Power-aware Scheduler for Many-core Real-time Systems

AMIRHOSSEIN TAHER KOUHESTANI

Master of Science Thesis
Stockholm, Sweden 2013

TRITA-ICT-EX-2013:154
Power-aware Scheduler for Many-core Realtime Systems

Master of Science Thesis

AMIRHOSSEIN TAHER KOUHESTANI

Examiner:
Mats Brorsson, KTH

Supervisor:
Detlef Scholle, XDIN AB
Barbro Claesson, XDIN AB

TRITA-ICT-EX-2013:154
Acknowledgements

My deepest gratitude goes to my supervisors, Detlef Scholle and Barbro Claesson who provided me with the opportunity to work this thesis at XDIN. Using their constructive guidelines, I have been able to tackle different challenges during my thesis work.

I would like to express my sincere gratitude to Mats Brorsson, my examiner, who kindly provided me insightful guidelines during the implementation phase of my thesis and also gave me the opportunity to implement my design on Parallella platform.

I thank Joana Larsson, Cheuk Leung, José Pérez, Robin Hultman and Tobias Lindblad, my fellow students doing their theses at XDIN. I found our technical discussions very useful and I certainly enjoyed the cheerful environment created with their presence.

I would also like to thank Yaniv Sapir from Adapteva and Artur Podobas for their supports.

Last but not the least, I thank my parents for their endless support during my life.
Abstract

MANY (Many-core programming and resource management for high performance Embedded Systems) project aims to develop a programming environment for many-core embedded systems which would make faster development of applications possible. MANY focusses on exploiting parallelism and resource awareness.

This thesis contributes to the project by investigating possible solutions for scheduling real-time tasks on many-core embedded systems while aiming to reduce power consumption whenever it does not affect the performance of the system.

This thesis has implemented a basic power-aware scheduler for many-core real-time embedded systems. The system is capable of meeting hard-real time deadlines and it can save power by turning off cores which are not needed. This thesis lays the foundation for further developments in the mentioned field.
Contents

1 Introduction .......................... 1
  1.1 Background .................................. 1
  1.2 Problem statement .......................... 1
  1.3 Team Goal .................................. 2
  1.4 Method .................................. 2
  1.5 Delimitations .......................... 2

2 System Requirements .................. 3
  2.1 Requirements overview .................. 3
  2.2 Requirements discussion .................. 3

3 Many-core Systems .................. 5
  3.1 Introduction .......................... 5
     3.1.1 From a single core to many cores .......................... 5
     3.1.2 Architecture Overview .......................... 6
  3.2 Memory Organization .................. 6
  3.3 Interconnection .......................... 6
  3.4 Case Studies .......................... 7
     3.4.1 Kalray MPPA 256 .......................... 8
     3.4.2 Tilera TilePro 64 .......................... 8
     3.4.3 Adapteva Parallella .......................... 9
  3.5 Summary of Many-core Systems ............ 11

4 Many-core Programming ............ 13
  4.1 Parallel Programming .................. 13
     4.1.1 Decomposition .......................... 13
5 Many-core Real-time Scheduling Algorithms

6 Power Management in Many-cores

7 Specification for System Design

8 Implementation: A Power-aware Real-time Scheduler for Parallella
CONTENTS

8.1 Structure of the Scheduler ................................. 37
  8.1.1 Host application ............................................. 37
  8.1.2 Communication between Host and Epiphany ............ 38
  8.1.3 Local Scheduler ............................................... 39

8.2 Scheduler’s overhead ........................................... 42
  8.2.1 EDF scheduler overhead calculation ..................... 43
  8.2.2 Overhead of the implemented scheduler ................. 44

8.3 Summary of the Implementation ............................... 44

9  Power Measurement of Epiphany ............................... 47
  9.1 Power Measurement Method ..................................... 47
  9.2 Power Measurement Results .................................... 47
     9.2.1 Memory management effects .............................. 48
     9.2.2 Power consumption of each core ......................... 49

  9.3 Summary of Power Measurement ............................... 51

10 Conclusion and Future Work .................................... 53
  10.1 Conclusion on the Project ................................... 53
  10.2 Limitations ...................................................... 53
  10.3 Future Works .................................................... 54

Bibliography
List of Figures

3.1 Conventional interconnections [37] ........................................... 7
7.1 Design space of multi-processor real-time scheduling [29] ........... 31
7.2 Task partitioning and splitting .................................................. 33
7.3 Different Scenarios when a new tasks arrives ............................ 34
7.4 An example of scheduling tasks using EDF ............................... 35
8.1 Partitioner pseudo code ........................................................... 38
8.2 Flowcharts of local scheduler and timer interrupt routine .......... 40
8.3 The prolog generated by the compiler for the interrupt handler routine . 41
8.4 Snippet of extended assembly code used in context store process .... 42
8.5 Snippet of extended assembly code used in context restore process .... 42
9.1 The code used to measure power consumption .......................... 48
9.2 Power consumption vs Execution time ...................................... 50
9.3 Power consumption of Epiphany cores with and without floating point unit .......................................................... 50

List of Tables

3.1 Hardware Comparison ............................................................. 11
9.1 Memory management [2] .......................................................... 49
List of Abbreviations

ALU Arithmetic Logic Unit
ASIC Application-Specific Integrated Circuit
CPU Central Processing Unit
DMA Direct Memory Access
DSP Digital Signal Processor
DVFS Dynamic Voltage and Frequency Scaling
FPGA Field-Programmable Gate Array
GPU Graphics Processing Unit
MMU Memory Management Unit
NoC Network on Chip
VLIW Very Long Instruction Word
WCET Worst Case Execution Time
Chapter 1

Introduction

1.1 Background

One of the major turning points in computer world, was the migration from single core to multi-core systems. As it can be seen today, multi-core systems are widely growing and multi-core hand held devices have already emerged in daily life. As the need for computation power grows, it will not take long to see that multi-core devices are being replaced with many-core systems. Many-core systems, accommodating bigger number of cores on a chip, offer enormous performance gain and at the same time introduce some technical challenges. One of the major challenges regarding many-core technology is the power consumption, as the number of cores is growing. Power efficiency becomes of more importance and necessity when it comes to the embedded systems where resources are limited.

A considerable part of embedded systems are real-time systems. Systems which are supposed to execute periodic tasks within certain amount of time and meet the deadlines. Some of power consumption reduction techniques decrease the performance of system which might result in missed deadlines. In many-core hard real-time systems, reducing power consumption becomes a challenge since meeting the tasks deadlines must be guarantied.

1.2 Problem statement

While designing a power efficient embedded system, many factors come into play from hardware and software point of view. One of the areas in which there is a possibility to improve power consumption is system level software of the system. This thesis focuses on improving power consumption through the scheduler of the system. Different methods of scheduling and task assignment to cores, greatly affect the power consumption of the system. The aim is to provide the infrastructure to implement a power-aware scheduler for many-core real-time systems by investigating the state of the art algorithms existing in this field of study. In order to achieve this goal, following questions will be answered:
• What scheduling algorithms exist for many-core real time systems?
• What power management techniques are exploited by the many-core systems?
• How does power consumption relate to scheduling algorithm?

1.3 Team Goal

The developed scheduler would be part of a bigger project which is development of a real-time distributed system, emulating a brake by wire system. The system will include different platforms such as Linux, OSE and Android systems. A middleware will be developed to schedule and manage the distributed embedded system which will work based on message passing technique.

1.4 Method

The thesis is roughly consisted of two phases. The first phase is the theoretical part which is studying the available literature on the subject including previous theses, papers and books. An investigation will be carried out to evaluate a variety of power-aware many-core real-time schedulers. At the end of this phase, the platform for implementation is determined and a system specification is proposed for the scheduler.

The second phase would be the implementation part which includes implementing a power-aware scheduler for a selected many-core system as part of a bigger system which is described in the previous section. The implementation will be based on the proposed scheduler specification and will follow the results of the studies previously done in the first phase.

1.5 Delimitations

Generally in real life systems, a system includes a number of tasks which some of them may depend on each other for execution. In a real-time systems, tasks interdependency introduces challenges which may jeopardize the reliability of the system in meeting hard deadlines. Therefore, the proposed scheduler considers tasks to be independent, eliminating the need for inter-task communication which simplifies the system and increases the overall reliability of a hard real-time system.
Chapter 2

System Requirements

In order to provide a better overview of the system, some requirements are considered for the scheduler, directing the academic study and the implementation of the system. Scheduler aims to satisfy the following requirements.

2.1 Requirements overview

REQ_1 For periodic tasks, the scheduler must meet the hard deadlines.

REQ_2 For aperiodic and sporadic tasks, the system shall try to meet the deadlines.

REQ_3 The scheduler must aim for the least power consumption on processor level.

REQ_4 The scheduler shall be able to support task migration among cores.

REQ_5 The scheduler shall be scalable as the number of cores changes.

2.2 Requirements discussion

In the following, above requirements are discussed and motivated.

REQ_1 For periodic tasks, the scheduler must meet the hard deadlines.

The scheduler targets many-core hard real-time systems, and as in any hard real-time system, deadlines must be met. In hard real-time systems, in order to improve the system’s reliability, tasks are known to the system, meaning that the necessary information of all tasks such as worst case execution time and period, exist in the system, prior to the execution.

