Performance and energy-aware real-time scheduling for heterogeneous embedded systems

Francisco João Feliciano Gaspar

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Doctor Leonel Augusto Pires Seabra de Sousa
Doctor Aleksandar Ilic

Examination Committee
Chairperson: Doctor João Manuel Torres Calhúns Simões Vaz
Supervisor: Doctor Leonel Augusto Pires Seabra de Sousa
Members of the Committee: Doctor José Carlos Campos Costa

October 2014
Acknowledgments

I would like to thank my supervisors, Doutor Leonel Sousa, Doutor Aleksandar Ilic and Doutor Pedro Tomás, for their guidance and advice throughout the development of this thesis as well as for all the priceless revisions of the final work. I would also like to thank Luís Taniça for his help with the integration of the Beeps API and driver on the framework. Furthermore I deeply appreciate all the support I got from family and friends over this period. Finally, I would like to thank IST and INESC-ID for all the resources made available, without which this thesis could not have been completed.

This work was supported by national funds through FCT – Fundação para a Ciência e a Tecnologia, under projects PTDC/EEI-ELC/3152/2012 and PTDC/EEA-ELC/117329/2010.
Abstract

With an ever growing demand for energy efficient systems, heterogeneity has found its way into processors through the combination of high performance and energy efficient cores. The scheduling of tasks on these systems presents a challenge since it is desirable to achieve a good performance while saving power. Therefore, this thesis aims to analyse current scheduling techniques applicable to heterogeneous systems, and to propose a new task management topology that can optimize overall performance and efficiency by maintaining performance fairness among running tasks while limiting energy usage by only allocating the resources needed for the applications to achieve all their performance targets. To achieve this goal, a framework is proposed that gathers data from Quality of Service programs (integrated with a real-time performance reporting Application Programming Interfaces). It achieves task level share control, to ensure task performance fairness, dynamic frequency control (reduces the amount of available system resources which saves energy) and migration controller, enabling the control of single- and multi-threaded applications simultaneously.

The proposed performance and energy aware, real-time scheduling, for heterogeneous embedded systems framework, achieves energy savings of up to 65% while improving the performance error in up to 122x. These results were achieved by comparing the proposed framework with the default system tools on an Odroid-XU+E development board (featuring a big.LITTLE powered system on a chip) with applications selected from the PARSEC benchmark suite that were modified to report their performance.

Keywords

Heterogeneous multi processor; scheduling; embedded systems; quality of service; big.LITTLE; task migration; dynamic frequency control
Resumo

Com o aumento crescente da procura de sistemas energicamente eficientes, os processadores começam a apresentar soluções heterogéneas combinando núcleos de processamento de alto desempenho com núcleos eficientes de baixo consumo de energia. O agendamento de tarefas nestes sistemas constitui um desafio, de forma a garantir um bom desempenho com reduzido consumo energético. Assim, esta tese pretende analisar técnicas actuais de agendamento dinâmico de tarefas aplicáveis a sistemas heterogéneos, para propor um novo método capaz de optimizar o desempenho global do sistema bem como a sua eficiência. Estes objectivos são alcançáveis com uma atribuição justa (do ponto de vista do desempenho) do tempo do processador e da redução dos recursos atribuídos para os mínimos necessários para os atingir. Para isso, os programas deverão reportar o seu desempenho através de uma API comum, encarregue de monitorizar o desempenho dos programas em tempo real, cujos dados são processados por um controlador composto por 3 módulos: controlador de tempo (que garante uma divisão justa de performance), controlador de migração (que permite o controlo de aplicações com qualquer número de tarefas) e um controlador de frequência (que limita os recursos disponíveis poupando energia).

A implementação do controlador de agendamento de tarefas proposto para sistemas heterogéneos em tempo real provou a viabilidade da ferramenta desenvolvida, reduzindo a energia consumida até 65% e reduzindo o erro relativo do desempenho até 122x. Estes resultados foram obtidos numa placa de desenvolvimento Odroid-XU+E, que contém uma implementação da arquitectura big.LITTLE, através da comparação de resultados da execução de programas selecionados do conjunto de testes PARSEC com os resultados obtidos pelos mesmos programas quando executados com as ferramentas padrão do sistema.

Palavras Chave

Processadores heterogeneos; agendamento transparente; sistemas embebidos; qualidade de serviço; big.LITTLE; gestão de tarefas; controlo dinâmico de frequência
## 1 Introduction

1.1 Motivation .................................................. 3
1.2 Objectives .................................................. 4
1.3 Main contributions ........................................... 5
1.4 Outline ...................................................... 5

## 2 Embedded Heterogeneous Platforms

2.1 ARM big.LITTLE Architecture ............................... 8
   2.1.1 ARMv7-A Instruction Set Architecture (ISA) ........ 9
   2.1.2 Cortex-A15 and Cortex-A7 Microarchitecture ........ 10
   2.1.3 Performance Monitor Unit (PMU) ...................... 11
2.2 Task migration techniques .................................. 11
   2.2.1 big.LITTLE Task Migration Model .................... 11
   2.2.2 big.LITTLE MP ........................................ 13
2.3 Available development boards ............................. 13
2.4 Summary .................................................... 14

## 3 Scheduling and Dynamic Voltage and Frequency Scaling (DVFS)

3.1 Processor demand scheduling ............................... 18
3.2 Quality of Service (QoS) aware scheduling ................ 20
3.3 Current default Linux scheduler - CFS .................... 21
   3.3.1 Priorities and nice ................................... 23
   3.3.2 Migration across cores ............................... 24
3.4 Dynamic Voltage and Frequency Scaling (DVFS) .......... 25
3.5 Summary .................................................... 25

## 4 Framework for the performance and energy-aware real-time scheduling for heterogeneous embedded systems

4.1 Framework general overview ............................... 28
4.2 Capturing system/application interaction: applications monitoring .................................................. 29
   4.2.1 Performance counters for application characterization .................................................. 30
   4.2.2 Applications that report their performance .................................................. 30
4.3 Performance monitoring Application Performance Interface .......................... 32
4.4 Controller ................................................................. 36
  4.4.1 Share controller ................................................... 37
  4.4.2 Frequency controller ............................................. 40
  4.4.3 Migration controller ............................................ 43
4.5 Summary ................................................................. 45

5 Proof of concept and experimental Evaluation 47
  5.1 System behaviour simulation ....................................... 48
  5.2 Controller's triggering ............................................. 52
  5.3 Benchmarks ............................................................ 54
  5.4 Experimental setup ................................................. 57
    5.4.1 Characterizing the Platform: Cortex-A7 vs Cortex-A15 .......................... 58
  5.5 Experimental results ............................................... 61
    5.5.1 Evaluation of application-wide Share and Frequency controller .............. 62
    5.5.2 Evaluation of the proposed full framework for performance and energy-aware real-time scheduling for heterogeneous embedded systems ......................... 65
  5.6 Summary ............................................................... 69

6 Conclusions ............................................................... 75
  6.1 Future work ............................................................ 77

A Countable events on the ARMv7-A ISA .............................................. 83

B Cycles to complete memory micro-benchmark on the Cortex-A7 and Cortex-A15 87
# List of Figures

2.1 Diagram of a possible big.LITTLE system using the CCI-400 and 64-bit cores ................................. 9

2.2 Pipeline diagram of the Cortex-A7 .................................................................................................. 10

2.3 Pipeline diagram of the Cortex-A15 ............................................................................................... 10

2.4 Steps to be taken while performing a task migration on the Task Migration Model .................... 12

3.1 Control loop used on a state of the art QoS aware controller ......................................................... 21

3.2 Red-black tree example ................................................................................................................. 22

3.3 A graphical interpretation of the basic scheduler functionality ....................................................... 23

3.4 An example of a thread combination where the CFS load balancing would not achieve load fairness ............................................................................................................................................. 25

4.1 Overview of the Controller, Beep subsystem and System integration ........................................... 29

4.2 Execution period where a) All threads report performance and B) All threads collaborate but only one reports performance ......................................................................................................................... 31

4.3 Data structures used by the Beeps framework .................................................................................. 34

4.4 Communication data structures used by the beeps framework ..................................................... 34

4.5 Example of the data structure used in the controller ......................................................................... 36

4.6 Overall view of the desired controller modules ............................................................................... 37

4.7 Cases where it is possible and unlikely to have performance fairness across all applications ........ 38

4.8 Graphical interpretation of the share calculator ............................................................................ 38

4.9 Graphical interpretation of the frequency scaling effect .................................................................. 41

4.10 Range of virtual and their correspondent real frequencies for a heterogeneous platform (Exynos 5410 on the Odroid-XU+E) using cluster migration .................................................................................................................. 42

4.11 Migration controller concept .......................................................................................................... 43

4.12 Migration controller fairness contribution ...................................................................................... 44

4.13 Performance verification in the Migration controller (with forced shares update) ......................... 45

5.1 Performance and nice level of modeled applications (all threads reporting performance) with approach 1 ............................................................................................................................................. 51

5.2 Performance and nice level of modeled applications (all threads reporting performance) with approach 2 ............................................................................................................................................. 51
5.3 Performance and nice level of modeled applications (all threads reporting performance) with approach 3

5.4 Performance and nice level of modeled applications (parent thread reporting performance) with approach 1

5.5 Performance and nice level of modeled applications (parent thread reporting performance) with approach 2

5.6 Performance and nice level of modeled applications (parent thread reporting performance) with approach 3

5.7 The Odroid-XU+E development board used

5.8 Data structure and iteration scheme of memory micro-benchmark

5.9 Block diagram of the first controller implementation

5.10 Application performance during the simultaneous execution of benchmarks with the application share and frequency controllers

