized tasks will always be running in the preemptive state. These tasks can use the CPU when the RT0 task (or any other hard real-time task) isn’t running. If the soft real-time tasks are about to become non-preemptive for some reason, the tasks must be migrated to another core in order for the RC to ensure its hard real-time properties. If the migrated real-time tasks are having soft real-time requirements a deadline miss is not fatal for the system.

A heavy loaded RC might be declared conservative. This would mean that the core has a flag set that prevents other cores from migrating tasks onto it. Cores of this kind can eventually remove the migration and the monitor mechanism altogether because migrations, from-and-to, is not going to happen.

5.2.1 Migration of soft real-time tasks

In this section, the migration of soft real-time tasks is discussed. By migrating all tasks from one core in a mixed-critical system, the core can be shut down and thus keep the energy consumption minimized. The soft real-time tasks in such a system are the first candidates for the migration. Soft real-time tasks have soft real-time deadlines that have to be kept to a certain degree to maintain the required quality of the system. If the required quality is momentarily decreased, the system allows more deadline misses and therefore grants more freedom to reduce energy consumption.

The primary reason for migrating tasks is, using this policy, to save energy. When a core is completely shut off it consumes minimal energy, but have a long wake-up time. A non-real-time system could simply monitor the CPU loads and their load trends in order to shut down or wake up various cores in the system as shown in chapter 4, but in a real-time system the tasks cannot be arbitrary scheduled on any CPU core at any time. This fact exists because certain tasks must keep their deadlines independent of the CPU load. The deadline could be missed for example by a wake-up overhead, and the whole system must therefore have a certain degree of intelligence and a set of rules used to predict if a particular operation can be performed or not.

In [42] a suggestion of making decisions by determining the Quality of System (QoS) is stated. The QoS is a measurement of the expected performance of the system at a given time. The selected performance could vary in different cases; for example
latency, bandwidth or response time can be a measurement of QoS. In order to have a migration mechanism for real-time systems, there must be a way of monitoring the QoS. The migration mechanism will make the migration decisions based on the information from the monitor, which is basically a way of making a dynamic energy policy for the migration mechanism according to the system’s current need. According to the work presented in [42] the default power states could be described with a so called power state machine as shown in figure 5.1.

The machine displays the different power states a task could be in. This standard PSM shows only three states: run, idle, sleep. These three states represent three different levels of energy consumption. The sleep state could also consist of many levels of sleep states. If a core goes into a sleep state there will be minimal energy consumption but a wake-up time due to latency is introduced into the system. This latency could cause a real-time system to miss its deadlines. In case of many sleep states, the deeper the sleep state is, the longer the wake-up time is and the longer the latency is. The PSM can be used to measure the current state of the whole system i.e. how many cores are running and how many needs to be running. This model is considered ineffective when implemented on a real-time system, since the time lines containing the deadlines could be disturbed when not taking into account the real-time properties.

When choosing if a device should be turned on or off, the importance of the currently executing jobs must be determined. The authors in [42] suggest a way of determining the QoS in the system. In this case the importance of response time and calculation time is measured as Quality of System.

A new model for determining the performance need was suggested. The new model called Extended Power State Machine is illustrated in figure 5.2 and consists of several QoS levels which are represented in the state machine with the state $Q_n$. The state $Q$ is a way of describing the importance of quality in the system at the current time. A state $Q_n$ changes to a state $Q_{m}$ when listed events $E$ takes place. The state changes
is described as transitions $\delta$ which makes up a set of events $E$ and states $Q$ that must coincide to trigger a state change. The events $E$ can be triggered by events like battery level, external inputs or some internal logic. Transitions are logical rules that control the sequences.

![Extended power state machine](image)

Figure 5.2: Extended power state machine

Examples of transitions can be defined as:

$$
\delta_1 = Q_i \times (E_1 \& \& E_2) \rightarrow Q_j \\
\delta_2 = Q_k \times (E_3 \| E_4) \rightarrow Q_l
$$

Where the first transition $\delta_1$, which sets the system in state $Q_j$ takes place when events $E_1$ and $E_2$ are both happening while the system is in state $Q_i$. These different states show that the policy could be dynamically changed to determine the use of more or less cores in the system. Figure 5.2 shows an EPSM with four different states. The states $Q_1$, $Q_2$, $Q_3$ and $Q_4$ can be entered and exited depending on which transition $\delta$ takes place. The value of $Q_n$ does not necessary mean that $n$ cores are activated, but that a policy of grade $n$ is activated, and will more likely make use of more cores than the state $Q_{n-1}$.

A monitor is running as a process on each CPU core, and will provide the migration mechanism with information to make the right migration decisions. The monitor does not take into account the trends in CPU load but only looks at the real-time properties.
of the overall system. The real-time property in a real-time system is very important and could determine if a system is applicable or not. The monitor gives therefore the migration mechanism a lower budget on how much of the system can be shut off and still meet the real-time requirements.