In this scheduler as well, the system is considered to have a number of already known periodic tasks. These tasks should be profiled and the feasibility of the
taskset should be verified. The system must meet the deadlines for these hard periodic tasks.

**REQ_2** For aperiodic and sporadic tasks, the system shall try to meet the deadlines.

The main goal of the system is to meet the deadlines of periodic hard real-time tasks. However, the system tries to schedule aperiodic and sporadic tasks, if and only if, scheduling these tasks does not result in missed deadlines of main periodic tasks. In other words, any new arriving task, could be scheduled if there is enough capacity in the system, otherwise the new task is discarded.

**REQ_3** The scheduler must aim for the least power consumption on processor level.

Generally in embedded systems, resource management plays a vital role and among resources, power is of high importance. There are different techniques to reduce the power consumption of a system. The power consumption could be reduced in different areas of a system such as memory and cache, input/output, processor and etc. The proposed scheduler focuses on reducing the system power consumption on processor level. As it is further discussed in chapter 6, techniques such as DVFS and clock/power gating are exploited to reduce processor power consumption. The scheduler uses one or more of these techniques to satisfy this requirement.

**REQ_4** The scheduler shall be able to support task migration among cores.

In some classes of scheduling tasks over many-core systems, task migration could improve the performance of the system. Task migration allows a task running on a specific core, to migrate and continue execution on another core in the system. As it is further discussed in chapter 5, scheduling algorithms which allow task migration have higher rates of system utilization. On the other hand, task migration could impose a considerable amount of overhead on the system. Therefore, task migration is considered for the scheduler, if the gain of higher system utilization is more than the drawback of it’s overhead.

**REQ_5** The scheduler shall be scalable as the number of cores changes.

As the number of cores in many-core systems are growing, in order to have a generic and reusable system, the scheduler shall be able to work with variable number of cores. This requirement suggests that the scheduler should be designed and implemented in a way that it does not rely on the number of cores as a structural factor. Number of cores should only be considered as an input to the scheduler to organize the system accordingly.
Chapter 3

Many-core Systems

3.1 Introduction

3.1.1 From a single core to many cores

Single core to Multi cores

In the single core era, as the size of transistors shrank, it was possible to fit more transistors in a chip and by increasing the operating frequency, more performance could have been gained. However in the last decade, this trend of technology scaling hit the wall. Power consumption, thermal issues and fabrication problems turned out to be crucial limiting factors. The continuous demand for higher performance on one hand and problems regarding enhancing performance in single core architecture on the other hand, triggered the migration from single core to multi core CPUs.

Multi cores to Many cores

As the demands for higher performance increase, we have witnessed a continuous grow in the number of cores. Systems with multiple cores are quite trivial nowadays. It is anticipated to have systems with more than a thousand cores within the upcoming years [11]. It has not been clearly defined what the exact difference is between multi-core and many-core systems, regarding the number of cores. Generally, systems with less than 8 cores are called multi-core and systems with more than that are many-cores.

From application point of view, it may seem that there is not much difference between multi-core and many-core systems, however it is not completely true. Shifting from multi-core to many-core brings some challenges into play. In a multi-core system, usually tasks are mapped statically to the cores, but many-core systems tend to use dynamic task mapping [39]. The other difference is that multi-core systems usually have more complicated cores, regarding the pipeline depth, out of order execution and branch prediction. However, cores have simpler architectures in many-core systems. One other major difference is the role of the network on chip in many-cores. In multi-cores, due to the few number of cores, the interconnection
network can be neglected, but in many-cores, this is not the case. The network on chip in many-cores has a considerable effect on the performance, depending on the topology and the routing algorithm which is implemented [32].

3.1.2 Architecture Overview

A many-core system is mainly consisted of processing cores, interconnection network and main memory. Based on the type and similarity of the cores, many-core systems are divided into two categories:

**Homogeneous**

Homogeneous many-core systems are consisted of several identical cores. Today, most of the many-core systems are homogeneous. All cores share the same instruction set architecture which makes them easier to program, and they also have the same performance metrics [37].

**Heterogeneous**

Heterogeneous many-core systems are consisted of at least two cores with different architecture. A heterogeneous system can include different combinations of general purpose CPUs, GPUs, DSPs, FPGAs and ASICs. Heterogeneous systems are mostly used for application specific systems [37]. It is predicted that the future of many-core systems will be more heterogeneous [11].

3.2 Memory Organization

Typically memory in many-core systems is a non-uniform memory architecture. A many-core system usually has one or two levels of caches dedicated to each core and a shared memory accessible by all the cores. The memory access time is different for each core depending on their location in the interconnection network.

3.3 Interconnection

There exist different topologies to provide the core to core, core to memory and core to I/O communication. Common bus, cross-bar, ring and on-chip mesh networks are some of the conventional interconnects. Figure 3.1, taken from [37], illustrates the mentioned interconnects. The main concerns regarding the design of an interconnection are the latency, bandwidth and scalability.

First multi-core systems use a common bus as the interconnect. The bus also implements cache coherency. There are two major issues when using a common bus as the communication interface. The first problem is the latency of the common bus due to long wires. The second problem is limitation of bandwidth that a common
bus imposes. Since every node in the system is connected to the bus, practically, the bandwidth is limited to the share of each node from the bus [37].

To improve the performance of the interconnection considering bandwidth and scalability, packet switched on-chip networks are exploited. There is a router at each node, connecting it to the neighbour nodes with short wiring. There are four parameters that can be used to define an on-chip network. The topology, routing algorithm, flow control protocol and router architecture. Each of these four parameters plays a vital role in providing the necessary functionality of the on-chip network and depending on them, several architectures have been proposed. Although packet switched on-chip networks outperform buses and cross-bars, care should be taken about the power and area consumption of such interconnection [26].

![Figure 3.1. Conventional interconnections [37]](image)

### 3.4 Case Studies

In this section, three many-core systems are introduced.
3.4.1 Kalray MPPA 256

Kalray MPPA 256 [24] is a homogeneous, shared memory, C/C++ programmable many-core chip. It is consisted of 16 clusters, each consisting of 16 cores, 4 I/O subsystems and two NoCs connecting them all.

Each Cluster

Each Cluster is consisted of 16 identical cores, a system core, 2 MB of shared memory and a DMA. Each cluster supports dynamic voltage and frequency scaling and dynamic power switch off. The cores in a cluster are connect to the memory by a cross-bar.

Each Core

Each core is a 32-bit VLIW processor, including a branch/control unit, two ALUs, a load/store unit with simplified ALU, a multiply-accumulate/floating point unit, a MMU, 8 KB L1 data and 8 KB L1 instruction caches.

Network on Chip

The network on chip in MPPA 256 is a 2D-wrapped-around tore structure. Each cluster is connected to 4 neighbor clusters and the side clusters are also connected to the I/O subsystems.

3.4.2 Tilera TilePro 64

Tilera TilePro64 [36] is a homogeneous, shared memory, C/C++ programmable, many-core computing platform based on Tilera iMesh technology. The TilePro64 is consisted of 64 tiles (cores), structured as 8*8 mesh network on chip, all connected to each other by iMesh. Each tile is also connected to the memory and I/O.

Each Core

Each tile is a 32 bit integer VLIW processor with 3-stage pipeline, including a memory management unit, a register file, 16 KB L1 and 64 KB L2 cache and a DMA. L2 caches of 64 tiles form a 5.6 MB virtual L3 cache. Considering the available components, each tile is functionally complete and is capable of running an OS individually.

Memory

TilePro64 has a flat memory with 32 bit addressable space. It provides TLB to support paged virtual memory system. There are three different memory modes: physical memory mode, cashe as RAM mode and striped memory mode.
Network on Chip

iMesh is the interconnection network of TilePro64, consisted of 6 separate networks with different functionalities, including 5 dynamic networks which exploit packet based communication and a static network. The networks are as follows:

- **UDN** The user dynamic network can be used to provide inter process communication between tiles. This network is accessible through software.
- **IDN** The I/O dynamic network can be used both for inter process communication and I/O communication. IDN is accessible through software and it is meant to be used by supervisor processes.
- **MDN** The memory dynamic network is only accessible by the cache engine. It is used to provide the cache to cache and cache to memory communication.
- **TDN** The tile dynamic network is also for cache to cache communication, in case a tile intends to use other tile’s cache.
- **CDN** The coherence dynamic network is used by the cache coherence module to maintain cache coherency.
- **STN** In contrast to the above networks, the static network is not dynamically routed. STN provides a static point to point communication over a fixed route which makes it favorable for high performance communication. STN is accessible through software.

### 3.4.3 Adapteva Parallella

Parallella [1] board is an open heterogeneous computing platform which is consisted of a Xilinx Zynq family FPGA and an Adapteva Epiphany with 16 or 64 cores coprocessor. The communication between the dual-core ARM processor and the Epiphany chip is provided through the eLink interface and AXI bus.

#### Zynq FPGA

Zynq FPGA is the host to an ARM Cortex-A9 dual-core CPU and also leaves some programmable logic for user defined hardware modules. The ARM processor has 32 KB level 1 and 512 KB shared level 2 cache and can operate at frequency up to 1 GHz.