5.11 Frequency and power consumption during the execution of benchmarks with the application share and frequency controllers

5.12 Runtime performance, frequency and power consumption of two instances of a benchmark (4 threads each)

5.13 Runtime performance, frequency and power consumption during the execution of 3 benchmarks (2 threads each)

5.14 Runtime performance, frequency and power consumption during the execution of benchmarks with different thread numbers

5.15 Runtime performance, frequency and power consumption during the execution of 4 benchmarks (4 threads each)

5.16 Load of the proposed framework while handling 18 threads
List of Tables

2.1 Main characteristics of the development platforms considered ........................................ 14

5.1 Performance targets and deviation in each simulated application (all threads reporting 
    performance) with the different approaches ................................................................. 50

5.2 Performance targets and deviation in each simulated application (one thread reporting 
    performance) with the different approaches ................................................................. 52

5.3 Available tests on the PARSEC benchmark suite ......................................................... 55

5.4 Summary of the number of cycles to complete memory micro-benchmark ....................... 59

5.5 Program inputs and signal location used to test the benchmark performance on the A7 and A15 cores ................................................................. 60

5.6 Uncontrolled benchmark performance on the A7 and A15 clusters ................................ 60

5.7 Uncontrolled benchmark energy consumption on the A7 and A15 clusters .................. 61

5.8 Uncontrolled benchmark average power ratio (A7/A15) .............................................. 61

5.9 Frequency scaling times (DVFS) on the ODROID-XU+E ............................................ 61

5.10 Program inputs, signal location and maximum performance used to test the algorithm 
    using only the frequency and share controllers ......................................................... 62

5.11 Result summary for the controller combining application share and frequency control .... 63

5.12 Program inputs, signal location and maximum performance used to test the full implemen- 
    tation of the framework ............................................................................................... 66

5.13 Result summary for the full implementation of the framework ..................................... 66

A.1 Countable events through performance counters on the ARMv7-A ISA ..................... 85

B.1 Number of cycles to complete memory micro-benchmark ........................................... 89
## List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASTPI</td>
<td>Average Stall Time per Instruction</td>
</tr>
<tr>
<td>CFS</td>
<td>Completely Fair Scheduler</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DVFS</td>
<td>Dynamic Voltage and Frequency Scaling</td>
</tr>
<tr>
<td>DWRR</td>
<td>Distributed Weighted Round-Robin</td>
</tr>
<tr>
<td>EDP</td>
<td>Energy Delay Product</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>HMP</td>
<td>Heterogeneous Multi Processing</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated Circuit</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>KKT</td>
<td>Karush–Kuhn–Tucker</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PARSEC</td>
<td>Princeton Application Repository for Shared-Memory Computers</td>
</tr>
<tr>
<td>PID</td>
<td>Proportional-Integral-Derivative</td>
</tr>
<tr>
<td>PMU</td>
<td>Performance Monitor Unit</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>SHT</td>
<td>Signature History Table</td>
</tr>
<tr>
<td>SoC</td>
<td>System on a Chip</td>
</tr>
<tr>
<td>TDP</td>
<td>Thermal Design Power</td>
</tr>
<tr>
<td>WED</td>
<td>Weighted Euclidean Distances</td>
</tr>
</tbody>
</table>
1
Introduction
In the last decade, the computer (and processor) market has shifted from desktops, without strict power restrictions, to laptops and even more recently to tablets and smartphones. This tendency for mobility is creating an increasing demand for low-power and high performance devices. Power considerations are also becoming more relevant on desktops, since the density of modern Integrated Circuits (ICs) creates a power dissipation problem (currently identified as the “power wall” [1]). To overcome this issue, current system designers aim at developing efficient hardware-level techniques to achieve reductions in power consumption.

Most smartphones and tablets rely on ARM architectures to power their devices. In order to keep up with the increasing market demands, ARM architectures have evolved into true heterogeneous embedded platforms, where each new device generation delivers a greater performance, while consuming approximately the same (or ideally less) power. In order to achieve this high energy-efficiency, current embedded architectures turned to highly heterogeneous and asymmetric designs, where parallel computations can be performed across different clusters of multi-core processors that share the same Instruction Set Architecture (ISA). In addition to different sets of high-performance and energy-efficient cores, ARM architectures also combine Graphics Processing Units (GPUs) packaged in the same System on a Chip (SoC).

However, efficiently exploiting the full capabilities of nowadays asymmetric embedded systems is not and easy task [2, 3]. It is firstly required to carefully orchestrate the execution of parallel applications across a set of high performance, but power hungry, cores and a set of low power cores with limited performance. Furthermore, it is also required to drive these decisions according to the applications’ demands and characteristics, while respecting the tight energy budgets of embedded platforms. While this problem has given rise to the attention of researchers, existing solutions mainly focus on task distribution on homogeneous architectures by considering system thermal constraints or operational frequency [4, 5]. When compared to homogeneous systems, addressing this issue in heterogeneous architectures is significantly more challenging, since the number of parameters to consider when performing task scheduling is significantly increased.

There are only rare attempts that tackle this problem in heterogeneous systems [6–8]. These strategies range from using thread phases to build performance tables and guide task assignment decisions [9], to asserting if a migration can provide performance improvements based on stall time [10]. Nevertheless, for current asymmetric platforms, the existing approaches mainly combine a variety of metrics, such as Quality of Service (QoS), power awareness and core differences to distribute single-threaded tasks to the appropriate core based both on control theory and price theory [11, 12].

The work herein presented focuses on developing methods for optimizing the management of tasks on these heterogeneous architectures. These methods explicitly consider the application such that fairness among running tasks is achieved while creating the room for energy savings. Specifically, an adaptive performance-aware task management and frequency scaling method is proposed for modern heterogeneous embedded systems. This mechanism relies on online performance monitoring to explicitly capture the run-time behaviour of multiple parallel applications running on the underlying architecture. Based on the monitored application-specific parameters, the proposed method provides the
decisions regarding the allocation of shared system resources, such that the target performance levels are achieved, as well as to guarantee performance fairness among the running applications.

The proposed mechanism also relies on Dynamic Voltage and Frequency Scaling (DVFS) to manage the system energy-efficiency levels and to further augment the scope in which performance can be scaled such that energy consumption savings are achieved. Furthermore, the proposed lightweight method allows amending the heterogeneous system’s task scheduling, resource utilization and DVFS decisions, coupled with cluster migration, without the need to perform any kernel level modifications. This work is fundamental to attain high performance and energy-efficient execution on nowadays embedded system, as well as future full-scale server systems [13].

1.1 Motivation

Existing power and dissipation constraints on nowadays SoCs, make it hardly possible to simultaneously employ all the available resources, a problem which is expected to worsen due to the presence of Dark Silicon [14]. To sustain efficient task execution on today’s systems, which are growing to be ever-more heterogeneous, advanced task management mechanisms are required. First, task management needs to be adaptive, in order to allow amending the scheduling decisions according to the current state of the execution platform and during application run-time. Second, in order to guarantee the satisfaction of the target performance levels for a certain set of concurring applications, these mechanisms also need to be application-aware. As a result, by capturing the interaction between the applications and the underlying architecture, these self-learning approaches will allow amending the scheduling decisions according to the realistically assessed application requirements and the capabilities of the architecture to satisfy them. It is thus clear that these task management decisions not only need to consider the adequate thread placement and their migrations among different cores and clusters, but also the configuration of the execution platform, such that an energy-efficient execution is attained by setting the frequency levels according to the application needs.

Nowadays Operating Systems (OSs) provide two separate mechanisms for task management and frequency scaling, namely: i) the default task scheduler, i.e., Completely Fair Scheduler (CFS) [15], which is responsible for workload distribution among computational resources; and ii) DVFS mechanisms for run-time adjustment of the running voltage and frequency. Current DVFS strategies are nevertheless mostly unaware of the application performance, since they rely on the current system load to set the frequency level. As a result, as long as the running tasks are not significantly I/O bounded, they usually set voltage and frequency levels to the maximum. On the other hand, the CFS [15] tries to equally split the CPU time among the running tasks, thus implying that, although fairness is achieved in the time domain, the application performance might significantly differ from the desired levels. Since current systems generally do not include mechanisms to adapt the scheduling decisions according to the running workloads, the attainable application performance is usually bounded by what the system can currently deliver. However, attaining the application performance targets does not always require all available computational resources to be assigned (neither to be equally shared). In fact, the resources
should be allocated according to the determined application requirements, thus creating the opportunity for further energy savings and increasing the resource availability for other tasks.

The development of heterogeneous processors paves the way for new alternatives that combine, in the same SoC different hardware cores with micro-architectures better suited to run certain sets of tasks. Nevertheless, a bigger scheduling challenge is created since the managing software should be aware of the different micro-architecture characteristics. As stated before, a large body of previously conducted research work in this field does not take into account heterogeneity. However, the research on this subject is of the utmost importance to guarantee an efficient utilization of current and future embedded platforms, as well as to sustain further development of techniques possibly leading to improved devices. Hence, the goal of task scheduling in embedded systems is to optimize energy consumption while maintaining (or improving) the experience to the end user. Previous studies \cite{11, 12} have already shown advantages of taking into account heterogeneity, QoS and power awareness and they also confirm that scheduling techniques can be further improved when considering the existing hardware platforms.

In this thesis, a new task migration and scheduling framework, is proposed which takes into account both QoS and core differences. The proposed strategy is expected to fill the gap that existing methods leave, since they try to maximize the performance of a given task blindly, which might not be strictly necessary and possibly implies the allocation of unneeded resources. For instance, if the objective of a task is to decode a movie for playback, it only needs to respect a given frame rate and thus this is a QoS task. On the other hand, if the task aims to do computationally intensive work, such as transcoding a video file, it would be desirable to get the highest available performance, where methods based on stall time for non-QoS can be used (e.g., \cite{10}). As a result, a method that achieves a trade-off between QoS, performance, resources’ management and energy saving is evidently in demand, thus finding one such method represents one of the main goals of this thesis.