**Monitor interference in a real-time system**

Monitoring real-time processes *per se* interferes with the system. The monitor, which sends messages periodically, will consume a certain time slice of CPU time. This time slice cannot interfere with the real-time deadlines, especially not in a hard real-time system. The interference of the monitor must be predicted in the schedulability analysis [39] when constructing a system. The analysis is done by investigating periodic and aperiodic tasks that are about to be allocated in the system provided by a given scheduling policy. The measurement must also foresee the worst case scenario.

*Acquaviva & Co.* tested in [5] ways of minimizing power consumption while maximizing performance on a system producing a certain amount of video frames per second using one, two or three cores to carry out the calculations. The first experiment was to determine the overhead of migrating a task by using the replication method.

The system was of master-slave type which means that one core runs a master demon which decides what tasks on which core to migrate. A slave demon runs on all the other cores and provides statistics about their CPU load to the master demon. The statistics provided by the slaves was transferred to a shared memory region which was accessible by the master.

This experiment was done to show the monitor overhead when using the stated configuration. The overhead presented consist of several parts, pictured in figure 5.3. The first part was a measurement of the CPU utilization the master demon requires to check the statistics provided by the slaves. The red curve in figure 5.3 shows the over-

![Figure 5.3: Migration overhead [5]](image-url)
head a decision making mechanism used. The blue curve represents only the statistics check. The update frequency shows how regularly the system reads from, or writes to the statistics table.

This experiment shows that the overhead increases in a multi-core system with task migration, especially when a decision making mechanism is used. The decision making mechanism is used to determine the target core, to which a task is been migrated. The amount of tasks seems not to affect the outcome highly. The complexity selection mechanism and the complexity of the decision making mechanism was not stated in this experiment, nor was overhead due to traversing the statistics table.

The second measurement in [5] determines the slave demon overhead. The slave writes the CPU load statistics to a shared memory location, but does not take any part in the decision making when migrating a task. The operating system that was used (uCLinux) allocates at least 64KB of user space memory even if the task is smaller. This shows that very small tasks might not be efficient to migrate.

![Graphs](image)

(a) Migration overhead  
(b) Common input core

Figure 5.4: Overhead measurements [5]

The test is hereby showing in figure 5.4a that the overhead of writing to a shared memory will increase both by increasing the amount of tasks and the check frequency. By comparing the outcome with the previous test one can state that writing to shared memory is noticeable slower than reading from the memory. The CPU utilization required to write 16 tasks at 100 Hz is nearly identical to reading 16 tasks, including the execution of the decision making mechanism.

The total cost for one migration can then be calculated by counting the required amount of CPU cycles the migration mechanism in total needs. In figure 5.4b the
result (from [5]) shows the CPU cycles as a function of task size. The amount of cycles the CPU uses from the migration checkpoint in the program, to the complete resumed task on the target core is pictured in figure 5.4b.

5.2.2 Migration of hard real-time tasks

Migration of hard real-time tasks (HRT) differs greatly from soft real-time tasks. Firstly: in order for a real-time system to be theoretically schedulable the condition:

\[
\sum_{i=1}^{n} \frac{e_i}{p_i} \leq 1
\]  

(5.1)

where \( e \) is the execution time, \( n \) is the number of tasks and \( p \) is the period, must be valid for all cores on the platform that runs HRT: s. This states that the jobs running on a CPU core cannot be more than the capacity of the processing element. Secondly: the HRT are not allowed to miss any deadlines. If a hard real-time task misses a deadline the cost for a company could be enormous or people risk getting injured. A deadline miss on a hard real-time system such as an airplane or an elevator could be fatal. The response time of a HRT could however vary as long as it not affects the deadline [43]. This fact could be exploited during eventual migrations of hard real-time tasks.

The reason for migrating a HRT might involve energy consumption or performance, as stated in previous sections. If the best-case and worst-case execution for a HRT differs greatly, the utilization in the best-case might leave room for other jobs. If the worst-case execution time occurs while the core is utilized with other tasks, they are simply delayed until the HRT has completed its job. The system also might improve, in terms of energy reduction, of merging two HRT: s into one core if both their deadlines will be met.

A HRT is running on a core that is called a safety-critical core. This core is declared more conservative and paranoid than a normal core in order to ensure the sufficient time slices to its HRT: s. In terms of virtualization, this type of core may not run a virtualization layer such as a hypervisor or a more restrictive type of virtualization layer. Also communication should be used in a very restrictive way. Task migration or other interaction between other cores must still ensure that the deadlines are met at any cost.

A migration, from-or-to, a RC is only feasible if the whole system remains schedulable
after the introduction of the migration overhead i.e. if no hard-deadline is missed after the migration has been accomplished. The deadline of the migrating HRT, as well as the eventual HRT deadlines on the target core must also not be missed as a consequence of the migration. In order to make this migration possible, the system must use a migration mechanism that has a completely predictable overhead. The worst case scenario of this overhead must be considered to ensure the whole migration delay. This is a problem for example when using Network-on-Chip configurations, since the time of the data transfer depends on the current load of the network. The target core for the HRT must not have any locking mechanism [41] or other exceptions present from the point when the HRT is transferred to the other core, until the migration is completed.