#### Epiphany

The Epiphany is a shared memory, C/C++ programmable multi-core, structured as a 2D array of mesh nodes. Each mesh node includes a RISC CPU, a DMA engine, a network interface and local memory. The epiphany is compatible with many
programming models such as single instruction multiple data, single program multiple data, host slave programming, multiple instruction multiple data programming, shared memory multi threading and message passing.

Memory

The Epiphany has a 32 bit addressable memory. Each node has 32 KB of local memory which is accessible only by the node itself, serving as cache. Every node has an identifier that enables other nodes to address a specific node, making the inter node communication possible.

Network on Chip

Epiphany’s interconnection network is called eMesh. It is a 2D network which is consisted of three separate channels with different functionalities and routers on every mesh node. Each router is connected to it’s mesh node and to north, south, west and east. A transaction between two adjacent nodes takes 1.5 clock cycles. The three networks that connect all the mesh nodes are as follow:

- **cMesh** This channel is used for on-chip write transactions, providing a throughput of 8 bytes/cycle.
- **rMesh** This channel is used for read transactions, with a throughput of 1 read transaction every 8 cycles.
- **xMesh** This channel is used for off-chip write transactions.
3.5 Summary of Many-core Systems

In this chapter, the migration from single core systems to many-core systems is discussed. An overview on the general architecture of many-core systems is provided and eventually three many-core platforms are briefly introduced. Table 3.1 compares different characteristics of the three mentioned many-core systems. Information in the table are extracted from Kalray brochure [24], Tilera brochure [36] and Parallella reference manual [1].

Table 3.1. Hardware Comparison

<table>
<thead>
<tr>
<th></th>
<th>TilePro64</th>
<th>MPPA-256</th>
<th>Parallella-16</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Architecture</strong></td>
<td>Fully homogeneous</td>
<td>Homogeneous divided into clusters</td>
<td>Heterogeneous with a CPU and a coprocessor</td>
</tr>
<tr>
<td><strong>Number of cores</strong></td>
<td>64</td>
<td>16 clusters of 16 cores</td>
<td>Dual-core ARM CPU + Epiphany 16 cores</td>
</tr>
<tr>
<td><strong>Frequency</strong></td>
<td>700MHz, 866MHz</td>
<td>400MHz</td>
<td>800MHz</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>-</td>
<td>230 GFLOPS</td>
<td>25 GFLOPS</td>
</tr>
<tr>
<td><strong>NoC</strong></td>
<td>2D Mesh with 6 channels</td>
<td>2D Wrapped around tore</td>
<td>2D Mesh with 3 channels</td>
</tr>
<tr>
<td><strong>NoC bandwidth</strong></td>
<td>3.4 GB/s</td>
<td>3.2 GB/s</td>
<td>1.6 GB/s between Zynq and Epiphany, 3.2 GB/s inside Epiphany</td>
</tr>
<tr>
<td><strong>DVFS support</strong></td>
<td>No</td>
<td>For each cluster</td>
<td>No</td>
</tr>
<tr>
<td><strong>Core sleep</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Power</strong></td>
<td>19-25 Watts</td>
<td>N/A</td>
<td>5 Watts</td>
</tr>
</tbody>
</table>
Chapter 4

Many-core Programming

4.1 Parallel Programming

To enable an application to exploit the parallelism available at hardware layer, the application should be decomposable to different portions. This way each available hardware unit could execute a portion of the application in parallel with other hardware units. The application decomposition should be followed by synchronization of different portions of the application to make sure that the application consistency is preserved.

4.1.1 Decomposition

There are two fundamental approaches toward decomposition, data decomposition and task decomposition. Selecting of one approach over the other, depends on the nature of the problem.

Data Decomposition

In data decomposition approach, the data that should be processed is divided into chunks. Same instructions operating on different data chunks, can be executed in parallel on different cores. This approach is efficient when the data chunks can be processed independently [31].

Task Decomposition

In task decomposition approach, the whole problem or parts of it are divided into tasks. A task is a sequence of program that can be executed independently, in parallel with other tasks. This approach is beneficial when tasks maintain high levels of independency [31].
4.1.2 Communication and Synchronization

Often it is not possible to have fully independent tasks in the real world and tasks need to be able to communicate with each other. There are two main reasons why synchronization and communication between tasks are vital. The first one is the data dependency, meaning that task A needs input from task B to be able to continue its job, therefore it should wait for task B to prepare the necessary data. Data dependency is related to how the program is being decomposed. The second reason is resource contention which is due to the fact that many resources are shared among tasks. Resources can be the data that is being operated on, peripherals, memory and so on. Many solutions have been implemented to tackle this problem. The key idea of all of them is maintaining a flow in which access to shared resources happens in turn [37]. Locks, semaphore, monitor, critical section and transactional memory are examples of proposed solutions.

4.2 Inter-process Communication

Fundamentally, there are two approaches toward inter-process communication, shared memory and message passing. Depending on the hardware architecture and the application, one of the two or a combination of them may be chosen as the communication method.

4.2.1 Shared Memory

Shared memory is the most commonly used [37] and easiest [22] way of communication between processes and threads. In this method, memory is shared between processors and it is used as a medium of communication in a sense that a processor can write in a certain memory address and other processors can read it and also write on it. In this approach the main challenge is to preserve the memory consistency.

Traditional Shared Memory

In traditional shared memory method, all processors have access to all the address space available in memory. There are two major issues regarding this method. The first issue lies in the application layer. Memory accesses to the same location must be monitored and if necessary, prohibited. As a result, synchronization methods such as locks, semaphores and transactional memory need to be deployed to serialize the memory accesses, which will eventually lead to delayed execution. The second issue is on the hardware level. Since each processor has a local copy of some memory locations in its cache, the cache coherency must be maintained to preserve the memory consistency. Implementing cache coherency protocols increases to the execution time of the application and also results in a more complex hardware.
The traditional shared memory method lacks proper scalability. As the increase in number of processors leads to more accesses to the shared memory, the execution time of applications will suffer drastically, since more threads would have to wait to access the memory [37].

**Follow the Data Pattern**

In order to tackle the two major issues of the traditional shared memory method, a method called follow the data has been proposed. In this method, a thread can only access memory through a specified core called the resource guardian core. When a thread requests a memory access, it is blocked and then migrated to the resource guardian core where it can access the shared memory. This method is beneficial because it is possible to avoid memory access conflicts and also it is more efficient in exploiting the memory bandwidth [38]. The drawback of the simplified follow the data pattern method, is that even threads whose memory accesses are conflict-free, will be blocked. To resolve this problem, generalized follow the data pattern, groups critical sections and appoints a resource guardian core to each group of critical sections. This way the mentioned problem is resolved and the scalability is improved, however such a design is prone to dead-locks. This problem could be overcome by having resource guardian cores communicate with each other.

Follow the data pattern method offers a dead-lock free solution for using shared memory. It also improves the memory performance for applications that the size of the data is much larger than size of the code. There is another positive point to this method which addresses the hardware. Since all the memory accesses go through the resource guardian cores and there is no shared memory on the hardware layer, the hardware components for implementing cache coherency protocols is not needed any more and could be removed. It would increase the predictability and determinism of the system and also enhance the power efficiency of the processor [37, 38].

**4.2.2 Massage Passing**

Massage passing approach provides inter-process communication through exchanging messages between threads and processors. Unlike shared memory which is based on sharing variables, message passing is based on the idea of isolating the threads and processors. Message passing implementation and programming is more difficult compared to shared memory approaches [18], however it increases the system’s reliability since threads are isolated and a thread’s malfunction is less probable to affect the functionality of the system. Message passing is the natural choice for architectures with distributed memory [37].

**4.2.3 Partitioned Global Address Space**

Shared memory programming models are generally easier to use while message passing programming models offer more scalability. The partitioned global address space model was proposed to combine the positive aspects of shared memory and
message passing models. The main reason why shared memory programming models do not scale well is due to their inability to exploit locality effectively. In PGAS programming models, the shared memory is partitioned and each thread gets to have its own local memory, enabling threads to exploit locality [15].

4.3 Parallel Programming APIs

In this section some of the popular parallel programming APIs have been briefly studied. None of the following APIs is used in the implementation phase of this thesis, however since they provide a valuable insight to the parallel programming, these APIs are introduced here.

4.3.1 POSIX Threads

This programming model uses threads as the basic blocks to maintain parallelism. A thread is an execution flow which has its own program counter and stack. Posix threads or Pthreads is a low level application programming interface close to the operating system level. It is implemented as a set of routines in C language packed in libraries to create, destroy and manage threads. As the Pthreads is a shared memory programming model, it needs to deal with the challenges that exist in shared memory model. Therefore, it provides the developer with necessary tools such as mutex and semaphores to protect critical sections. In [18], it is stated that Pthreads is not easily scaled to large number of cores because of its unstructured nature and also the fact that the number of threads is regardless of the number of the processors.

4.3.2 OpenMP

OpenMP is an application programming interface based on shared memory programming model. Unlike Pthreads, OpenMP functions on the user level. It can be added to Fortran or C/C++ sequential applications. OpenMP is implemented as compiler directives, library routines and a run time system. Using OpenMP, developer can specify the portion of the code that needs to be executed in parallel and also the number of threads that should be created to run that portion of the code. OpenMP is easy to learn and use, since its semantics is close to the sequential programming. It also hides the complexity of parallel programming details from developers and lets the OpenMP compiler take care of them [13].