The implementation of a new OS level scheduling technique may cause practical problems, such as the need of program adaptation. While this is undesirable, Application Programming Interfaces (APIs), such as Application Heartbeats \cite{16}, have been developed allowing for the monitoring of any task’s QoS with the addition of only half a dozen lines of code. In some systems, this data can be enriched by the use of performance counters, which will allow the task scheduler to also consider, in real time, other factors such as memory accesses and/or branch mispredictions. Since, accessing the performance counters may not be feasible in every system, the controller should be robust enough to rely solely on the reported application performance.

1.2 Objectives

Based on the previous considerations, the main goals of this thesis can be summarized as follows:

- Modeling of application’s behavior on a given (heterogeneous) multi-core system;
- Run-time evaluation of current and attainable performance of applications;
• Design of a task management controller capable of maintaining performance fairness among all running applications on the system;

• Development of the controller for allowing energy-efficient application execution by considering frequency scaling through [DVFS]

• Evaluating the impact of micro-architecture differences on the application performance during a process’ run-time;

• Development of different metrics to evaluate the most appropriate core to execute a given task by considering all the available resources and according to the task characteristics;

• Design of a task scheduling framework applicable to a variety of tasks on embedded heterogeneous systems, by merging the above-referred functionalities;

• Experimental evaluation of the obtained performance with the proposed scheduler by measuring throughput and power consumption.

To achieve these goals, one must first answer three questions: “How?” (to tackle the problems presented by the objectives), “What is currently done?” and “What can be improved?”. While the first two questions require a study of the state of the art of the specific topics and an in depth analysis of the target platforms, the latter provides an escape route to develop new methodologies that can tackle the identified problems in innovative ways. This document will present all the findings and contributions that were made to develop a new task scheduling framework that respects as close as possible the proposed objectives.

### 1.3 Main contributions

In this thesis, a task management framework is proposed, which tunes the decisions made by the default system scheduler [CFS] by adjusting not only the (time) shares assigned to threads running on a core, but also by determining the most adequate system frequency and thread to core allocation (flexible enough to handle both single- and multi-threaded applications). The proposed method allowed reducing the relative performance error of all running applications to a given target (from 11.800 to 0.096 in some cases), while also significantly decreasing the energy spent while executing them (up to 65% on the tested scenarios) when compared to default system tools.

This work was communicated in the proceedings of the SBAC-PAD 2014 international conference with the following paper:


### 1.4 Outline

The remainder of this document is organized as follows: Chapter 2 presents the embedded heterogeneous platforms and architectures targeted by this work. Chapter 3 introduces current state of the art
scheduling methodologies and provides an overview of the default system scheduler in Linux as well as of some other important system components. The proposed performance and energy-aware real-time scheduling for heterogeneous embedded systems framework is introduced in Chapter 4. Chapter 5 then provides a thorough evaluation of the framework, followed by some conclusions in Chapter 6.
Embedded Heterogeneous Platforms

Contents

2.1 ARM big.LITTLE Architecture .................................................. 8
  2.1.1 ARMv7-A Instruction Set Architecture (ISA) ......................... 9
  2.1.2 Cortex-A15 and Cortex-A7 Microarchitecture ......................... 10
  2.1.3 Performance Monitor Unit (PMU) ........................................ 11
2.2 Task migration techniques ..................................................... 11
  2.2.1 big.LITTLE Task Migration Model ....................................... 11
  2.2.2 big.LITTLE MP .......................................................... 13
2.3 Available development boards ............................................... 13
2.4 Summary ............................................................................. 14
The increasing complexity of mobile devices’ software and applications requires processors to provide higher performances at the same energy budgets. To tackle this challenge, processor architects are turning to heterogeneous multi-core designs. One of the companies responsible for such architectures, ARM, develops not only the ISA used on most mobile devices (currently, ARMv7-A for 32-bit computing and ARMv8-A for 64-bit) but also proprietary cores (e.g., the ARM Cortex-A series). As such, it comes with no surprise that most chip makers choose to implement ARM IP cores on their systems. As a way to cope with today’s mobile computational demands, ARM has developed the big.LITTLE heterogeneous platform [17], a technology that combines cores with different characteristics but a common ISA in the same SoC.

Other manufacturers, such as Nvidia and Qualcomm, have developed their own heterogeneous SoCs. Nvidia has its Tegra family [18] (combining Cortex-A15 cores with a power efficient one) and Qualcomm has expanded its heterogeneous family started with the Snapdragon 800 to the Snapdragon 810 series harvesting the power of heterogeneity in the 64-bit mobile computing world [19]. With ARM being the reference for most of today’s mobile systems, the focus of the research herein presented will be the ARM big.LITTLE embedded heterogeneous platforms. However, the proposed techniques are general enough to be applied even to other system architectures, i.e., they are not specifically bound to ARM technologies. Taking this into consideration, the remaining of this chapter is organized as follows: first a brief general explanation about the big.LITTLE architecture will be given, followed by an overview of the ARMv7-A ISA and the architectures of the cores used on big.LITTLE. The focus will then be shifted to the migration techniques used in such heterogeneous systems and an analysis of development platforms powered by ARM.

2.1 ARM big.LITTLE Architecture

big.LITTLE is a technology developed by ARM, which combines in the same SoC multi-core processors designed for different power and performance budgets, thus allowing for a more efficient system [17].

In current big.LITTLE implementations, Cortex-A15 (higher performance) cores are combined with Cortex-A7 (energy efficient) cores, both using the ARMv7-A ISA. Tasks can be assigned to these cores dynamically, according to their needs. This execution paradigm allows the system to save energy by shifting simple tasks to the energy efficient cores and maintain the performance of more demanding tasks by running them on the higher performance cores. Since all cores share the same ISA, existing programs can run natively in either type of core and be transparently migrated.

Each cluster, which is a group of similar cores that share the same L2 cache, is composed by different core types (e.g., A7 or A15). Currently both 32-bit (A7 and A15, e.g., Samsung’s Exynos 5422 [20]) and 64-bit (A53 and A57, e.g., Qualcomm’s Snapdragon 810 [19]) implementations of big.LITTLE are present on the market. The merging of these two clusters of Central Processing Units (CPUs) is possible through the use of the CoreLink CCI-400 Cache Coherent Interconnect, and the GIC-400 Interrupt Control to distribute interrupts across the clusters. ARM has developed other cache coherent interconnects (CCN-
504 and CCN-508) that can connect 4 and 8 clusters, respectively, opening the door to the creation of other types of heterogeneous systems with growing complexity and capacities. One particularity of these interconnects is their ability to connect hardware accelerators and media processors, such as the Mali-T600 series [GPU] to all the CPUs. This presents another possibility for optimizations since most of current [GPU] support OpenCL (allowing for general computations to be run on [GPU]). A diagram of a possible implementation of a big.LITTLE system, featuring interconnects, (64-bit) CPUs and GPU is depicted in Fig. 2.1.

Figure 2.1: Diagram of a possible big.LITTLE system using the CCI-400 and 64-bit cores (Cortex-A57 and Cortex-A53)

2.1.1 ARMv7-A Instruction Set Architecture (ISA)

One of the main characteristics of the ARMv7-A [ISA] (used on the Cortex-A7 and Cortex-A15 that will be the focus of this work) is that it can be implemented on a wide range of microarchitectures with different performance and efficiency levels allowing diversified types of CPUs to be developed. From the application point of view, the ARMv7-A [ISA] implements 13 general-purpose 32-bit registers and 3 extra 32-bit registers providing special features [21]. It uses a Reduced Instruction Set Computer (RISC) architecture enriched with the ability to combine arithmetic or logical operations with shifts on the same instruction and auto-increment (or decrement) addressing modes. The latter allows optimizing program loops to load (or store) multiple consecutive memory addresses to the registers (or vice-versa) in a single instruction. This architecture further allows the conditional execution of several instructions to maximize data throughput. The by-product of all these factors is the balance of high performance, small program size, low power consumption and small die area which ARM processors can achieve.

Although the architecture used in this work is the ARMv7, which provides 32-bit instructions, the new ARMv8 architecture has now also reached the market. This new architecture features 64-bit instructions and is deployed on ARM's 64-bit mobile CPUs, paving the way for 64-bit computing on mobile and low power processors [22].
2.1.2 Cortex-A15 and Cortex-A7 Microarchitecture

The Cortex-A7 and Cortex-A15 processors, which are used throughout this work, implement the full ARMv7-A ISA including Virtualization and Large Physical Address Extensions, as well as NEON instruction set extensions [17], thus allowing for a seamless migration of tasks between processors apart from the performance difference.

The distinctions between cores are nevertheless clear at a micro-architecture level. The Cortex-A7 is an in-order, dual-issue processor with a pipeline structure of 8 to 10 stages as represented in Fig. 2.2. On the other hand the Cortex-A15 is an out-of-order, triple-issue processor, with 15 to 24 pipeline stages (see Fig. 2.3).

![Figure 2.2: Pipeline diagram of the Cortex-A7](image)

![Figure 2.3: Pipeline diagram of the Cortex-A15](image)

Since the amount of energy spent for the execution of an instruction partially depends on the number of pipeline stages, a difference on this number significantly affects the energy consumption of the core.
This fact, allied with the increase in complexity (and consequently power consumption) needed for the extra capabilities of the A15, makes the A7 a more power efficient core.