There are two possible occasions to perform a migration according to [21]. The first possibility is when a job is released, and the second is when a job has completed its execution. The tasks are therefore only migrated before they have carried out any calculations or after they have performed all the calculations. It is up to the programmer to insert checkpoints in the program at the any of these locations. The checkpoint must be carefully added either in the very beginning of the task, or right before the task goes into the waiting state. When using this checkpoint paradigm there will be no extra overhead inside the actual task which leads to a more predictable execution time. At a checkpoint, the offline statistics about worst case scenario overhead is taken into account to determine if a migration is theoretically feasible.

### 5.2.3 Task migration hazards in a hard real-time system

All kind of interaction involving a safety-critical core must be handled with caution. Interaction means that the core is requested to provide the caller with some kind of information. If a non-safety-critical core requests information from a safety-critical core the danger is low because the non-safety-critical core can, with non-strict deadlines, wait until the information is provided. The non-safety-critical core could even calculate the exact overhead if the information is given by a HRT. A hard real-time task’s dependencies should be kept minimal, because if data that a HRT is dependent on is delayed, the real-time requirements could be violated. The dependencies of a HRT must be guaranteed to be delivered on time.

Various overheads related to task migration and inter-task communication must be measured if a safety-critical core should consider task migration. Transfer size when moving the task itself and the time it takes to send the data is difficult to estimate
exactly. For example a dynamically allocated array in the heap could have different size at different times and hence the migration time could vary. The following list presents the different overheads and situations that could interfere with migration of real-time tasks.

**Communication overhead** The communication cannot instantaneously be guaranteed if another communication is taking place over the same line. Depending on which type of communication is used, different unpredictable overheads and waiting states are introduced.

**Critical sections** An operating system is on special occasions inside a so-called critical sections. In these sections the operating system is not interrupted and hence an instantaneous response is not guaranteed. The critical sections are often short, but if a hard real-time system should be certified there must not be any risk of missing a deadline.

**Statistics** The overhead inserted in the calculations is based on statistics i.e. previous measurements on the system’s behavior. These statistics might not withhold the absolute information to the monitor when making the feasibility test.

**Dependencies** A HRT might be dependent on information of another HRT on the same core. This is not a problem as long as these tasks are running on the same core, but if one of the tasks is migrated to another core, the response time for information exchange might not fulfill the requirements stated. A HRT with strong dependencies may need to be declared conservative.

**Checkpoints** The programmer must manually ensure that a migration checkpoint is placed on a suitable place. The checkpoint overhead must be calculated exactly because the task will run the checkpoint mechanism regularly, which leads to that the execution time for a job will be slightly longer. Furthermore a miss placed checkpoint could endanger the migration time as well as the check time.

**Sleep states** A CPU core must not be in a sleep state as the overhead for a HRT is calculated. The wake-up time differs depending on architectures and the level of the
sleep state used. Sleeping cores can also simply be ignored when a HRT migration is executed.

5.2.4 Weighted policies

In order to help the migration mechanism making correct decisions and eliminate eventual unnecessary calculations, weights could be placed on certain tasks and cores. When saving energy by migrating tasks, all tasks from a core will be migrated. These selected tasks are chosen due to their low-load trend over a time as show in the previous section. This trend could however be proven false and the load could rise drastically after the migrations. This leads to that the monitor monitoring the CPU load will receive a load peak and the migration mechanism improving performance must kick-in to solve the problem. The following suggested points could be dealt to different tasks in order to help calculate if a migration is appropriate:

Trust points

Every state $Q$ has a trust point level which tells how many incoming tasks a core have the courage to accept in a time period. As the system is running in a high $Q$ state, the overall paranoia for lack of performance is high and thus the trust points are getting very low. The cores will, in this state, accept only a few migration requests if many are given. If the system runs in a low idle $Q$ state the need for energy saving is of importance and the running cores will accept many incoming tasks but at the same time risk having more soft deadline misses. The trust points will only affect many-to-few migrations i.e. many running cores that want to migrate tasks into a group of few running cores. The other way around i.e. few-to-many migration is not affected because migration in this direction will decrease deadline misses. The more a core is loaded, the more likely it is to save energy, but a dynamic load limit can state when a CPU core does not accept any more migrations. A paranoid state having few trust points will not migrate tasks to a highly loaded core whilst a trustworthy state, with many trust points, could migrate tasks to a core that even is fully loaded.

Real-time points

Real-time points are introduced to aim migration in first hand to cores without real-time tasks running on them. Cores with many real-time tasks will register few real-time
points to the monitor and are therefore not likely to be a target of migration. Different priorities could also be registered with the real-time tasks which increases or decreases the amount of real-time points. These points could also change dynamically in time. A core without real-time points is called conservative and will never accept incoming migrations. This type of core could be of use when defining a safety-critical core with many hard real-time tasks.