OpenMP uses a fork/join programming pattern [31]. At the beginning of each parallel region, a number of threads are created(forked) and at the end of the parallel region, the created threads are synchronized(joined). After OpenMP 3.0, it is possible to define explicit tasks, enabling developer to dynamically generate tasks. This new capability increases OpenMP power to offer more parallelism. OpenMP is suitable for both data and task decomposable problems.
4.3.3 MPI

MPI is a standard for message passing programming model, implemented as a library that should be linked with normal C/C++ or Fortran programs. It is best suited for distributed memory architectures with separate address spaces. MPI is mainly consisted of two fundamental elements, process groups and a communication context. A process group includes a number of processes that work on a computation. Unlike OpenMP, MPI is concurrent from the very beginning [28]. All processes working on a computation will start together and the user can manage how different process groups interact. The communication context provides the communication means for processes, managing the delivery of messages from senders to receivers [31].

In MPI, similar to Pthreads, workload distribution and task mapping is done by the developer. MPI provides different communication models such as point to point operation for passing messages between two processes in a synchronous manner, collective or global operation for communications of more than two processes, on-sided communication providing asynchronous message passing suitable for remote memory accesses and parallel I/O operations to access external devices [18].

4.3.4 OpenCL

OpenCL is a parallel programming standard designed for heterogeneous systems. OpenCL is mostly used in architectures consisting of CPUs and GPUs, however it also works on architectures with CPUs and other types of accelerators. OpenCL defines kernels as the basic blocks of parallelism. OpenCL is implemented as a C like language called OpenCL C to write kernels and a C language API to invoke the kernels [23]. In OpenCL model, The host manages the execution and communication of the computation across different computing devices. OpenCL executes instances of kernels called work-items on different OpenCL devices. Work-items can be grouped in work-groups for communication and synchronization reasons [18].

OpenCL offers functional portability and it suits SIMD and SPMD [31] programming patterns [33].

4.4 Summary of Parallel Programming

In this chapter, basics of parallel programming are discussed. Different approaches that exist to deploy hardware level parallelism are introduced, data decomposition and task decomposition. Further on, different methods and means of communication between processes are discussed, explaining their pros and cons. Shared memory is the easier method of inter-process communication, however it needs memory monitoring and coherency protocols and it is not easily scalable. On the other hand, message passing is a more scalable and robust communication method, but it is more difficult to implement. Eventually four popular parallel programming APIs
are briefly introduced, each having different approach to parallel programming, suitable for different hardware architectures and different type of applications.
Chapter 5

Many-core Real-time Scheduling Algorithms

As today, multi-core systems are being used in embedded systems, it will not take long to witness many-core systems are embedded in process demanding real-time systems. This chapter covers the existing real-time scheduling algorithms for many-core systems as one of the two main requirements of this thesis.

5.1 Attributes of Scheduling Algorithms

5.1.1 Task Model

In real-time systems, generally two types of tasks exist, periodic and sporadic tasks. In periodic tasks, jobs of the tasks execute periodically with a fixed time interval. In sporadic tasks, a job of a task could happen at any time.

This thesis uses the same notations as [16] to define real time tasks. Each task $\tau_i$ has a set of parameters including a relative deadline $D_i$, a worst case execution time $C_i$ and a period $T_i$. The utilization $u_i$ of task $\tau_i$ is $\frac{C_i}{T_i}$ and the utilization of a taskset is $u_{sum}$ which is the sum of all the utilizations of the tasks included in a taskset.

5.1.2 Taxonomy of Scheduling Algorithms

Many-core real-time scheduling algorithms can be categorized in different classes depending on the underneath hardware, task model and scheduling approach.

Hardware

The hardware clearly affects the design of the scheduler and depending on the type of hardware, whether homogeneous or heterogeneous, different scheduling algorithms exist.
Task Interdependency

Real-time scheduling algorithms can be divided into two groups regarding the tasks interdependency. The first group of algorithms which most researches focus on, consider the tasks to be independent from each other. However, this is not always the case in the real world. The second group of algorithms take the task interdependency into account and consider the blocking time of accessing shared resources in their scheduling.

Allocation

There are two classes of algorithms while it comes to allocating tasks to processors. The first class of algorithms called partitioned scheduling, appoints each task to a processor and do not allow the task to migrate to other processors. The second class of algorithms is the global scheduling in which tasks are dynamically appointed to processors and depending on the circumstances, tasks can be migrated from one to other processors.

Priority

Scheduling algorithms can be categorized in two groups regarding the tasks priorities. In fixed task priority algorithms, a task has a static priority which will remain constant through time. In dynamic priority algorithms, tasks can have different priorities at different times.

5.1.3 Scheduling Algorithms Metrics

To be able to compare different scheduling algorithms, some performance metrics are required.

Utilization Bounds

For tasksets in which deadline of the tasks are equal to periods of the tasks (implicit-deadline tasks), worst case utilization bound $U_A$ for scheduling algorithm $A$, is the minimum utilization of any taskset that is only schedulable using algorithm $A$. Therefore, any taskset with utilization less than $U_A$ is schedulable using algorithm $A$ and any taskset with utilization greater than $U_A$ is not schedulable with algorithm $A$ [16].

Approximation Ratio

A scheduling algorithm is considered to be optimal, if all the feasible tasksets that comply with the algorithm’s task model can be scheduled using this algorithm. Approximation Ratio $R_A$ is a comparison of an algorithm $A$ with an optimal algorithm [6]. In the context of multi-core real-time scheduling algorithms, it is used to compare the number of processors that an algorithm $A$ needs to schedule a taskset with
the number of processors an optimal algorithm needs. $R_A = 1$ signifies an optimal algorithm and $R_A \geq 1$ with smaller value of the approximation ratio, implies that algorithm A is a more performance effective scheduling algorithm [16].

5.2 Scheduling Algorithms for Homogeneous Systems

5.2.1 Partitioned Scheduling

In partitioned scheduling the scheduling can be divided into to phases. The first phase is to appoint tasks to available processors which is comparable to NP-hard bin packing problem [20]. At this phase the bin packing heuristics such as first fit, next fit, best fit and worst fit have been used, while the utilization list of tasks is sorted in a descending order. The second phase is to schedule appointed tasks on each processor which can be done using rate monotonic, deadline monotonic and earliest deadline first scheduling algorithms.

Many partitioned scheduling algorithms have been proposed in the last decades and they mostly combine bin packing heuristics with RM and EDF scheduling algorithms. RMST [12], EDF-FF [20], EDF-BF [20], EDF-WF and RBOUND-MP-NFR [4] are important examples of such algorithms.

5.2.2 Global Scheduling

Global scheduling can be divided into two classes. The first class of algorithms is designed for tasks with fixed priority. The core idea of most of these algorithms is global EDF or RM and DM, depending whether it is fixed job priority or fixed task priority. The second class of algorithms is designed for tasks with dynamic priority. These algorithms are based on the concept of fluid scheduling.

Tasks with Fixed Priority

Global EDF scheduling is the basic idea for algorithms that have fixed job priority. However, when applying single core priority assignment algorithms to multi-core systems, the system will suffer from a problem known as the Dhall effect [17]. The Dhall effect occurs when a taskset with low utilization (compared to what the system can handle) cannot be scheduled. This problem happens when a high utilization task is blocked by some smaller tasks, leading to missed deadline. Algorithms such as EDF-US [35], EDF($k$) [21], EDF($k_{min}$) [7] and EDF-DS [9] have been proposed to work around the Dhall effect. The general idea of these algorithms is to assign higher priorities to task with high utilization.

Global scheduling algorithms designed for fixed task priority, mostly use RM or DM algorithms. Algorithms such as RM-US [3], SM-US [5], DM-DS [10] are proposed as fixed task priority algorithms that can nullify the Dhall effect.
Tasks with Dynamic Priority

Global dynamic priority scheduling algorithms are based on the fluid scheduling model [16]. Pfair [8] algorithm (and its variants) and LLREF [14] are optimal scheduling algorithms for implicit deadline tasksets. However, there is no optimal global dynamic priority algorithm for preemptive scheduling of sporadic tasksets.

Pfair is a scheduling algorithm designed for periodic tasksets with implicit deadlines. In Pfair scheduling algorithm, time is divide into quanta or slots and each task is also divided into pieces that can fit into time slots. At each time slot, the scheduler is invoked to make scheduling decisions. In this approach, tasks progress is proportional to their utilization. Pfair maintains fairness in the sense that each task receives a share of processor's time, enabling them to make simultaneous progress through time. It is demonstrated that Pfair is an optimal algorithm for mentioned tasksets. Pfair has some variations which each of them improves a certain aspect of this algorithm. One example of algorithms that improves Pfair is BF algorithm introduced by [40] and it is similar to Pfair conceptually. However the scheduler is only invoked at period boundaries, not at the end of each time slot. This will lead to 25%-50% less scheduler invocations, thus less overhead on the system, compared to Pfair and its variants.

LLREF is also an optimal scheduling algorithm based on the fluid scheduling model, maintaining the fairness. LLREF exploits an abstraction called Time and Local Execution Time Domain Plane. Tasks are scheduled based on the local remaining execution time. The fairness is maintained by assigning higher priorities to tasks with more local remaining execution time. This algorithm was further improved by REF to a new algorithm called LRE-TL [19]. LRE-TL states that there is no need to select tasks with largest local remaining execution time. Selecting any task that has some local execution time left will also lead to the same result. LRE-TL effectively reduces the number of tasks migrations.