### 2.1.3 Performance Monitor Unit (PMU)

Both A7 and A15 cores include performance counters as part of their Performance Monitor Unit [23]. Both have a cycle counter in addition to six (in case of the A15) or four (A7) other counters, capable of counting events that occur in the processor. The absolute values recorded may present variations due to pipeline effects but these should be negligible, unless the counters are enabled only for a very short amount of instructions.

To trigger and access performance events the programmer must first configure the Performance Monitor Unit in order to assign an event to each of these counters, such as branch conditions, mispredicted branches, cache accesses, refills and write-backs and memory accesses. The standard events which are countable in all ARMv7-A processors are listed in Table A.1 of Appendix A. While there are other processor specific events that may represent a significant addition to the listed events, their availability in each of the cores must be checked independently.

These counters can be used either with an external debugger or can be accessed by the processor itself in order to benchmark and profile the execution of tasks in real time. However, in order to access this functionality, the counters must be enabled by an external signal, whose specification is manufacturer dependent.

### 2.2 Task migration techniques

Two main methods are used for task migration in big.LITTLE systems namely: big.LITTLE Task Migration and big.LITTLE MP [17]. Both migration schemes share some important common factors, which are mainly related to the handling of the migration. Due to the CCI-400 interconnect, memory coherency is guaranteed by the hardware without the need of software managing (i.e., the need for the CPU to copy the cache contents to the main memory, so that that data can be used by other components, such as a different core cluster or GPU). The software will nevertheless have to handle the supporting mechanisms for the migration. Examples of these are the state save-restore tasks, bringing the processors in an out of coherency, control snooping in the interconnect and migrate interrupts (via the GIC-400 Interrupt Control).

#### 2.2.1 big.LITTLE Task Migration Model

The big.LITTLE Task Migration Model represents the simplest way to migrate tasks on big.LITTLE platforms, and can be viewed as a natural extension to Dynamic Voltage and Frequency Scaling (DVFS) (i.e., the adjustment of the operational frequency of the processor).

Several OSs are already energy aware and implement DVFS in order to optimize the system energy budgets by changing the CPU operating frequency and voltage to the most appropriate level according to tasks’ demands. For example, big.LITTLE can take advantage of this method by running less demanding
tasks on the A7 cluster with lower voltage and frequency. When the A7 cluster reaches its maximum
performance point, this task migration model migrates all the tasks from the A7 cluster to the A15.

DVFS could then continue to be applied, but with new (higher) performance points calibrated for the A15
cluster.

In order to ensure that coherence is maintained, task migration requires several steps to be taken on
the inbound and outbound clusters (or CPUs). These steps are, illustrated in Figure 2.4, and they include
the powering on and reset of the inbound processor, the transference of state (integer, advanced SIMD
and configuration register contents), snooping enabling (to accelerate the warm up of the L2 cache in
the inbound processor) and finally the power down of the now inactive CPU (or cluster) to prevent power
leakage.

![Figure 2.4: Steps to be taken while performing a task migration on the Task Migration Model](image)

One important factor to consider is the task migration cost (in time) since performing too many mi-
grations in a short period of time might provoke overall performance degradation due to frequent context
switching. Regarding the migration time interval, ARM reports that the system is designed to migrate
in less than 20,000 cycles (or 20 µs at 1 GHz), while reports migration times as high as 3.83 ms (fre-
quency dependent). Another relevant point is that instead of migrating the tasks from one whole cluster
to another, migration might be performed between CPUs only, i.e., by keeping both clusters running at
the same time. However, both for cluster and CPU migration under this model, the system should have an equal number of CPUs in each cluster and the OS will only see half of the total number available at any time, i.e., if the system has 2 A15 and 2 A7, under this model, the system will only see 2 cores available, whose type can change during execution.

2.2.2 big.LITTLE MP

Due to the use of the CCI-400 Cache Coherent Interface and the GIC-400 Interrupt Control, all A7 and A15 cores can be simultaneously powered on and used to run applications. The big.LITTLE MP topology, which is the essence of Heterogeneous Multi Processing (HMP), allows for a better division of tasks between these cores. Since the A15 cores consume a high amount of power, they should only become active when specific applications require a significant amount of computational power, while the A7 cores can keep on working on less demanding tasks. For instance, user interactive tasks (e.g., graphics related) can be run on the A15 cores to ensure a smooth experience, while background tasks (such as email and notification checking) can be performed on the A7 cores by considering that a small delay is acceptable.

Implementing the big.LITTLE MP topology presents a greater challenge in comparison to both the previously presented Task Migration Model and homogeneous platforms, because the right core must be found for each task. This attribution is typically done by the scheduler that has to be aware of the core’s topology and characteristics. Note that in the previously presented migration scheme a DVFS frequency point would map directly to a given type of core for a task. Here, each cluster will have its own DVFS management and tasks must be migrated accordingly.

2.3 Available development boards

Development platforms are an essential tool to develop and analyse new scheduling policies since they provide the possibility of implementing and testing these policies on real hardware and to experimentally evaluate how performance is affected. Several of these platforms, powered by SoCs with ARM’s Cortex-A processors capable of running both Android and Linux systems, can be found on the market. However, not all of the available development boards feature big.LITTLE. Nevertheless, a brief analysis will be presented on the ones that do support the big.LITTLE heterogeneous architecture. The most complete boards in terms of features are the ones built by ARM, specifically the Versatile Express and the Juno. While the former can have different CPU configurations, one of them featuring big.LITTLE via a mix of A7 and A15 cores (32-bit), the latter has only one available SoC featuring a 64-bit implementation of big.LITTLE with A57 and A53 cores. Both of these options offer tools to measure energy consumption and support all big.LITTLE migration techniques.

Other manufacturers also provide big.LITTLE enabled boards, mostly powered by Samsung SoCs. The Arndale Octa Board, Odroid-XU+E and Odroid-XU3 are examples of such boards, unfortunately the first two lack support for all big.LITTLE migration modes, which is only available on the Odroid-XU3 due to SoC related issues. However, this board is a rather recent one and was not available
throughout the development of this work. This coupled with the difficulty in getting access to the ARM development boards dictated the selection of the Odroid-XU+E, which supports only cluster migration but includes current sensors to measure its power consumption. A Summary of the main characteristics of the Versatile Express, Odroid-XU+E, Arndale Octa Board, ARM Juno and Odroid-XU3 is presented in Table 2.1, furthermore, a more detailed examination of the Odroid-XU+E [24] will be made while reviewing the experimental platform in Chapter 5 where further hardware and software details shall be explained.

Table 2.1: Main characteristics of the development platforms considered

<table>
<thead>
<tr>
<th>Board</th>
<th>Versatile Express</th>
<th>Odroid-XU-E</th>
<th>Arndale Octa Board</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td>3 x Cortex-A7; 2 x Cortex-A15</td>
<td>4 x Cortex-A7; 4 x Cortex-A15</td>
<td>4 x Cortex-A7; 4 x Cortex-A15</td>
</tr>
<tr>
<td><strong>RAM</strong></td>
<td>2 GB DDR2</td>
<td>2 GB of LPDDR3</td>
<td>2 GB of LPDDR3e</td>
</tr>
<tr>
<td>Maximum Frequency</td>
<td>800 MHz - A7; 1 GHz - A15</td>
<td>1.2 GHz - A7; 1.6 GHz - A15</td>
<td>1.2 GHz - A7; 1.8 GHz - A15</td>
</tr>
<tr>
<td>big.LITTLE modes</td>
<td>Cluster/Core Migration; MP</td>
<td>Cluster Migration</td>
<td>Limited to A15 cluster by software</td>
</tr>
<tr>
<td>Available Accelerators</td>
<td>Optional FPGA Logic Core</td>
<td>PowerVR SGX544MP3 [GPU]</td>
<td>ARM Mali-T628 MP6 [GPU]</td>
</tr>
<tr>
<td>DVFS</td>
<td>Per cluster</td>
<td>Per cluster</td>
<td>Per cluster</td>
</tr>
<tr>
<td>Performance counters available</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Board</th>
<th>ARM Juno</th>
<th>Odroid-XU3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td>4 x Cortex-A53; 2 x Cortex-A57</td>
<td>4 x Cortex-A7; 4 x Cortex-A15</td>
</tr>
<tr>
<td><strong>RAM</strong></td>
<td>8 GB DDR3L</td>
<td>2 GB of LPDDR3</td>
</tr>
<tr>
<td>Maximum Frequency</td>
<td>850 MHz - A53; 1.1 GHz - A57</td>
<td>1.4 GHz - A7; 2.0 GHz - A15</td>
</tr>
<tr>
<td>big.LITTLE modes</td>
<td>Cluster/Core Migration; MP</td>
<td>Cluster/Core Migration; MP</td>
</tr>
<tr>
<td>Available Accelerators</td>
<td>Optional FPGA Quad Core MALI T624</td>
<td>Mali-T628 MP6 [GPU]</td>
</tr>
<tr>
<td>DVFS</td>
<td>Per cluster</td>
<td>Per cluster</td>
</tr>
<tr>
<td>Performance counters available</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

2.4 Summary

In this chapter a brief introduction to heterogeneous platforms currently employed in embedded systems was made. The focus of the analysis falls on ARM’s big.LITTLE [17] and different SoCs using both this and other technologies were identified [18-20]. The ARM multi-core architecture that is the target of this work was presented, and a collection of proposed thread migration schemes is discussed. The architecture is designed for low power operation while providing performance capable of keeping up with today’s demand. The ISA used on this technology was also introduced, specifically the analysis focused on the 32-bit [21] ISA although new 64-bit [22] ISA was also referred.