**Mobility points**

If information about a tasks behavior is provided, the task could be loaded with a certain amount of mobility points, which show the suitability of migrating a task. A task is suitable for migration if the code structure fits the checkpoint approach well, if the task does not communicate with other tasks or the outer world, if the task does not require much CPU time, or if the task has as very predictable load pattern. These kinds of tasks could with higher probability and low risk be migrated to other cores without altering the real-time properties.

### 5.2.5 Emergency task migration

A fault in the service could force a migration of one or many tasks without any policy decision. Emergency migration [44] can be used both for soft and hard real-time tasks, which means that both the safety-critical and the non-safety-critical part of the system need to have an emergency plan. Migration of a HRT could be a vital emergency strategy if a hardware error or communication error disables parts of the system. The emergency migration mechanism moves the HRT or non-HRT to another core for a safe shutdown of the system and possible error log generation. Performance reduction is expected as a result of emergency migration, as the system must allocate more tasks into fewer CPU cores without making compromises. The QoS will eventually drastically decrease, but as a last-resort this might be applicable. The emergency task migration is only conceivable as long as the data that need to be migrated and the migration mechanisms themselves not are physically corrupted.

As a task migration is dependent on pre-saved data, there is need for checkpoints in order to perform a migration. The checkpoints are used to save the state of the task regularly so it later can be migrated during an emergency. When the task is migrated, the program counter starts to run from the last saved state at a checkpoint in the task.
This means that the task will not continue exactly from where it was interrupted, and thus eventually not produce relevant data. Unfortunately there is no information about when an emergency migration is going to happen, so saving the task state must be done arbitrary. The checkpoints also introduce overhead to the task, so the developer needs to decide on how important a task is and which tasks need to be saved.

The chosen target of an emergency migration need to be quickly determined when a HRT is about to be migrated. The point of emergency task migration fails if the HRT reaches its deadline before the task has been migrated and completed its calculations. The HRT must be migrated to a CPU core which is idle enough, and has a sufficiently low utilization to support the incoming task. If these criteria are not met, the system must decrease the QoS value in order to ensure that the HRT will meet its deadline. If the system does not manage to support the incoming task even though the QoS is set to zero, the HRT must be migrated to another core. The target core for a HRT must, in case of no suitable SC be a NSC. The achieved configuration which has a HRT running on a NSC is the least to prefer, because a NSC cannot ensure the same safety and hard real-time properties as a SC. In this exception case the system need to safely shut down and await recovery. Migration of a HRT to a NSC might however be the ultimate decision in certain cases; since the emergency migration is not feasible if a currently running HRT must give its time slice to the incoming task and decisively miss its own deadline.

5.2.6 Elevator example

This section gives an example of a system using an ESPM to maintain a sufficient QoS of the system; at the same time as energy saving is of importance. Consider an extensive elevator which has a computer system consisting of multiple CPU cores. This elevator can provide the user with a special touch keypad for extra functionality, music playback and playback of HD-video for commercial reasons. This elevator has a mixed-critical configuration with soft real-time tasks for music and video etc. and hard real-time tasks for the breaks and the motor. A simplified version of the elevator’s EPSM is shown in figure 5.5. In the normal case the elevator has only the normal keypad activated, and the video playback is disabled during night time. In this idle case the computer system can shut down most of its parts in order to save energy. The QoS of the system is now allowed to be low, which strives to a policy with low energy consumption (Q1). Even though the system is in the low power state, the HRT: s are
running with the highest priority on the safety-critical part of the system. The safety-critical core will not be put into a sleep mode even if the elevator not is currently in use, because otherwise the hard real-time properties for the breaks and motor etc. would not be preserved. The safety-critical part of the system simply ignores the power scheme model.

Figure 5.5: Model of the states in an elevator

When a user activates the special keypad at the same time as music is playing inside the elevator the system notices the increasing load and eventually soft deadline misses (lower QoS). The system must then activate more CPU cores to prevent the user from noticing jerky music and slow touch response. The system goes into \( Q_2 \), which has a little more aggressive policy than \( Q_1 \) and is likely to involve more calculation units than \( Q_1 \). The system goes back to \( Q_1 \) when the normal keypad is used or if the music stops. The \( Q_3 \) state is having the same behavior as \( Q_2 \), only that this state is activated when more calculation power is required for example in case of a video playback. The \( Q_4 \) state is an emergency state in which no energy saving is prioritized, but only a as fast as possible response time. In this state the system activates all the calculation units and will only leave this state upon a manual reset. The emergency state will not activate any more safety-critical cores since all of these always are activated independent of which state that was active before the emergency. The emergency system uses the non-
safety-critical cores for additional emergency equipment such as emergency videos or VoIP services, which do not have the need for hard real-time properties because of the human inability to notice minor soft deadline misses.