5.2.3 Hybrid Scheduling

Each of the partitioned and global scheduling approaches has its own pros and cons. In partitioned scheduling approach, the task assignment to cores is based on heuristics which will reduce the optimality of an algorithm. Another major issue is that the partitioned scheduling approach will result in processing capacity fragmentation which drastically reduces performance. [29] states that the maximum utilization bound for partitioned scheduling is only 50%. On the other hand global scheduling approaches could impose considerable overhead on the system. Also migration of jobs will result in increased number of cache misses and also increases the interconnection network's traffic. It should also be considered that managing a global task queue involves more complexity than managing several local queues. Another issue regarding the global scheduling algorithms is the fact that if a core fails to function correctly, the problem would be propagated to the whole system and may lead to a system failure. To address these issues, hybrid algorithms have
been proposed that try to mitigate the mentioned problems by combining aspects of the two approaches.

**Semipartitioned**

Semipartitioned algorithms use task splitting to consume the fragmented processing capacity. The basic idea of these algorithms is that some tasks are scheduled using some of the partitioned algorithms and the remaining tasks are split into a number of components and each component is executed by a processor. This way the fragmented processing capacity is filled with portions of tasks.

**Clusters**

In cluster scheduling approach, few number of processors form a cluster and tasks are assigned to the clusters rather than processors. Using this approach, the complexity of task assignment is reduced, since the number of bins is decreased. Also the number of migration is reduced since the migration is only allowed within a cluster.

### 5.3 Summary of Many-core real-time scheduling algorithms

In this chapter, different classes that exist in scheduling many-core real-time systems are introduced, as well as metrics that help evaluate an algorithm. Further on, this chapter answers the first question raised in the introduction chapter: 

*What scheduling algorithms exist for many-core real-time systems?*

The answer to this question is provided through introducing different real-time scheduling algorithms for homogeneous many-core systems.

As explained in this chapter, partitioned algorithms cannot utilize CPU more than 50\%, however they have less overhead than global scheduling algorithms. Global scheduling algorithms, theoretically can reach 100\% utilization, however they are hardly implementable, considering their excessive overhead on the system. Intermediate solutions are provided which improve the utilization in comparison with partitioned algorithms and has less overhead compared to global scheduling algorithms. Examples of this class of algorithms are semi-partitioned and cluster algorithms.
Chapter 6

Power Management in Many-cores

Intel CPU cancellation due to its massive power consumption was a sign indicating the single core era has come to an end. Power issues were one of the main reasons that triggered the migration from single core processors to multi and many-core systems. Many-core power management becomes of more importance as many-core chips will be deployed in embedded systems. With a limited number of logic transistors, it is more beneficial to have multiple smaller cores rather than having a larger, more complex core. The benefit comes from the fact that the performance of a smaller core reduces by the square root while the power is reduced linearly. It seems that many-core systems are able to improve the performance while still fitting in the power envelope. However, there are important factors which must be considered, like the number of cores operating at the same time and the power consumption of the network on chip connecting the cores.

6.1 Power Model in Many-cores

6.1.1 Processor Power Model

Power in CMOS circuits is consumed in two manners, static power consumption and dynamic power consumption.

\[
P_{Total} = P_{Static} + P_{Dynamic}
\]

(6.1)

Static power is consumed when the chip is powered on, even if it is not in use and transistors are not switching. The static power is consumed because of the current that leaks from the different components of a transistor. The current leakage is mainly fed by subthreshold leakage and gate oxide tunnelling. In the following equation, \(I_{CC}\) is the overall leakage current and \(V_{CC}\) is the supply voltage [34].

\[
P_{Static} = I_{CC} \times V_{CC}
\]

(6.2)

In CMOS technology, by scaling down the size of the transistors, it is possible to increase the operating frequency and also decrease the supply voltage. However,
by reducing the size of transistors, the leakage current would exponentially grow, inducing a serious power dissipation issue.

Dynamic power is consumed whenever a transistor switches from one state to the other. Dynamic power is consumed to charge the load capacitance in the output. The following equation calculates the dynamic power consumption, where \( N \) is the number of bits switching [34].

\[
P_{\text{Dynamic}} = N_{SW} \times C_{Load} \times V_{CC}^2 \times f
\]  

(6.3)

6.1.2 Network on Chip Power Model

In many-core systems, apart from the core’s power consumption, a substantial amount of power is consumed by the interconnection network. Therefore considering the NoC power consumption is of high importance when discussing power management in many-core systems.

Different power models have been proposed by researchers. However, the basic components of NoC power consumption is the same in all of them. The total consumed power is the summation of power consumed by routers at each hop and power dissipated by links or wires. Following is the energy model suggested by [30] for average package traversal energy:

\[
E_{\text{pkt}} = H_{\text{Avg}} \cdot (E_{\text{Queue}} + E_{\text{SF}} + E_{\text{ARB}}) + L_{\text{Avg}} \cdot E_{\text{Link}} + E_{\text{Queue}}
\]  

(6.4)

In above equation, \( H_{\text{Avg}} \) and \( L_{\text{Avg}} \) are the average number of hops and average distance between the source and the destination. In a switching hop, the power is consumed by an input queuing buffer \( E_{\text{Queue}} \), switching fabric \( E_{\text{SF}} \) and arbitration logic \( E_{\text{ARB}} \). \( E_{\text{Link}} \) is the energy consumed to transmit packet over a unit of link which is determined by the fabrication technology.

6.2 Processor Power Management Techniques

6.2.1 Gating

Clock gating and power gating are one the most common techniques in reducing the power consumption of processors. In gating techniques, clock and power supply of the system are gated off resulting in less power consumption.

Clock Gating

Clock of a system is responsible for a vast amount of power consumption in a CPU. An effective technique to reduce this substantial power consumption is clock gating. In clock gating, the clock of a unit in the system which is left unused, could be turned off. By doing so, the unit’s transistors are prevented from switching, leading to zero dynamic power consumption. The only power dissipation source would be the leakage current.
6.3. POWER MANAGEMENT IN MANY-CORE REALTIME SYSTEMS

Power Gating

Power gating is one of the techniques to conquer static power consumption. In power gating, the supply voltage of unused units is cut off to prevent the power dissipation due to subthreshold leakage current.

In many-core systems, gating techniques can be exploited to gate the power or clock of one or multiple cores. An important issue which must be considered while using gating techniques, is the overhead imposed by such solutions. The delay of re-enabling a unit could substantially affect the performance of the system.

6.2.2 Dynamic Voltage and Frequency Scaling

DVFS is a power saving technique that scales the voltage and frequency of a processor/core according to the workload. In order to apply DVFS voltage regulators are needed. Off-chip regulators are too slow and do not allow fast adjustment of voltages, therefore on-chip voltage regulators are used to provide flexible voltage scaling.

In many-cores, DVFS could be applied on three levels, Per-chip DVFS, Per-core DVFS and cluster DVFS. Per-chip DVFS systems use a single regulator which scales all cores in the same manner. It is a rigid architecture that does not allow flexible voltage adjustment, therefore it limits the power saving that could be achieved. Per-core DVFS systems use a regulator for each core, enabling to control each core individually. Such a design would impose power and area overhead on the system, because of their large number of on-chip regulators. A cluster DVFS system proposed by [27] suggests an intermediate solution by grouping the cores into clusters and considering a regulator for each cluster of cores.

6.3 Power Management in Many-core Realtime Systems

Power management can be done on different levels such as hardware level, system level and application level. Many researches have tackled the power consumption issues from different angles such as considering the power consumption of memory transactions and contentions, working on hot spots and some researches have proposed using different variations of DVFS to reduce power consumption in many-core systems.

Power management in many-core real-time systems is tightly entangled with the scheduling policy which is deployed. Power consumption depends on how the tasks are allocated to cores. Task allocation affects the inter process communication, load balancing and number of cache misses, consequently affects the power consumption. Therefore power management should be done in combined with scheduling of the system.
6.4 Summary of Power Management in Many-cores

This chapter answers the second and third questions raised in the introduction chapter:

What power management techniques are exploited by the many-core systems?
How does power consumption relate to scheduling algorithm?

In this chapter, the processor and the network on chip power model in a many-core system is presented. Different hardware level power saving methods are introduced.

The power consumption of the system is directly affected by the task allocation and the inter-process communication of the system, therefore the power management should be done along with the scheduling of the system.
Chapter 7

Specification for System Design

In this chapter, it is explained which of the three previously introduced hardware platforms is selected for the implementation of a power-aware real-time scheduler. Further on the design of the scheduler is elaborated.

7.1 Hardware Platform

After studying three different many-core systems, Adapteva Parallella is selected as the platform for implementing a power-aware real-time scheduler. Parallella is selected for the following reasons:

Bare Metal

Epiphany is a bare metal environment providing direct access to the hardware. Therefore scheduler has full control on how the system is running. Using bare metal environment leads to more predictability in a real-time system. Operating systems often make it complicated to calculate the worst case execution time and also it is more difficult to make sure that the deadlines are met.