The microarchitecture differences between the Cortex-A7 and Cortex-A15 (a common big.LITTLE pair) were also reported, as well as the available counters on the PMU. Available thread migration meth-
ods in big.LITTLE [17] were also addressed, which further develops the idea of efficiency by combining power efficient cores with performance optimized ones on the same SoC. To conclude this chapter a selection of development boards that have big.LITTLE powered SoCs was also presented. While ARM’s development solutions offer the most flexibility, their unavailability dictated that the selected board to conduct this work was to be the Odroid-XU+ E [24], which presented a good compromise.
3 Scheduling and Dynamic Voltage and Frequency Scaling (DVFS)

Contents

3.1 Processor demand scheduling ........................................... 18
3.2 Quality of Service (QoS) aware scheduling ...................... 20
3.3 Current default linux scheduler - CFS .............................. 21
   3.3.1 Priorities and nice ............................................. 23
   3.3.2 Migration across cores ...................................... 24
3.4 Dynamic Voltage and Frequency Scaling (DVFS) ............... 25
3.5 Summary ..................................................................... 25
Several dynamic task scheduling techniques have already been studied in the literature, both targeting homogeneous and heterogeneous processors. In general, current state of the art task scheduling techniques can be categorized in two separate groups, namely: processor demand and QoS-aware scheduling approaches. The former approaches try to maximize the performance of all the tasks in a system by looking at CPU utilization data, whereas the latter is aware of the performance demand of individual tasks and can combine tasks in optimal ways to achieve that performance without allocating more system resources that it is required for an individual task.

Some of these techniques, considered to be relevant to the established objectives, are herein briefly described. Analysing them not only reveals important background on state of the art scheduling techniques, but also provides the foundations where new schemes can settle on. However, these techniques often rely on changes to the scheduler’s structure of the system and may be specifically tailored for certain systems/applications combinations. Since one of the main goals of this thesis is to amend the decisions of the default scheduler (by introducing an external controller that interfaces with it), its basic functional principles are also briefly described herein. As a complementary aspect to scheduling, the concept of DVFS and its current implementation is introduced, since it represents an important facility to attain further control over the system resources to achieve energy savings.

3.1 Processor demand scheduling

As previously referred, processor demand scheduling relies on metrics and measures obtained directly from the processor (as opposed to application feedback) to evaluate how the system resources are being utilized by the running applications and evaluate if a different allocation of resources could possibly improve the current performance.

L. Sawalha et al. [9] proposed a method to assign tasks to cores according to the corresponding application phase. This method is aware of core heterogeneity and optimizes the task distribution according to which core is the most appropriate for a given phase of the current task. The characterization of a phase is performed on its first occurrence, and it is then used to predict the performance of the program when it resurges further on during execution. A phase can be determined by periodically monitoring an application execution, e.g., by the number of instructions per cycle. The first time a phase occurs, it is run across all available cores in order to evaluate the performance on each one. Having built a Signature History Table (SHT) for the known phases in a thread, the tasks scheduling is accomplished by looking at the SHTs and choosing the best combination of performance levels for all the running phases of current threads. This technique requires extensive experimental testing to characterize application phases and tuning of some parameters, namely the sample window size, and it does not consider DVFS.

P. Nie and Z. Duan [10] suggest another scheduling technique for heterogeneous systems, by analysing the Average Stall Time per Instruction (ASTPI), which represents the average time an application waits for memory contents. Since an application is monitored periodically, it also allows for the identification of different program phases. For this, the memory hierarchy of each core is assumed to
be the same, i.e., the stall time is considered to be similar independently of the type of core. A small ASTPI is then expected to translate in a greater ratio of arithmetic over memory operations, thus providing possible insights about whether or not a task benefits from migrating to a fast core. It was verified in [10] that the ASTPI was in fact constant across core types\(^1\) and that the speed up factor of a thread is monotonically decreasing as ASTPI grows. One clear advantage of this method is the lack of the need to test a task across all types of cores, but it does not take into account other micro architectural differences nor performance information from the task.

A different approach was adopted by P. McKenney et al. [25], where the impact of non-performance-critical work invoked by performance-critical code is reduced. In this specific case the authors addressed the read-copy update mechanism which is used in the Linux Kernel (and now also in user space) to read-write lock data (where several threads can read the same data but only one can write it). Two methods to achieve power reduction on this operation are considered. The first method reduces the number of scheduling-clock interrupts appearing on an otherwise idle processor while a wait-for-readers operation is pending (replacing it with sparser, timed checks). The second method is to offload the referred operations to a lower performance CPU. The combination of these two methods does not yield better results in terms of energy efficiency, since both methods aim to reduce the same busy period on a high performance CPU, and can actually lead to a degradation of performance due to the additional overhead. These methods equally reduced power consumption, but enforcing a larger idle period had a more significant (although low) impact on performance due to the increase on the grace-period.

Another interesting contribution was made by J. Chen and L. John [26] who rely on the euclidean distance between a core’s capabilities and the program’s resource needs as a metric to guide the scheduling of tasks. In this method, the programs are first profiled to obtain a given number of relevant metrics. These metrics are then projected onto a unified multi-dimensional space where metrics coming from the processors configuration are also projected. For instance, the metrics on the processor side can be: the issue width, branch predictor and L1 data cache size while on the program side instruction level parallelism, branch predictability, and data locality can be used. Then, the distance between task and core is measured in the space where the projection occurs (where different dimensions of the vectors can have different weights) and the tasks are attributed to the closer processor. This method, which was called Weighted Euclidean Distances (WED), proved to have a good correlation with Energy Delay Product (EDP) with greater distances yielding a worst match between core and task.

The work in [27] aims at improving the scheduler’s fairness in order to guarantee that the shares attributed to a task are correctly enforced across cores. This technique is applied both for homogeneous and heterogeneous platforms [28], however it does not target the enforcement of performance levels (or energy budgets). Instead, it targets the equalization of the tasks’ shares (according to their weight), thus contributing for an improved load fairness in the system.

Other examples that target energy budgeting can be found in the literature, but they mainly address homogeneous systems [29–31]. [29] uses offline models to build lookup tables that are used to decide core placement and DVFS levels of the running workloads. [30] focuses on chip-level monitoring, control

\(^1\)The test was run on both an Intel and AMD multicore machines with different core types simulated by setting different frequencies and enabling/disabling cores to create differentiated memory organization schemes.
and dynamic management of power while respecting objectives such as prioritization, power balancing and throughput on different benchmarks. Finally, [31], relies on a three step approach to control power budgets and frequency levels per core, while considering mixed groups of applications (single- and multi-threaded). Nevertheless, these methods do not consider heterogeneity, they rely on performance counters and consider per core frequency scaling, characteristics which are incompatible with the systems targeted in the scope of this thesis.

3.2 Quality of Service (QoS) aware scheduling

In contrast to the previously referred scheduling techniques that rely on total execution time, QoS aware scheduling approaches rely on the QoS of the tasks as a migration metric. This scheduling class is particularly important for tasks interacting with the user. These tasks, such as video playback and games, may require a minimum processing rate to be respected, although a maximum one can also be set.

An important challenge in QoS awareness is how to practically assess the QoS of individual tasks. Hoffmann et al. [16] proposed an API to measure QoS called Application Heartbeats. This API was developed with the intent of improving task efficiency by adjusting resources in order to strictly fulfill the task’s necessities. A drawback of this approach is that it requires changing the application’s source code (although these changes can be made in less that half a dozen lines of code). The application can then set a minimum and maximum heartbeat rate (or the time interval between two consecutive heartbeats), which can be analyzed internally or externally to adjust either the program or the resources attributed to it.

T. Muthukaruppan et al. [11] designed an integrated scheduling solution for controlling several parameters of the processor with the help of Application Heartbeats for QoS tasks. This approach was directly applied to big.LITTLE, thus proving the applicability of most of the discussed concepts to the target platform of this thesis. Due to the fitting of this research in the scope of this thesis a more in depth analyses of it will be made herein.

The technique proposed in [11] consists on building a Proportional-Integral-Derivative (PID) feedback control loop which analyses and adjusts several parameters, such as tuning QoS targets, adjusting DVFS to a given cluster and migrating tasks when the core where they currently run does not deliver the desired performance level. An overview of the control loop that implements the per-task resource share controller, per-cluster DVFS controller and per-task QoS controller is depicted in Figure 3.1. This diagram shows the control flow of the whole system, where a Chip-Level Power Allocator is used for specifying a throttling level for QoS tasks and CPU utilization factor. The QoS Controller is responsible for applying the specified throttling and its functionality is tightly coupled with the resource share controller and balancer block and cluster DVFS controller. The migration is activated whenever a task cannot fulfill its specified QoS while running at the maximum frequency of the A7 core.

A central focus was given to the Thermal Design Power (TDP), responsible for setting a limit to the power a chip can consume in order to respect thermal requirements. If the power consumption is
above the limit, a throttle to QoS is set along with a request for higher CPU utilization (resulting in a reduction of the CPU frequency). These controllers are called at different intervals in order not to cause any performance degradation, by oversubscribing the CPU with scheduling tasks, as well as to respect the inherent response latency of system components (such as thermal response). Correspondingly, the per-task resource share controller and load balancer are more frequently invoked than the per-cluster DVFS and per-task QoS controllers. The Chip-Level Power Allocator is called less frequently than the other controllers, since the thermal response is the slowest among all the considered components.

While the work proposed by T. Muthukaruppan et al. [11] allows achieving significant energy savings, it still leaves room for improvement. For example, it only handles single threaded tasks and non-QoS tasks are not explicitly considered, thus they can only obtain the maximum performance that remains from the QoS tasks. In a more recent work [12] by the same authors, some inefficiencies on the handling of task distribution and heterogeneity are identified and a new, even more complex controller is proposed whose implementation is based on price theory. In fact, the complexity of this controller underlines the difficulty of handling the heterogeneity, performance awareness and energy savings in a real control system.