A fault in the system is treated, according to the policy, in the same way as Q4. The system notices a faulty component, either in the safety-critical part or in the non-safety-critical part. The system need to move all currently executing tasks from the faulty part of the system to another working part. The elevator has only an emergency plan for the safety-critical part which handles the breaks. If the core running this software notices a fault, the task is immediately migrated to another core to guarantee the safety of the people inside the elevator. The video and audio has no emergency checkpoint which leads to that if the NSC running the audio fails, the service will simply freeze. This result might not have catastrophic consequences and therefore the increased QoS and the lower energy consumption has been prioritized.

5.3 Simulations

Simple behaviors when migrating tasks can be simulated by constructing a simulation platform which is able to create real-time tasks, and migrate them to a selected CPU core accordingly. A Matlab Simulink scheme for migration simulation was created, which simulates migration from-and-to both real-time and non-real-time cores. The simulation environment creates periodic tasks on the selected CPU cores. The period, as well as execution time, deadline and phase is selected by the user. When saving energy, the less loaded CPU core tries to migrate over its task to the other CPU core without missing any deadline, or by trying to keep as high QoS as possible. The block diagrams for the simulations can be found in appendix A.

5.3.1 Safety-critical tasks on RT cores

Even though certification of safety-critical migration might not be to consider, the simulation of such tasks could easily be studied. A safety-critical task or HRT must not miss any deadline or make any other HRT lose its deadline as a result of the migration. Besides making the real-time scheduling scheme, the extra overhead must be added to the execution time and always be taken into consideration. The overhead in a task migration between CPU cores might be difficult to estimate or calculate because of the complexity of communication and the moving of data between memory locations.
The Simulink model supports overhead insertion when making a schedulability test. The overhead is inserted by the user who has value of the time slice a migration between cores consumes. If the time estimate not is accurate, the user can easily add an extra safety overhead to make sure that the schedulability test is feasible.

Figure 5.6 shows tasks that are partly overlapping each other. The deadline — which is equal to the period — allows a certain re-organization in the schedule. In this case the migration was successful, even though the introduced overhead of migrating the tasks.

Figure 5.7: Two hard real-time tasks running on one CPU core

Figure 5.6 shows two CPU cores with one HRT allocated on each. The less loaded CPU is trying to migrate its task to the other core in order to be self shut down. The result of merging the two tasks is shown in figure 5.7.
The simulation algorithm starts by checking the overlap by comparing the time axis on the both CPU cores. If no tasks are overlapping each other (purple bar), they can be migrated to one core with or without considering the overhead. If the task schedules overlap each other (yellow bar), one task has to be delayed a certain time in order to fit into the real-time scheme. The delay cannot exceed a selected deadline; otherwise the scheme is not feasible. After the eventual delay has been introduced, there is one more check for overlapping. If no tasks are overlapping and the deadlines are kept, the task migration was successful; otherwise the tasks simply cannot merge into one core. Finally the simulation software prints out the result.

5.3.2 Non-safety-critical tasks on RT cores

In the same way, non-safety-critical tasks with soft real-time requirements can be simulated. The SRT on one core tries to migrate onto the other core without lowering the QoS of the system. In the simulation, the periodic SRT: s on both cores can be run in both preemptive and non-preemptive state.

![Figure 5.8: Load generators of soft real-time tasks](image)

The simulation is carried out in the same way as the HRT migration with the exception that a task is allowed to miss a deadline. The deadline, period, execution time and phase are similarly selected by the user as in the HRT case. If a task misses a deadline the QoS will drop. Depending on a selected factor the QoS is more or less dependent on the currently running task.

Figure 5.8 shows two CPU cores with one task each. The first core is trying to
migrate its SRT to the second core. Problem arises when the period and deadline of the first task reaches the next period. The task cannot be delayed any longer and must steal CPU time from the task on the second CPU, which has a set deadline equal to the execution time. The second task will miss its deadline slightly and as a result the QoS will drop. The result in figure 5.9 shows some jobs that directly fits into the scheme (blue bar), but some will be forced to delay their execution (purple bar) and miss the deadline. The result in the lowest part of figure 5.9 shows that the QoS value drops every time a deadline is missed.

5.4 Summary

Besides having a policy just for maintaining the appropriate energy consumption, the real-time properties must be preserved. In a real-time system the specified deadlines for each HRT must be kept at any cost, while the SRT must keep their deadline to a certain degree. This degree is specified by measuring the QoS for the system, which measures how well the system is performing according to a stated definition. If the system recognizes a too low QoS, more CPU units must be activated in order to achieve the desired performance. Supporting real-time requirements will eventually not reduce
energy consumption as well as just an energy policy because of impossible ways of allocating tasks in some cases. The real-time requirements must however be followed if the system shall provide safety and reliability to the user.

The real-time policy increases the complexity of the system further since another part of the CPU monitor must also be running. The real-time monitor, which monitors deadline misses, reports its information to the task migration mechanism. This mechanism only migrates a task if the real-time properties not are violated, i.e. if the system remains schedualable.