The scheduler is placed on epiphany, working almost independently from the ARM processor. By using this approach, the slow communication between Epiphany and ARM processor could be avoided.

Simple Memory Organization

Epiphany has a simpler memory organization, compared to TilePro64 or MPPA-256 which have two or three levels of caches. A simple memory organization increases the system’s predictability since the worst case execution time of tasks could be calculated easier and more precisely.
7.2 Task Model

The scheduler will execute periodic tasks, including both aperiodic and sporadic tasks. The task model considered for the system is similar to the task model introduced in section 4.1.1. A task $\tau_i$ is defined by its relative deadline $D_i$, a worst case execution time $C_i$ and a period $T_i$. The utilization $u_i$ of task $\tau_i$ is $\frac{C_i}{T_i}$ and the utilization of a taskset is $u_{taskset}$ which is the sum of the utilizations of all tasks included in a taskset. The tasks are considered to be independent and no inter-communication is required among tasks.

The scheduler will be designed to work with tasks that follow the above task model. Tasks are considered to have implicit deadlines, meaning that the deadline of a task is the same as its period. Tasks are all released as soon as they are loaded into the system.

7.3 Scheduling Policy

As previously mentioned, fully partitioned algorithms suffer from low utilization bounds and global algorithms are not implementable due to their excessive overhead because of high frequency of preemption, context switch and migrations. Therefore a semi-partitioned approach is suggested to be used as the scheduling algorithm. As it can be seen in figure 7.1 taken from [29], the class of semi-partitioned algorithms which use EDF has the highest utilization bound with 66%.
The scheduling policy proposed for the design is similar to a semi-partitioned scheduling algorithm with task splitting called EDF with Window-constraint Migration (EDF-WM) designed by Kato et al [25].

In EDF-WM tasks are statically assigned to a core. These tasks will not migrate and are fixed on each core. A task is only allowed to migrate if there is no core that the whole task can fit in. Then the migratory task is split and it will be executed over more than one processor.

One of the differences between the proposed algorithm and original EDF-WM is the task partitioning method. In EDF-WM, tasks are allocated to cores according to first-fit heuristic. However, in this design, cores are filled with tasks in an ordered manner. Each core is filled with tasks unless there is no available task which can fit in the remaining capacity of the core.

The idea is to use as few cores as possible to keep the energy consumption at minimum level. After determining the required number of cores, the remaining Epiphany cores are put to sleep. In case of arrival of a sporadic task, if the task
does not fit in the currently in-use cores, a new core is woken up to execute the task. After completion of the task, the awakened core is put back to sleep.

7.4 Scheduler

The scheduler is consisted of two main parts, partitioner and local schedulers. The partitioner runs on the host computer and local schedulers run on Epiphany cores.

7.4.1 Partitioner

The partitioner distributes tasks among cores. Each core is filled with tasks until there is no task in the global queue which can fit in the core. A task can fit in a core if the utilization of the task (the FORMULA) is less than the free space of that core. Cores, one after one, are filled with tasks until all tasks are assigned to cores. Unused cores can be put to sleep.

After distributing tasks among cores, the partitioner will select the core which is utilized the least. The selected core will be checked to see if it is possible to remove the tasks assigned to it and split them over other active processors. To be able to this, other active cores must have enough free space to fit portions of tasks which are assigned to the selected core. If such free space exists and all tasks of the selected core can be moved to other active cores, the selected core can be put to sleep. However if such free space is not available, no task splitting would take place. By splitting the tasks, the utilization of active cores is improved and also it is possible to spare one or more cores and therefore reduce the power consumption. After determining tasks which are supposed to split, new worst case execution time, new period, new deadline and new release time is calculated for portions of the split tasks. Cores which execute a split task, receive different task descriptions for the split task. figure and explain window constrained migration.

Figure 7.2 demonstrates how task splitting is done. The figure on the left shows the state of the system after distributing tasks among cores and figure on the right shows the state of the system after task splitting phase is done. As it can be seen, the task on the last active core is removed from the core, is split and runs on cores number 1 and 3.
Further on, 16 lists of tasks are passed to Epiphany for execution. Each core stores tasks assigned to it in a local run queue.

In case of arrival of a new task, the partitioner will add the new task to the global task queue. Three scenarios are possible to occur:

A *There is a core which has sufficient space to fit the whole task.*

B *There is no core with sufficient space, but there is enough aggregated space among active cores to fit the new task. In this case, the task is split and it will run over multiple cores.*

C *There is no core with sufficient space and there is no enough aggregated space among active cores either. In this case a core which is in sleep mode is activated and the new task is assigned to it. Now that another core has been added to the system and the capacity of the system has increased, the scheduler will try to utilize the new core to transform the split tasks back to normal tasks and therefore eliminate unnecessary task migration.*

Figure 7.3 demonstrates three different scenarios. Figures on left show state of the system when a new task arrives and figures on right show state of the system after adding the new task.
Figure 7.3. Different Scenarios when a new task arrives
7.4.2 Local Scheduler

The local scheduler is an EDF scheduler running on each core. The local scheduler receives the task list from the partitioner and executes those tasks.

In pre-scheduling phase, the local scheduler computes the hyper period of the taskset. Hyper period of a taskset is the least common multiple of periods of tasks in that taskset. The order of task executions is the same in different hyper periods if the taskset has not changed.

After the pre-scheduling phase, the scheduler begins executing tasks according to earliest deadline first policy. In EDF, the task which has the earliest deadline is selected to execute. The scheduler periodically interrupts execution to check if there is a task which has an earlier deadline. If so, the current task is preempted and the task with earlier deadline takes over the core. Figure 7.4 shows an example of scheduling three tasks using EDF scheduling policy. The red lines indicate period and deadline for each task and the orange arrows indicate when a preemption occurs.

![Figure 7.4. An example of scheduling tasks using EDF](image)

7.5 Summary of the Specification for System Design

In this chapter, various reasons why Parallella has been chosen as the hardware platform, is explained. Further on, a design has been proposed for the power-aware real-time scheduler. The proposed design is based on the EDF-WM algorithm and is consisted of a partitioner and local EDF schedulers. The scheduler aims to minimize the number of active cores, required to meet hard real-time deadlines.
Chapter 8

Implementation: A Power-aware Real-time Scheduler for Parallella

This chapter describes the implementation of the power-aware real-time scheduler. The implemented scheduler is as described in previous chapter, but it does not have the task migration functionality. Therefore it is a fully partitioned scheduler rather than a semi-partitioned scheduler. Further on in this chapter, the overhead of a general EDF scheduler is formulated and the overhead of the implemented scheduler is calculated.

8.1 Structure of the Scheduler

The hardware platform used to implement the scheduler is an Adapteva Parallella prototype board. As of this moment, the final product which is described in chapter 2, has not yet been released. This is why the prototype system is used instead. The prototype system includes a desktop Linux machine and an Altera Stratix development board. The Epiphany chip is mounted on a cross shaped board which is connected to the Altera Stratix. The Altera board plays the role of a bridge to connect Epiphany board to the host computer, the Linux machine. The desktop Linux machine plays the role of the ARM processor in the final Parallella product.

As previously mentioned, the scheduler is composed of two main components, partitioner and local schedulers. The partitioner runs on the host machine and local schedulers run on Epiphany cores.

Real-time tasks are implemented as functions and they are called by the scheduler whenever they are supposed to run. Tasks are independent.

8.1.1 Host application

The host application is the interface of the scheduler to the outside world. Each task has a task description which includes the tasks worst case execution time, deadline,
period and release time. Tasks descriptions are provided to the host application as the input of the scheduler.

**Partitioner**

Host application starts by connecting to the Epiphany board. When connection is established, the host application loads the tasks descriptions and makes a linked list of them to form a global queue. Further on it sorts the list according to tasks deadlines in a non increasing order. As stated in [25] sorting the tasklist in this manner increases the processors utilization.

After making the list(global queue) and sorting it, the partitioner assigns tasks to cores. To fill each core, the whole global queue is traversed to find tasks that can fit the core. Figure 8.1 shows the pseudo code of the partitioner.

```plaintext
for ( i from 0 to NUMBER_OF_CORES) {
    while (!end_of_list) {
        if(task has not been already assigned to a core) {
            if(the core has sufficient space to fit the task) {
                Add the task to the core's local run queue;
                Mark task as assigned to a core;
            }
        }
    }
}
```

*Figure 8.1. Partitioner pseudo code*

**Splitter**

The host application can recognize tasks that could be split and it calculates the necessary information needed for task migration such as new period, deadline and release time for task portions. However, task migration is not supported on local schedulers on Epiphanny cores. This is one of the future works that could be done regarding this thesis.

**8.1.2 Communication between Host and Epiphany**

When the host application has completed the calculations and the tasks assigned to each core are determined, this information should be passed to Epiphanny cores and local schedulers. A common buffer is used as the communication means. The common buffer is allocated on the shared DRAM which both host and Epiphanny
has access to. The host writes tasks that each core must execute to the common buffer. When this is done, the host sends a synchronization signal to the Epiphany indicating that Epiphany shall start scheduling. The host uses e read and e write to access the shared DRAM.

8.1.3 Local Scheduler

As previously mentioned, local scheduler runs on each core. Main components of the local scheduler are hardware timer, pre-scheduler, EDF scheduler and context switcher.