3.3 Current default Linux scheduler - CFS

The Linux Process Scheduler is the portion of the Linux Kernel responsible for managing a system’s tasks workload distribution. Being an essential component of a system it affects a diverse number of metrics, namely the throughput, latency and fairness. As such, most of the techniques presented in the previous sections require altering the Linux scheduler, in order to be implemented. Although such modifications are diverse, most of them can be seen as an extension to the existing scheduler. The scheduler in current Linux systems has thousands of lines of code causing a detail description of its full functionality to be highly impractical. Instead a brief description of the standard scheduler is provided
As in other operating systems, scheduling policies in Linux evolved (and are evolving) gradually with alternatives and optimizations appearing constantly [15]. Mainline Linux distributions started to incorporate the idea of scheduling classes in version 2.2, thus allowing for different policies to be taken for real-time tasks, non-real-time tasks, and non-preemptible tasks.

Version 2.4 of the Kernel implemented a O(N^2) scheduler that assigned time slices to tasks allowing them to execute up to a maximum (fixed) time interval. Version 2.6 introduced a new scheduler (called the O(1) Scheduler), solving some issues of the previous scheduler. One of the main differences was that it did not have to iterate over all the tasks (thus the O(1)). However, its implementation complexity proved to be a significant downside, thus the Completely Fair Scheduler (CFS) was introduced.

The CFS main objective is to maintain fairness, i.e., distributing the available CPU time among active tasks equally. This scheduler also includes a concept of sleeper fairness to ensure that sleeping tasks (such as tasks waiting for I/O) get a fair share of the processor when they actually require it. A key aspect of the CFS functionality is that it maintains tasks in a time-ordered red-black tree (Figure 3.2). This tree has the properties of being self-balancing, where a path in the tree is never longer than twice the distance to any other. Furthermore, operations (such as reinsertions) can be performed in O(log(N)), thus allowing for fast insertions and deletions from the tree. It is worth noting that although the O(1) scheduler provides a better performance than the CFS, but the simplified structure of the latter, resulted in the dropping of the O(1) in favor of the CFS.

As depicted in Figure 3.2, the scheduler picks the node on the left-most side of the time-ordered tree (marked as 1st), which represents the task with lower run-time. After the task it represents is running, the scheduler updates the run-time of the task and reinserts it into the tree. This creates a migration from right to left in the tree, which allows the scheduler to maintain fairness.

CFS handles the specification of a task priority by scaling down the time a task is allowed to execute when its priority is reduced (or scaling up the time if the priority is increased). It further ensures fairness thanks to group scheduling. Group scheduling is particularly useful in tasks that spawn many other

---

2 The O(•) notation means that an algorithm can execute on a time which is proportional to •. O(1) means that the algorithm can be performed in constant time (independent of the number of inputs), O(N) linear time (N being the number of inputs), O(log(N)) a time proportional to the logarithm of the number of inputs and so on. In the scheduler’s case, the inputs are the tasks to schedule.
tasks, such as a server application that handles several input connections, and attempts to provide a fair behavior in each group. In the previous example, the CFS allocates a given slice of usage to the server task (divided among the other tasks in the system) and its sub-tasks would receive an equal fraction of that time. This way fairness is maintained at several levels (groups).

There is also the possibility of having different scheduler classes on the same system and allowing tasks to be attributed to the range of the referred classes. This can be useful to change the scheduling scheme of tasks that require real time processing. Another important concept is scheduling domains, that create the possibility of hierarchically grouping processors to make a better load balance and segregation of tasks between them. Since processors can share scheduling policies, this scheduling methodology is particularly important for multiprocessor systems, such as the case of HMP. Nevertheless, and as will be discussed further on, current implementations may, by default, lose some fairness while handling core balancing in multi-core systems [27].

3.3.1 Priorities and nice

As previously stated, current Linux systems use the CFS [15] as the default system scheduler. The aim of this scheduler is to maintain fairness (in the time domain) among running tasks and it does so by maintaining an ordered red-black tree based on the current virtual run-time of the running tasks. To calculate this virtual run-time not only the real run-time of the task is taken into account, but also a weighting factor. This means that a given number of real run-time units does not necessarily directly corresponds to the same number of virtual time units for every task.

In practice, the execution of multiple independent tasks is managed by attributing a small time slice to each running task and by periodically switching their execution on the assigned CPU core. As a result, by switching the execution of different tasks very quickly, the end-user becomes unaware of these context changes and has a perception of a seamless execution. Having the liberty to control how virtual runtime scales means that a task can be made to execute for a longer period of time effectively allocating more CPU resources and thus increasing its priority in relation to other tasks.

Figure 3.3 presents an example of the execution switch among three tasks (i.e., A, B and C) within a scheduling period (epoch). In this figure, Epoch i illustrates a default case where all three tasks are assigned an equal time slice to execute, i.e., the overall scheduling period is equidistantly partitioned among the running tasks and their execution is performed in a round-robin fashion. Hence, the CFS [15] splits an epoch such that each task has a similar virtual run-time and no indications of application performance are taken into account when deciding time shares.

Figure 3.3: A graphical interpretation of the basic scheduler functionality

The aforementioned weighting factor depends on the nice value that was assigned to a given task (whose default value is 0). This parameter is worth exploring since it will have a key contribution for the implementation of this work. In order to change the priority of a task, one should change the task's nice...
value, which also scales the weight (and virtual run-time) of the task. The nice values are selected from a predefined range of integer values, namely from -20 to 19 [32]. Accordingly, the time share ($s_i$) of a task $i$ running on a certain core can be calculated as follows:

$$ s_i = \frac{1024}{\sum_{j=1}^{N} 1.25^{n_j}}, $$

(3.1)

where $N$ is the total number of running tasks in the core and $n_i$ is the nice level of task $i$. It is worth noting that the time share of a particular task $i$ is calculated relatively to the nice levels of all $j$ tasks currently assigned to execute at the same core, i.e., $n_j$ nice values. Hence, if all tasks $j$ maintain their nice levels, the time share of the single task $i$ can be increased by lowering the nice value; naturally, increasing the nice level implies a smaller time share.

As a result, the nice values provide a way to control the time shares on a per-task basis. An example of unbalanced time shares due to different nice levels can be seen in Fig. 3.3 for epoch $i+1$. In detail, task $B$ has a larger time share than tasks $A$ and $C$ due to the fact that the nice levels are set in the following order: $B < C < A$, whereas all tasks have similar nice levels for epoch $i$. This illustrates that by controlling the task time share (i.e., execution time at the level of the epoch), it is possible to influence the achievable performance levels of multiple simultaneously running applications.

### 3.3.2 Migration across cores

The CFS handles migration across cores via its load balancing mechanism. It is worth emphasizing that fairness, in multi-core systems, is not guaranteed when migration is triggered, which means that possible deviations from the ideal distribution are bound to happen [33, 34]. The most important aspects of this scheme are summarized in the following text.

The CFS defines the load of a core’s run-queue, $Q_k$, as:

$$ L_k = \sum_{\tau_i \in S_k} W(\tau_i), $$

(3.2)

where $S_k$ is the set of tasks in $Q_k$ and $W(\tau_i)$ is the weight of task $\tau_i$. It then triggers load balancing at predefined time intervals (or if a run-queue is empty). The load balancing tries to move tasks from the busiest run-queue ($Q_{busiest}$) to $Q_k$, and the amount of load to be moved is defined as:

$$ L_{imbal} = \min(\min(L_{busiest}, L_{avg}), L_{avg} - L_k), $$

(3.3)

where $L_{avg}$ is an average system load and $L_{busiest}$ is the load of $Q_{busiest}$. In order to ensure fairness, the migration is only triggered if

$$ L_{imbal} \geq \min_{\tau_i \in S_{busiest}} (W(\tau_i))/2, $$

(3.4)

where $S_{busiest}$ is the set of tasks in $Q_{busiest}$.

The condition in (3.4) means that in certain situations, such as the one presented in Fig. 3.4 the CFS will not achieve fairness, since the minimum imbalance to trigger a migration will not be met (the imbalance is 158 and it needs to be at least $335/2=167.5$ to trigger the balancing). This means that,
proportionally to the assigned task weights, task 1 will run 4 times longer than tasks 2 to 5, although their weight difference may suggest that it should run for about 3.06 times longer.

To solve this issue, alternatives to the CFS migration scheme are proposed, such as Distributed Weighted Round-Robin (DWRR) [27]. DWRR maintains two run-queues per core (round-active and round-expired) and it periodically triggers migrations, such that the tasks run for a time period proportional to their weight, thus effectively enforcing fairness with a low additional overhead to the system.

### 3.4 Dynamic Voltage and Frequency Scaling (DVFS)

A complementary aspect to the system scheduler is DVFS. This mechanism scales the system frequency and voltage in order to achieve certain energy savings and it is usually applied at the level of the clusters of cores. The DVFS functionality is defined by different governors [35] that usually allow controlling the system frequency by relying on different static and dynamic strategies. For example, static strategies provide the execution modes for achieving the maximum performance or maximum energy savings by fixing the running frequency at the maximum or minimum value, respectively. On the other hand, the dynamic strategies allow run-time frequency scaling based on the current system load.

This brings the issue that current DVFS strategies mainly target energy savings from the architectural point of view, thus generally not implying an energy-efficient execution of parallel applications with performance constraints. As a result, there is room for improvement in this field by relying on the ability to set the desired operating frequency, such that the target performance of several simultaneously running multi-threaded applications is automatically achieved while minimizing energy consumption.