Separate points can additionally be dealt to the tasks either manually by the programmer or automatically by the system. These points increases or decreases the probability that a certain task will be migrated or not, and which cores are the best candidates for an incoming task.

In case of emergency, a contingency migration of tasks may move a HRT to a safety-critical core or a non-safety-critical core in order to preserve the hard real-time deadlines until the system is safely shut down or reset. This emergency migration introduces more overhead to the system, and is used for vital tasks which could jeopardize the safety of the users if not working.

Simulations can be used to plot results of task migration. The simulation could for example determine if a HRT should be declared conservative, or how much a SRT migration will lower the QoS of a system. The real-time properties and an estimate of the overhead must be known to the user when making a simulation in order to get the proper response from the system. Graphs can finally show the feasibility of the selected real-time configuration in use. The simulation gives also a overview about what need to be taken into consideration when constructing a migration mechanism and its monitor.
6 Future Work and Conclusions

6.1 Future work

This thesis has set the foundation for the future work of implementing a hypervisor and a task migration mechanism for multi-core embedded systems. Both mechanisms are to a large extent dependent on the way of achieving core-to-core communication. The task migration mechanism is intended to be integrated into FreeRTOS as an extension to the already existing operating system. The implementation must remark the different memory models and communication models that the selected platform will support. A full strategy and specification on how tasks are physically migrated need to be determined before such a construction.

Furthermore, this thesis suggests a monitor for collecting CPU load statistics, and a monitor for preserving the demands of a mixed-critical real-time system. The monitors can be implemented individually, but the introduced overhead must clearly be observed and included into the statistics. The migration of real-time tasks can be additionally simulated using the Simulink model. The simulator currently only supports one-way single task migration and could therefore be extended to support migration of multiple tasks both ways between many cores.

Finally a communication API for both monitors, the task migration mechanism and the inter-core communication is required. The API:s will most likely not be constructed completely generic because of the different communication models etc. Algorithms must be implemented according to the selected model, but should in any case strive to be as generic as possible.

6.2 Conclusions

Since modern embedded systems are becoming more and more complex, there is a need to adjust the implemented software accordingly. Multi-core embedded systems
have a potential to save cost and chip surface due to the decreased size and increased performance, but could compromise certification and re-certification of such systems.

Desktop systems and servers have for a long time used virtualization, and this technology is also now introduced into embedded systems. This thesis has reviewed possibilities of achieving virtualization of embedded multi-core systems with a hypervisor integrated into the existing operating system. The specified hypervisor enables core-to-core communication in a safe fashion, and protects the operating system from the outside environment. Another significant point has been how to accomplish task migration in multi-core embedded systems. Task migration is the groundwork for load balancing and intelligent task allocation. Task migration is defined as moving a running task from one processing element to another without resetting the task. This matter has been discussed in the chapter 3 and 4. A completely generic way of implementing a task migration mechanism is, as concluded, not feasible since the architecture of the platform in question determines the way of how the communication is taking place. One of the leading roles regarding task migration is the memory organization. The memory can be used to execute program code, but also functions as a communications channel or global storage place. A decision on migration can be made locally, globally or in a combined way — this again depending on the current platform, policy or real-time requirements.

Energy policy in mixed-critical systems has been a major issue in this thesis. A policy for minimizing energy consumption and maximizing performance cannot be simultaneously achieved. A system using many CPU cores in parallel will more likely have a higher performance than a system with many sleeping cores, but a sleeping core will require less energy than an active core. The complexity of this matter will increase as real-time requirements are set for the system. The last chapter showed strategies to maintain the rules stated in the energy policy, and follow the real-time requirements put on the system.

This thesis has, in conclusion, presented an underlying groundwork for future implementation of a virtualization layer and task migration mechanism including core-to-core communication for multi-core embedded systems.
**BIBLIOGRAPHY**


77


SAMMANFATTNING

PROCESSMIGRATION I VIRTUALISERADE FLERKÄRNIGA REALTIDSSYSTEM

Introduktion


Inledningsvis undersöks virtualiseringsmöjligheter i samband med olika arkitekturmodeller. Syftet med ett virtualiserat system är att skydda ett operativsystem från omvärlden, d.v.s. låta virtualiseringslagret sköta om kontakten med andra processorkärnor och annan hårdvara. Operativsystemet behöver således inte modifieras för att explicit fungera tillsammans med en viss typ av hårdvara.

Vidare undersöks möjligheter för processmigration (eng. task migration) och belastningsutjämning av processorkärnor i realtidssystem. Processmigration är en metod för att överföra processer mellan processorkärnor. Distribution av processer kan användas för att reglera prestanda och energiförbrukning i ett system. En orsak till varför
man undersöker processmigration är det faktum att energiförbrukning spelar en central roll i hur användbart ett inbyggt system är. Processmigrationsmekanismens krav på prestanda och energiförbrukning har också undersökts.

De uppnådda resultaten kan användas för att skapa en policy som både främjar effektivitet och uppfyller bestämda realtidskrav i ett flerkärnigt realtidssystem.