Each core keeps polling the common buffer, waiting for the synchronization signal. When the signal is received, each core loads the tasks that are assigned to it and stores them in it’s local run queue. After forming the local run queue, the pre-scheduler calculates the hyper period of the taskset. It also saves the period and deadline of each task in a buffer to be able to restore them later when a new hyper period begins. The period and deadline are saved due to the fact that whenever the scheduler invokes a task, it updates the period and deadline of the task.

At this point, the system is ready to schedule and execute the tasks.

EDF Scheduler

Each task in the system can have one of the following three execution states:

- Task has not yet been executed in it’s period. It must be executed before reaching it’s deadline.

- Task has been executed in it’s period, however it was preempted by a higher priority task and could not complete it’s execution. It must finish it’s execution before reaching it’s deadline.

- Task has been fully executed in it’s period and must not run until it’s next period

Every time the EDF scheduler is invoked, it searches the local run queue to find a task with the execution state of 1 or 2, which has the earliest deadline. Depending on the execution state, different actions are taken. If the task is in the execution state of 1, the timer is set and the task is invoked. If the task is in the execution state of 2, the context of the task is restored, the timer is set and the task resumes execution from where it had left. In case the scheduler cannot find a ready task to execute, meaning that all tasks are in execution state of 3, it sets the timer and waits in a loop until a task is ready to run. Figure x.x shows the flowcharts of the local EDF scheduler on the left and timer interrupt routine on the left.
Timer

An EDF scheduler needs to check the status of the system periodically, to see if a task has an earlier deadline than the task which is currently running. If there is such a task, the scheduler should preempt the current task and let the task with earlier deadline execute. To implement this periodic checks, the hardware timer is used. Whenever a task is invoked, the timer starts running. When the timer expires an interrupt is fired.

The interrupt handler routine first checks to see if the hyper period is over or not. If the hyper period is over, it resets the tasks execution states to 1 and also resets the tasks original periods and deadlines. Then it checks to see if there is a task with earlier deadline than the currently executing task. If there is no such a task, the currently running task resumes execution. However if there is such a task, the context of the current task is saved and the scheduler is invoked to switch the tasks and execute the task with the earlier deadline.
Context Switch

When the timer interrupt fires, the status of the system is checked. In the case that the currently running task is preempted because of a higher priority task, the context of the current task must be saved at the moment of preemption, in order to be able to resume its execution later on. The context of a task includes the contents of the registers and its stack. To save these information, a dedicated space is considered for each task.

When writing an interrupt handler function, the attribute 'interrupt' is used in the definition of the function. GCC notes this attribute and generates special prolog and epilog to the function. This compiler generated code, stores all the registers of the system on the stack, before entering the interrupt routine and restores all the registers from stack, while exiting the interrupt routine. This way, the context of the task which was running when the interrupt occurred, is saved and the behavior of the task remains intact. Figure 8.2 shows the prolog generated by the compiler right before entering the interrupt handler routine.

![Prolog generated by the compiler for the interrupt handler routine](image)

**Figure 8.3.** The prolog generated by the compiler for the interrupt handler routine

This feature of the interrupt handler routine is used to implement the context switch. By studying the assembly code generated by the compiler, it is possible to
know in which addresses of the stack, the registers are saved. Content of those stack addresses are the content of the registers at the moment that interrupted occurred. Therefore the content of those stack addresses are saved in the dedicated space which is considered for the task to save it’s context.

When an already preempted task is supposed to resume it’s execution, the content of the task is read from the task’s context switch space and is loaded into the registers. The same process applies to the store and restore of the task’s stack.

Figure 8.3 shows part of the extended assembly code written to save the content of registers in the task’s context switch space and Figure 8.4 shows part of the extended assembly code written to read the task’s context switch space and load them into the registers.

```assembly
asm volatile ("ldrdr r0, [sp, +0x11]");
asm volatile ("mov %[temp], r0": [temp] "=r" [context_switch_buf regs[10]]);
asm volatile ("mov %[temp], r1": [temp] "=r" [context_switch_buf regs[11]]);
asm volatile ("ldrdr r0, [sp, +0x10]");
asm volatile ("mov %[temp], r0": [temp] "=r" [context_switch_buf regs[12]]);
asm volatile ("mov %[temp], r1": [temp] "=r" [context_switch_buf regs[13]]);
asm volatile ("ldrdr r0, [sp, +0xf]"[no output]");
asm volatile ("mov %[temp], r0": [temp] "=r" [context_switch_buf regs[14]]);
asm volatile ("mov %[temp], r1": [temp] "=r" [context_switch_buf regs[15]]);
```

**Figure 8.4.** Snippet of extended assembly code used in context store process

```assembly
asm volatile ("mov r18, %[adr]": [no output]: [adr] "=r" [context_switch_buf regs[18]]);
asm volatile ("mov r19, %[adr]": [no output]: [adr] "=r" [context_switch_buf regs[19]]);
asm volatile ("mov r20, %[adr]": [no output]: [adr] "=r" [context_switch_buf regs[20]]);
asm volatile ("mov r21, %[adr]": [no output]: [adr] "=r" [context_switch_buf regs[21]]);
asm volatile ("mov r22, %[adr]": [no output]: [adr] "=r" [context_switch_buf regs[22]]);
asm volatile ("mov r23, %[adr]": [no output]: [adr] "=r" [context_switch_buf regs[23]]);
```

**Figure 8.5.** Snippet of extended assembly code used in context restore process

### 8.2 Scheduler’s overhead

In a real-time system, regardless of the deployed scheduling policy, part of the CPU utilization is consumed by the scheduler. The overhead of the scheduler must be measured and considered while designing a real-time system. By knowing how much of the CPU utilization is consumed by the scheduler, it is possible to consider this overhead in the system and have a hard real-time scheduler in which all tasks meet their deadlines. In this section, first a general formula is provided to calculate the overhead of an EDF scheduler and then the result of measurements done on the implemented scheduler is presented.
8.2. SCHEDULER’S OVERHEAD

8.2.1 EDF scheduler overhead calculation

An EDF scheduler is invoked when one of the followings occur:

- Resume the current task: timer interrupt has fired and the scheduler checks the state of the system. There is no higher priority task to take over, so the current task shall resume.

- Preemption and context switch: timer interrupt has fired and the scheduler checks the state of the system. There is a higher priority task to take over, so the current task must be preempted and the higher priority task shall take over the core.

- Selecting a task: task has finished execution. A new task shall be selected to run.

Depending on which of the above scenarios occur each time, the overhead of the scheduler is different. The overhead of the scheduler should be calculated for all three scenarios.

In the following, a taskset is considered to have \( m \) tasks, which each task has a period of \( P_i \) (\( 1 < i < m \)) and hyper period of \( HP \).

Every time that the timer interrupt fires the scheduler is invoked. Considering the described taskset, in a hyper period, the timer interrupt fires \( HP/P_{\text{timer}} \) times and invokes the scheduler. Among these \( HP/P_{\text{timer}} \) times that scheduler is invoked, sometimes the schedulers resumes the current tasks and some time the task is preempted and a new task is set to execute. The number of preemptions in a hyper period depends on the taskset and the periods and deadlines of the tasks within the taskset. The execution time consumed by the scheduler when a timer interrupt is fired is calculated by the following formula:

\[
[(HP/P_{\text{timer}} - N_{\text{preemption}}) \times E_{\text{resume\_task}}] + \lfloor N_{\text{preemption}} \times E_{\text{preemption}} \rfloor
\]  

(8.1)

In formula 8.1, \( N_{\text{preemption}} \) is the number of preemptions in a taskset, \( E_{\text{resume\_task}} \) is the execution time consumed by the scheduler when the timer interrupt fires and the current task resumes execution and \( E_{\text{preemption}} \) is the execution time consumed by the scheduler to preempt the current task and set a higher priority task to execute.

Whenever a task finishes execution, a new task shall be selected to execute. In a hyper period, task \( i \) executes \( HP/P_i \) times. It means that in a hyper period, the scheduler is invoked \( HP/P_i \) times for each task \( i \) to select a new task. The execution time consumed by the the scheduler to select a new task in a hyper period is calculated by:

\[
\sum_{i=1}^{m} HP/P_i \times E_{\text{select\_a\_task}}
\]  

(8.2)
In formula 8.2, $E_{\text{selectata}}$ is the execution time of the scheduler to select and run a task.

Considering formulas 8.1 and 8.2, the overall execution time of the scheduler, $E_{\text{Scheduler}}$ in a hyper period is calculated by formula 8.3:

$$E_{\text{Scheduler}} = [(HP/P_{\text{timer}} - N_{\text{preemption}}) \times E_{\text{resume_task}}]$$

$$+ [N_{\text{preemption}} \times E_{\text{preemption}}] + \sum_{i=1}^{m} HP/P_i \times E_{\text{select_a_task}}$$

(8.3)

Formula 8.3 calculates the execution time consumed by an EDF scheduler in a hyper period. This execution time must be considered while designing the system and evaluating the feasibility of a taskset.