### 3.5 Summary

This chapter summarizes existing dynamic performance- and energy-aware scheduling methodologies with distinct objectives, ranging from achieving maximum performance to achieving a good energy balance. The current default system scheduler in Linux (CFS) is also introduced, and some of its key functions are examined. Typically, performance-aware methodologies for heterogeneous systems al-
locate tasks to cores based on their memory hierarchy or some micro-architecture functionality better suited for a given task. Other methods are QoS aware, but lack the generalization for single-threaded and multi-threaded applications as well as efficient controller implementations.

As for the [CFS], the current Linux scheduler, a brief overview of the evolution of the scheduler was given and some important implementation details were mentioned, specifically how the scheduler manages its run-queue (in a red-black tree), how it handles priorities and their relation to the nice levels. Finally, it was discussed how [CFS] handles load balancing, a functionality whose current implementation does not guarantee fairness.
Framework for the performance and energy-aware real-time scheduling for heterogeneous embedded systems

Contents

4.1 Framework general overview ................................................................. 28
4.2 Capturing system/application interaction: applications monitoring .......... 29
   4.2.1 Performance counters for application characterization ......................... 30
   4.2.2 Applications that report their performance ........................................ 30
4.3 Performance monitoring Application Performance Interface .................. 32
4.4 Controller ............................................................................................... 36
   4.4.1 Share controller ................................................................................. 37
   4.4.2 Frequency controller ......................................................................... 40
   4.4.3 Migration controller .......................................................................... 43
4.5 Summary ................................................................................................. 45
The execution platforms targeted in this work are heterogeneous embedded systems, which contain several clusters of multi-cores with different microarchitectures that share the same ISA. This means that the resource allocation problem’s complexity is escalated by the fact that the same task can be attributed to resources with varying architecture-specific features, e.g., in-order/out-of-order execution, number of parallel instructions issued, branch prediction mechanisms and/or complexity and capacity of different memory hierarchy levels. As a result, in single-ISA heterogeneous platforms, this augmented diversity of architectural features may provide a great opportunity for improving the energy-efficiency, but it also causes a great challenge for managing the execution of parallel tasks.

In addition, performance of real-world parallel applications is not only limited by these microarchitectural features, but also by application-specific requirements, such as number of memory instructions, integer vs floating point operations, conditional statements and/or application input. In fact, applications may even contain several execution phases with different requirements, thus the achievable performance levels may also significantly vary across different execution periods of the same application. As a result, in order to determine the allocation of available system resources, it is crucial to provide the means for run-time monitoring and modeling of this application-architecture interaction.

In order to achieve the proposed objectives of improving performance fairness of running applications, this chapter presents a specifically designed and general framework for nowadays heterogeneous embedded systems. Through the use of several different integrated functional components, this framework captures, during run-time, the interaction between the running applications and the underlying architecture, thus attaining control over the applications’ performances and energy-efficiency based on both OS-level scheduling and DVFS. The proposed framework tracks and amends essential system parameters and decisions to introduce QoS-awareness to the OS scheduler. These parameters span to several different levels of parallel processing, such as microarchitecture, program characteristics and heterogeneity.

### 4.1 Framework general overview

The framework proposed in this thesis attains its main functionality by interacting with several specifically developed user-space and OS kernel-space components. Figure 4.1 illustrates the general layout of the proposed controlling framework. At the user-space level, it provides the necessary facilities for invoking and monitoring the execution of several parallel applications. In brief, monitoring of multi-threaded applications is achieved via specifically developed beeps subsystem. This subsystem provides the means for very accurate application monitoring by interacting with kernel-space scheduling facilities. In a nutshell, the application reports its current performance (along with performance target and window size) to the Beep API. The performance target represents the desired performance for a specific application (either set by the user or programmer), while the window size represents the amount of beeps (signals send by the application) to consider to average the application performance.

Then, the Beep driver communicates with the OS scheduler to gather the relevant execution times and sends that information to the Controller (see Info in Fig. 4.1). In addition, the controller also col-
lects information from other system drivers, such as the current operational frequency of the device. The Controller represents the key component to achieve the overall functionality of the proposed framework. According to the gathered system and application monitoring information, it determines the right allocations of the tasks, their shares and frequency levels. As a result, the Controller scales the applications’ performances by sending those values back to the different drivers (see Nice, Freq. and Mig. in Fig. 4.1). These drivers interact with the main System driver to enforce the desired applications behavior, by amending the OS scheduler and DVFS parameters. The System driver is composed by the tools included in the system kernel and provides the means to call the system and request both information and changes to parameters such as Nice levels, task-to-core allocations and Frequency.

Figure 4.1: Overview of the Controller, Beep subsystem and System integration

The following sections provide an in depth view on the main functionality behind all the components involved in the proposed task management framework (the details behind the OS scheduler and DVFS are introduced in Chapter 3).

4.2 Capturing system/application interaction: applications monitoring

A crucial part of the proposed control framework is to gather the run-time information and recognize the current application performance during run-time. In fact, gathering the performance of a given application is only part of the challenge, since a model of the application-system interaction must be devised in order to correctly predict the system behavior. Nevertheless, understanding how the microarchitecture of a given system affects the application’s performance, and what that performance translates into, is a complex task, which usually requires certain assumptions to be made based on the intended applicability of the model.

For models intended to a specific system/application pair, usually an extensive offline analysis is performed [9,29], since the behavior is assumed to be constant as it only depends on the inputs. In other modeling circumstances, the microarchitecture characteristics may be known, but characteristics of the running applications are not a priori known. In this case, a study of the type of instructions used in
the program may hint how well the application will perform on the specific architecture [36, 37]. However, the work proposed herein aims at addressing the challenges when specific features of heterogeneous architectures are not known, and when the general program characteristics are also unknown a priori. Hence, only an online observation of a program's performance can be used to provide significant insights about how the application behaves on the currently assigned architecture/core.

As a result, the execution model should be self-adaptable during run-time and require as little changes as possible to the system and the programs structure. Since performance is the key metric to be monitored, all running programs must provide some performance indication. This performance indication can be obtained through the system's performance counters and/or performance reported by the program itself during run-time, depending on which facility is available on the platform/application under analysis.

4.2.1 Performance counters for application characterization

Modern CPU architectures employ performance counters to track a limited set of events in the CPU, e.g., floating point operations, accesses and misses to a given level of cache, and stalls. From these, valuable information can be gathered regarding the application behavior, such as whether a process is computation or memory bounded (via the number of memory accesses), the ratio of floating point operations and missed branch predictions, or even achieved efficiency on a per instruction type basis.

This information may be helpful when deciding which CPU core, in an heterogeneous system, a given application should be assigned. For instance, if one core has higher floating point performance (while maintaining integer performance), a program with a significant number of floating point operations should be assigned to this core and leave the other free cores for programs rich in integer operations [26]. However, even when this information is gathered from the performance counters, the question “What is the target performance?” would still be unanswered. In fact, answering this question is far from being trivial, mostly because there are no two programs that will require the same type or number of instructions to accomplish a given milestone (e.g., a physics simulation requiring floating point instructions or video processing relying on integers).

One of the ways to express a target performance in a methodology based on performance counters is to run the program with a sample input. Then an offline analysis can be performed to instruct the online program mapping, i.e., the number of events triggered by the known sample could be used as a performance reference. However, this approach is mainly valid for the benchmarked representative input and it may not necessarily be applicable for different input sets. Hence, when diverse inputs are considered, another, and more efficient, approach would be to allow the program to report the performance during its run-time, thus guiding its mapping according to the number of triggered events.

4.2.2 Applications that report their performance

Ideally, to accurately capture the application real-time behavior on a given platform, the application should be able to report its own performance to the system in real time. The exact performance metric for such a reporting mechanism is generally irrelevant, as long as the application generates a signal
at meaningful parts of the code, e.g., completion of a frame or the processing of a set of data. In a system with no performance counters available, this reporting mechanism is essential for application monitoring, as it is the only true indication of application performance. Even if the program already reports its performance in real time, the system must receive this information in a standard way, i.e., a common reporting mechanism should be defined. Naturally, this mechanism requires changes at the program level and no indication of specific system resources utilization can be extracted.

To overcome this issue, APIs such as Application Heartbeats [16] were developed, providing a uniform way to report performance across any type of application with minimal program changes. Most systems can also report the percentage of time that an application is in run state, as opposed to waiting for I/O or other external interrupts. This information is usually coupled with the share the application is using in respect to other programs in the system [36]. This means that the share of time that the application uses the system resources can be mapped to its performance, although no information about the specific resources is acquired. In fact, predicting how the micro-architecture may affect application performance in this case is a nearly impossible task, since there is no code profiling involved. Nevertheless, the information about the share of the used resources is extremely valuable and can be used to instruct application performance control.

In this thesis, a specific focus is given to two different classes of parallel applications that report their performance. In particular, in applications from the first class all threads report their performance, as depicted in Fig. 4.2a. For the parallel applications of the second class, only one thread is responsible to report the performance of all the application’s threads (see Fig. 4.2b). In the former case, it is important to note that the performance of the whole application is the sum of the performance of the individual threads while on the latter the reported performance already accounts for all threads. The following paragraphs explain the mathematical concepts behind these types of applications.