**Virtualisering**

Virtualisering av ett system innebär att de fysiska resurserna göms för en användare eller operativsystemkärna, för att ersättas med virtuella resurser. De virtuella resurserna översätts sedan till fysiska resurser med hjälp av en virtualiseringsmekanism. I denna avhandling avbildas virtualiseringen med en s.k. hypervisor. Denna typ av virtualiseringsmekanism befinner sig på ett lager mellan operativsystemet och hårdvaran. Virtualiseringsmekanismen sköter om resursfördelning, kommunikation mellan processorkärnor och skyddar operativsystemet från den yttre världen d.v.s. från eventuella ingrepp från övriga processorkärnor.

Det är meningen att den presenterade virtualiseringsmekanismen implementeras i operativsystemet FreeRTOS, som är ett fritt realtidsoperativsystem. Operativsystemet kan skapa instanser av processer som schemaläggs på en processor genom att de sätts in i en arbetsyta. Kontextbytet i FreeRTOS schemalägger processer enligt prioritet och tillstånd, vilket inträffar — vid virtualisering — i virtualiseringslagret istället för i operativsystemkärnan. Virtualiseringsmekanismen tar således kontroll över beräkningseheten vid detta tillfälle och ger tillbaka kontrollen till operativsystemet när vissa berättigade instruktioner har utförts i virtualiseringslagret.

Virtualisering kan förenkla programmering eftersom ett program i viss mån inte behöver anpassas till en viss typ av hårdvara. Virtualisering gör programmeringen generisk, vilket leder till att ett program kan köras på flera olika arkitekturmodeller.

Virtualisationen skall ske under förutsättningen att processer till sin form inte ändras på grund av virtualiseringen, att reallidskrav för systemet inte bryts och att virtualiseringen sker med en acceptabel prestandaförlust.

Processer migreras endast vid vissa kontrollpunkter som sätts ut av programmeraren och som är ämnade för att registrera processens tillstånd på ett förutsägbart sätt. Kontrollpunkterna har en etikett till vilken programräknaren hoppar när en process har förflyttats till en ny kärna. Denna etikett sätts också i normala fall manuellt ut av programmeraren.

Mekanismen som sköter om processmigrationen kan antigen vara global eller lokal. En global strategi innebär att en processorkärna övervakar belastningen i de övriga kärnorna. Beroende på deras tillstånd görs förflyttningar av processer mellan kärnorna. Den kärna som hanterar övervakningsmekanismen kan i vissa fall också användas som en normal processorkärna för normala operationer. Den lokala strategin implementeras så att samtliga processorkärnor kör individuella instanser av övervakningsprogrammet. Med hjälp av kommunikation mellan processorkärnorna görs beslut om migration och till vilken kärna processen förflyttas. I det lokala fallet finns ingen global information överbelastningsutjämning av processer är en viktig orsak till varför processer migreras mellan processorkärnor. En belastningsutjämning är beroende av en policy som styr besluten om migration. Policyn behövs eftersom maximal prestanda och minimal ener-

En övervakningsmekanism kan, lokalt eller globalt, registrera processoranvändningen och på basis av detta fatta beslut om migration. Detta beslut kan dock inte enbart göras på basis av momentana mätningar eftersom belastningstoppar (eng. load peaks) ger felaktiga bilder av statistiken. Mekanismen som fattar besluten bör inneha en viss intelligens för att inte göra felaktiga migrationer. Intelligensen är beroende av systemets behov och prestanda och bör inte överstiga den gräns där effektiviteten börjar sjunka till följd av en för komplex övervakningsmekanism.

Realtidssystem med blandad kritikalitet


Processmigration och kommunikation i ett ifrågavarande system måste i första hand beakta realtidskraven som gäller för systemet. En operation på systemet måste garantera att hårda realtidskrav alltid hinner utföras före deras deadline. En processmigration kan endast utföras om detta krav fortfarande gäller efter migrationen, även
om detta skulle innebära högre prestanda eller att mindre energiförbrukning inte upp-
nås. Ett mjukt realtidskrav får till en viss del missa sin deadline, såvida systemets QoS (Quality-of-System) hålls tillräckligt högt. QoS är ett mått på hur väl systemet preste-
rar i förhållande till förutbestämda mått, såsom bandbredd eller svarstid. En policy för
realtidskrav implementeras för att styra processmigrationerna och kommunikationen
enligt bestämmelser om de hårdare realtidskraven och enligt QoS-måttet.

Simuleringar av dessa processmigrationer har gjorts med en Simulink-modell som
uppritar grafer över processmigrationer med både hård och mjuka realtidskrav. Si-
muleringarna kan användas för att undersöka om en migration är lämplig eller lönsam
för ett system med vissa bestämda förhållanden. QoS-måttet kan till exempel stude-
ras i takt med att processmigrationer görs mellan två kärnor med mjuka realtidskrav,
förutsatt att viss information om processerna är given.