### 8.2.2 Overhead of the implemented scheduler

Using the formula 8.3, the execution time of the implemented scheduler could be calculated. To calculate the scheduler’s execution time in a hyper period, the values of $E_{\text{resume_task}}$, $E_{\text{preemption}}$ and $E_{\text{selectata}}$ should be measured for the implemented local scheduler. The hardware timer available on each core is used to count the number of clock cycles elapsed in each part of the scheduler. Having the number of the clock cycles and the clock frequency which is 600 MHz, the execution time of each part of the scheduler is calculated. For the implemented scheduler, the measured values of $E_{\text{resume_task}}$, $E_{\text{preemption}}$ and $E_{\text{selectata}}$ are 128 micro seconds, 374 micro seconds and 300 micro seconds, respectively. Using these numbers in formula 8.3, the formula 8.4 is achieved which calculates the overall execution time for the implemented scheduler.

$$E_{\text{Scheduler}} = [(HP/P_{\text{timer}} - N_{\text{preemption}}) \times 128] + [N_{\text{preemption}} \times 374] + \sum_{i=1}^{m} HP/P_i \times 300$$

(8.4)

As it can be seen in formula 8.4, the execution time of the scheduler and therefore the overhead of the scheduler is directly affected by number of preemptions and period of the timer. Although the larger values for period of timer reduces the overhead of the scheduler, it decrease the precision of the scheduler. This trade off should be considered while designing the system and a reasonable value should be selected for the timer.

### 8.3 Summary of the Implementation

In this chapter, different components of the scheduler are described. The scheduler is consisted of a host application and local schedulers. Host application runs on a Linux desktop machine. This application is the interface between the scheduler
and the outside world and it is in charge of partitioning the tasks based on their descriptions. Local EDF schedulers run on Epiphany cores.

Further on, the overhead of the local scheduler is calculated by measuring it’s execution time using hardware timers. The overhead of the scheduler must be considered while designing the system to have a hard real-time scheduler.
Chapter 9

Power Measurement of Epiphany

In this chapter, the power measurement of Epiphany chip is described, explaining how the power consumption is measured as well as providing results.

9.1 Power Measurement Method

The power consumption of a circuit at a moment is calculated by the following formula:

\[ P = V \times I \]  

(9.1)

So to be able to measure the power consumption of Epiphany chip, the voltage that it is provided with and also the current that flows through it, are required.

On the cross shaped board on which Epiphany is located, there are several test points to monitor the status of different parts of the board. Three of these test points are used to measure the voltage and current of the Epiphany. TP5 is the ground of the board. TP14 is ’VCORE IN’. The value read at this test point is the voltage that the Epiphany chip is provided with. The value of this test point is 1.089 v. TP13 is ’VCORE(A)’. The value read at this test point is the value of a current sensor resistor. The current sensor resistor measures the current flowing in the chip and translates it to voltage.

Considering the mentioned test points and the general formula of power consumption, the power consumption of the Epiphany chip is calculated by the following formula:

\[ P_{Epiphany} = VCOREIN \times VCORE(A) \]  

(9.2)

By using a voltmeter, it is possible to read these test points and calculate the power consumption of Epiphany chip at different modes for different applications.

9.2 Power Measurement Results

Results of experimenting different codes and scenarios show that the two major factors that affect power consumption of Epiphany are floating point unit and memory.
To measure Epiphany’s power consumption, a small code is used that involves the floating point unit. Figure 9.1 shows the code running on cores while measuring the power consumption.

9.2.1 Memory management effects

Epiphany software development kit includes three linker files, Legacy, Fast and Internal. The below table taken from [2] summarizes three linker files and different memory management scenarios.

Power consumption of Epiphany is measured using the mentioned code and different linker files. Figure 9.2 shows the result of the measurement. The gray bars show the execution time of the code using different linker files. The yellow line shows the power consumption of Epiphany when all the cores are in idle mode. The red line shows the power consumption of Epiphany when all cores are actively running.

As it can be seen in the figure, when using Legacy linker file, the power consumption is very close to idle mode. The reason is that all the code, data, standard library, stack and heap are placed on external memory and the internal memory is not engaged. However the execution time is 2.868 seconds which is more than the scenario that uses Fast linker file and much more than the scenario that uses Internal linker file. On the other hand, when using the Internal linker file, the power consumption is almost twice of the Idle mode while the execution time is only 0.004 seconds. Internal linker file places all the code, data, standard library, heap and stack on the internal memory of each core which results in more power consumption and less execution time.
### 9.2. POWER MEASUREMENT RESULTS

#### Table 9.1. Memory management [2]

<table>
<thead>
<tr>
<th>File</th>
<th>USER CODE and DATA</th>
<th>STANDARD LIBRARY</th>
<th>STACK</th>
<th>NOTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>legacy.ldf</td>
<td>External SDRAM</td>
<td>External SDRAM</td>
<td>External SDRAM</td>
<td>Use to run any legacy code with up to 1MB of combined code and data.</td>
</tr>
<tr>
<td>fast.ldf</td>
<td>Internal SRAM</td>
<td>External SDRAM</td>
<td>Internal SRAM</td>
<td>Places all user code and static data in local memory, including the stack. Use to implement fast critical functions. It is the user’s responsibility to ensure that the code fits within the local memory.</td>
</tr>
<tr>
<td>internal.ldf</td>
<td>Internal SRAM</td>
<td>Internal SRAM</td>
<td>Internal SRAM</td>
<td>Places all the code and data in local memory, including the stack. Use to implement fastest applications. It is the user’s responsibility to ensure that the code fits within the local memory.</td>
</tr>
</tbody>
</table>

The execution time of the code is measured by hardware timers that are available on each core. The value of timers of all 16 cores are retrieved after execution and average of the 16 timer values is calculated. This process is carried out 5 times for each linker file and the result is used in figure 9.2.

#### 9.2.2 Power consumption of each core

In the second part of power measurements, two pieces of code are executed on Epiphany. The first code is exactly the same code used in previous measurements that engages floating point unit. The second code is the same code but the floating point numbers are replaced with integer numbers, therefore the floating point unit is not engaged. The Internal linker file is used to maximize Epiphany’s power consumption. The power consumption of Epiphany is measured 17 times for each code. At first, all cores are in idle mode and then they are activated one by one. Figure 9.3 shows the result of this measurement.
Figure 9.2. Power consumption vs Execution time

Figure 9.3. Power consumption of Epiphany cores with and without floating point unit
Each core averagely consumes 22 mili watts with floating point unit engaged and 16 mili watts without floating point unit. If the scheduler is able to put a core in idle mode, in the worst case scenario it can save at least 3.1% of the total power consumption. The worst case scenario is when all 16 cores are activated and only one core can be turned off. If the number of activated cores is smaller than 16, the percentage of power saving grows.

9.3 Summary of Power Measurement

In this chapter, the method used to measure the Epiphany’s power consumption is described. Further on, Epiphany’s power consumption is measured for a small application which involves floating point unit. The results of this measurement illustrate that memory management has great effect on the power consumption. Placing all the code, data, standard libraries, heap and stack in the internal memory increases power consumption vastly.

Regarding the implemented scheduler, if some of the code, data, standard libraries, heap and stack are placed in external memory, it would improve the power consumption. On the other hand it should be considered that by doing so, the execution time would suffer. Therefore, care should be taken that reducing power consumption results in increased execution time and this may lead to deadline misses. By profiling the system and testing different scenarios, a balance could be reached regarding this trade off.

Another experiment measures the power consumption of each core with and without engaging floating point unit. The results show that the scheduler could save at least 3.1% of total power consumption of Epiphany’s chip.
Chapter 10

Conclusion and Future Work

10.1 Conclusion on the Project

The thesis covers both academic study and design and implementation of a power-aware hard real-time scheduler with ability to handle sporadic tasks, on Parallellela platform. The thesis proposes a specification based on academic study, considering different aspects of the design of such scheduler.

The implementation part of the thesis aims the proposed specification and moves towards it. The implemented scheduler completes the basic requirements of the scheduler including meeting the hard real-time deadlines and conserving power when possible. It provides an infrastructure for further developments and improvements. The developed scheduler is capable of meeting the hard deadlines and also save on power, whenever possible.

10.2 Limitations

At beginning of the project, some of the limitations of the system were already known. During the implementation of the scheduler, some new limitations came into play. Important limitations of the system are listed below:

- The scheduler does not support task migration on Epiphany level, therefore the design is fully partitioned rather than semi-partitioned.
- There is no inter-dependency among tasks and tasks are independent and isolated.
- The scheduler takes the worst case of the execution of tasks as input. It does not measure the worst case execution time.
- Tasks are implemented as functions which are called by the scheduler.
- Tasks are loaded statically.
• The system does not detect if tasks overrun their execution time.

• Local scheduler is partially working on shared DRAM, which is slower than the internal memory.

10.3 Future Works

According to the above limitations, some future works are proposed that would improve the quality and functionality of the scheduler.

• Implement inter-core task migration, meaning that task could be preempted on a core and resumed on another core. This would require transferring the task’s context from the first to the second core.

• Extend the design to be able to manage tasks with inter-dependencies.

• Develop a task profiling application which can measure the worst case execution time of the tasks, when they are loaded.

• Develop a task loader for the scheduler.

• Decouple the tasks and scheduler completely, so that tasks are not called by scheduler, they should be invoked by the scheduler.

• Develop a task monitoring application which can check the status of tasks and detect it in case they overrun their execution time.

• Improve the execution time of the scheduler in different modes so that it would impose less overhead on the system.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