Sammanfattning

Denna avhandling har inledningsvis presenterat virtualisering av flerkärniga system
med hjälp av en virtualiseringsmekanism kallad hypervisor. Virtualiseringsmekanis-
men är konstruerad likt ett lager mellan operativsystemet och hårdvaran för att främja
säker kommunikation mellan CPU-kärnor och för att skydda operativsystemet från ut-
omstående händelser. Vidare har processmigration planerats enligt en ram av modeller
för kommunikation mellan kärnor, övervakningsstrategier och migrationsmekanismer.
Det är meningen att virtualiseringsmekanismen implementeras i operativsystemet Fre-
ereRTOS och detta skall eventuellt förverkligas i framtiden. Olika policy för minime-
ring av energikonsumtion har behandlats, och det har konstaterats att ingen allmän
policy är lämpad för samtliga situationer. Förutom en policy för minimering av ener-
gikonsumtion bör, i ett realtidssystem, även realtidskraven beaktas. I ett system med
bländad kritikalitet måste de hårdare realtidskraven alltid hålla för att systemet skall vara
genomförbart, vilket betyder att tilläggstid i processen till följd av processmigration
måste beaktas mycket noggrant. Simuleringar har gjorts i samband med processmigra-
tion av realtidssystem. Dessa ger en snabb överblick över vad en migrationsmekanis-
m måste baseras på och vilka situationer som måste beaktas vid processmigration.

Framtida implementation av en virtualiseringsmekanism kan förväntas. Ett beslut
om plattform och arkitektur har inte fattats, men härvid kan konstateras att en generell
modell för virtualisering och en mekanism för processmigration är svår att genomföra
i en praktisk implementation. En tänkbar lösning vore att konstruera den ifrågavarande mjukvaran enligt en fastslagen riktning som så väl som möjligt passar de olika modellerna.
A Appendix

A.1 Simulations

Matlab Simulink simulation models presented divided into separate sections depending on functionality.

A.1.1 Task migration to maximize overall performance

Figure A.1 shows the main block for simulating maximal performance. Specific information regarding the *cut limit* block can be obtained from figure A.3. Furthermore, the derivative part is shown in figure A.2 and the critical limit in figure A.4.

![Figure A.1: Block scheme to simulate maximum performance](image)
The simulation model sets scores on the time-axis depending on the load curve and its trend. Situations when the average load is high, as well as when the trend is going upwards will receive a high score. Low average load and high load peaks will receive low score.

Figure A.2: Subsystem defining the derivative of the curve

Figure A.2 is a part of figure A.1.

Figure A.3: Subsystem defining a cut-limit

Figure A.3 is a part of figure A.1.
A.1.2 Task migration to minimize overall energy consumption

The simulating model strives to migrate the task on the core with the lowest load to another core (figure A.5). The scores in this model will be given in form of a binary value attached to each CPU core. The value provides information when a migration from a certain core is possible. The lowest input from the load generators is calculated in the block labeled: check smallest input. The block in figure A.6 calculates the derivative of the input, as well as sets a limit on how high the derivative must be in order to receive a score. Figure A.7 calculates the integral over a certain time-window to determine the average load. Lastly figure A.8 runs logical operations on the results in order to give a migration suggestion.
Figure A.5: Block scheme to simulate minimum energy consumption

Figure A.6: Subsystem defining the derivative of the curve

Figure A.6 is a part of figure A.5.
Figure A.7: Subsystem defining the integral of the curve

Figure A.7 is a part of figure A.5.
A.1.3 Migration of hard real-time tasks

Migration of hard real-time tasks is achieved only if no deadline is missed. The block scheme in figure A.9 migrates the task in the first CPU core to the other CPU core if possible. In case the two tasks are overlapping each other (figure A.10 and figure A.11) the first task is delayed (figure A.12 and figure A.13) a certain time in order to fit into the schedule. The migration in successful if the first task does not miss its deadline due to the delay.
Figure A.9: Block scheme to simulate migration of hard real-time tasks

Figure A.10: Subsystem used to detect overlap situations

Figure A.10 is a part of A.9.

Figure A.11: Subsystem used to detect overlap situations

Figure A.11 is a part of A.9.
Figure A.12: Subsystem used to delay task

Figure A.12 is a part of A.9.

Figure A.13: Subsystem used to delay task

Figure A.13 is a part of A.12.
### A.1.4 Migration of soft real-time tasks

In contrast to the hard real-time task migration, the migrated task is now allowed to miss its deadline. The responsibility is put on the whole system to monitor the QoS value, and based on set rules define if a scheme is feasible or not. Figure A.14 shows the main block, and figure A.15 and figure A.16 shows the subsections containing the overlap checks. The delay in the main block is the same as in previous simulation (figure A.12).

![Figure A.14: Block scheme to simulate migration of soft real-time tasks](image1)

![Figure A.15: Subsystem used to detect overlap situations](image2)

Figure A.15 is a part of A.14.
Figure A.16: Subsystem used to detect overlap situations

Figure A.16 is a part of A.14.