In Fig. 4.2 beeps, which are signals generated at the completion of some processing step indicating one performance unit, are indicated as arrows. Application performance is defined by the amount of beeps \( b_a \), each with a value of 1, received from one multi-threaded application \( a \) over a given interval.
of time \( t \), \textit{i.e.}:

\[ P_a = \frac{\sum_i b_{a_i}}{t}. \quad (4.1) \]

Similarly, any thread’s performance \( p_i \) on the same interval \( t \) can be expressed as the amount of beeps generated by that thread \( b_{T_i} \) over the time interval:

\[ p_i = \frac{\sum_i b_{T_i}}{t}. \quad (4.2) \]

It is then possible to express application performance as the sum of the performance of all threads from application \( a \):

\[ P_a = \frac{\sum_i b_{a_i}}{t} = \frac{\sum_i p_i t}{t} = \sum_i p_i, \quad (4.3) \]

where \( b_{a_i} \) represents the beeps of application \( a \) in time period \( t \) and \( p_i \) represents the performance of the threads from the same application. The examples on Fig. 4.2a are merely illustrative, in reality \( t \) must be selected per thread and the thread performance is assumed to be approximately constant so that (4.3) holds.

As was previously said, tracking per thread performance directly is not always possible (see Fig. 4.2b). However, a thread’s performance might be derived from the overall performance. Let \( T_{intB_a} \) be the period \( t \) between beeps of application \( a \) and \( T_{execB_i} \) be the time a thread (in a core) from application \( a \) is running in interval \( t \), \textit{i.e.}, the cumulative run-time corresponding to all the blocks marked “A” in the same core between the dashed lines of Fig. 4.2b. In this case, application performance can be expressed as:

\[ P_a = \frac{\sum_i b_{a_i}}{T_{intB_a}}, \quad (4.4) \]

which in the specific case of Fig. 4.2b translates to \( P_a = \frac{1}{t} \). Since it is assumed that all threads from the same application equally contribute to the performance, the performance per thread \( p_i \) (of application \( a \)) can be express as:

\[ p_i = \frac{P_a T_{execB_i}}{T_{execB_a}}, \quad (4.5) \]

where \( T_{execB_a} \) corresponds to the sum of the run times of all threads (across all cores) of application \( a \) in time \( t \).

4.3 Performance monitoring Application Performance Interface

As previously referred, a crucial step required to implement the precise system control framework is to provide the means for the applications to report their performance and to accurately assess the performance by relying on OS-level scheduling facilities. For this, the Beep subsystem is specifically developed in the scope of this thesis to provide a standardized interface for all applications to report and accurately assess their performance. Although currently existing approaches in the literature, such as Applications Heartbeats \[16\], aim at providing similar functionality, they are generally not tailored to provide detailed and the most accurate OS-level profiling information (\textit{e.g.}, specific execution times, easy
to implement per thread performance report and CPU allocations. The beep subsystem is composed of two main components, namely, the Beep API and the Beep driver, as presented in Fig. 4.1.

This subsystem allows the acquisition of performance data via the Beep driver (based on [38]), which is obtained in real-time from the running applications via the performance reporting Beep API's (common to all applications). Beep API was created to provide an insightful look at the required application data. This is achieved by instrumenting the application code with a single call to the Beep beep(), with no additional parameters but an identifier generated at initialization time. Whenever the task completes a representative computational block, denoted by an sbeep_beep() call, an application performance event is generated, i.e., beep. The API is responsible for communicating the triggered beep to the Beep driver, which keeps the beeping and timing records for all the monitored tasks. This communication between the Beep API and the driver is made through an ioctl() system call. Furthermore, the API is also responsible for registering each application (at initialization) and transmitting the data stored in the driver to the task management controller.

The Beep driver is the low-level component responsible not only for keeping track of each target task’s scheduling path and beeping information, but also to provide the required input parameters to the controller and to handle the output requests from the controller. Since the driver resides at the OS level, it allows finer control over the tasks execution. This is achieved by directly detecting OS scheduling events (e.g., forks, migrations and scheduling switches) as it is described further below.

Tasks registration is made on a per-application basis and the leader task (from where other threads will spawn) must be registered into the driver by using the previously mentioned ioctl() command. When registering an application (or task group), a configuration structure (sbeep_attr) must be sent to the driver. This structure must contain the PID of the leader task, the application type, the sliding window size and the target performance for that application. Moreover, all the tasks descending from a registered leader task are automatically registered into the framework driver. This is accomplished upon detection of fork events performed by the registered tasks.

Figure 4.3 illustrates the main data structures used by the driver in order to keep track of the monitored tasks. The pid_table contains 32768 entries, where each entry key corresponds to a task PID. Whenever a new task is registered, a new sbeep_task is initialized and the corresponding PID entry in the pid_table is set as a pointer to the sbeep_task. The use of the pid_table does not only speed up the process of finding whether some specific task is registered or not, but it also allows direct access to the required sbeep_task struct. This is specially important for CPU scheduling events, which frequently occur and, therefore, a higher overhead when accessing the data structure might provoke a higher impact on the whole system performance.

Two different types of tasks are identified and registered by the driver, namely: i) leader tasks are registered whenever the framework is initialized; and ii) child tasks are registered automatically by the driver upon being forked from a leader tasks or whenever a task descends from a leader task. Therefore, the leader_list double-linked list (see Fig. 4.3) contains all the registered leader tasks that are currently active. Each leader task contains a list head to a group_list double-linked list, which connects all the tasks belonging to the same application.
Whenever a read request is made to the driver through the provided API functions, the leader_list is iterated and the timing and beeping information for each individual task is copied to the Controller space. The information is sent by using the OS memory calls `copy_from_user()` and `copy_to_user()`, which allow copying the data structures that contain the required execution information from the user-space to the kernel-space and vice-versa.

The communication data structures are illustrated in Fig. 4.4. Each time a read request is made to the driver, an `sbeep_rd_comm` structure is sent to the user-space, which contains the number of monitored task groups and an array containing each group information (`sbeep_rd_group`). Each `sbeep_rd_group` contains the PID of the group’s leader task, the beep window size, the target performance, the application type, the number of active tasks within that group and an array of `sbeep_rd_task` structures, containing the information related to each individual task. The `sbeep_rd_task` structure contains the PID of the task, the CPU where the task is currently running, the number of beeps since the task was registered, the window time interval (i.e., the wall time between the last `window_size` beeps), the time that the task was actually running during the `time_interval` and the timestamp of the last beep.

As previously referred, the beep information is calculated within the driver each time a beep request is sent from the application (through the beep API). Whenever a beep request is made, the driver updates the beep information of the corresponding task or all the tasks belonging to that task’s group. This information is updated depending on the application type i.e., when a leader task is responsible to report overall performance, Fig. 4.2b (if all tasks share the same beep window) or when individual
threads report their own performance, Fig. 4.2a (each of them has an individual window).

In order to obtain the accurate timing information within each task's window, the framework driver interacts directly with the OS scheduler, therefore allowing to detect the exact time when each monitored tasks is scheduled in or out from a specific core. In addition, it is also possible to detect forks, task termination and CPU migration. The interaction between the driver and the OS scheduler is made through system tracepoints. Tracepoints are present in different parts of the OS's code and allow one to register a custom callback function, which is called whenever the OS reaches a specified point of the execution. The Beeps driver makes use of four main scheduler tracepoints, thus allowing tracing the complete scheduling path of each monitored task:

- **sched_switch()** - tracepoint triggered whenever a task is scheduled into or out from a CPU core. Thus, it provides the accurate timestamps for the time scheduling path, which allow keeping track of each task's execution time;

- **sched_migrate_task()** - triggered each time a task migrates from one CPU core to another. This allows keeping track of where in the architecture each of the monitored tasks is currently running;

- **sched_process_fork()** - triggered whenever a new task is created. This tracepoint is used for registering tasks descending from previously registered tasks and it marks the newly created task's execution start;

- **sched_process_exit()** - triggered whenever a task terminates. This tracepoint is used to detect the end of the execution of the monitored tasks.

These tracepoints perform different functions within the Beeps driver. While sched_migrate_task(), sched_process_fork() and sched_process_exit() are used to keep track of the existing target tasks and to mark the beginning and end of their execution, the sched_switch() allows keeping track of the accurate execution time of each task. The execution time of a task can therefore be derived by relying on the beep timestamps (i.e., the timestamps for each sbeep beep() call) and on the sched_switch() timestamps, allowing the driver to extract the exact time a thread spent in execution state \( T_{execB_i} \) within the \( T_{intB_i} \) interval (i.e., the time interval between two sbeep beep() calls).

Finally, the data acquired through Beeps must be gathered and stored in the Controller in a structure which must provide easy access to several of its elements (e.g., threads per core, global application performance and threads per application). The structure is similar to that used by the controller but a duplication of some information is required to easily manipulate the stored information in the user-space. Figure 4.5 graphically illustrates this structure. Every thread in a core is connected through a double linked list, the same goes for every application and for every thread in an application. With this organization, going through the required data is a straightforward process, since there is a dedicated list for applications, threads in a core and threads in an application. Furthermore, to further optimize the access times to the required data, indexed lists (by process/thread ID) are maintained in memory, meaning that searches can be done in \( O(1) \) time.

35
4.4 Controller

The controller represents the key functionality of the proposed framework. It relies on the application profiling information from the Beep driver to provide system-wide decisions regarding temporal and spatial allocation of the running parallel tasks across the available heterogeneous computing resources. In detail, the Beep subsystem functionality is used herein to express the performance $p_i$ (in beeps per second) of each currently running task $i$. In addition, it is also used to express application/core execution affinity, as well as to experimental assess the performance degradation when several simultaneously co-scheduled applications compete for the system shred resources. The controller then compiles this information with the OS scheduler and DVFS parameters, in order to orchestrate the execution of parallel tasks such that their performance targets are respected by optimizing the utilization of system resources.

As presented in Fig. 4.6, the proposed performance-aware controller incorporates three different controlling mechanisms in order to achieve its functionality, namely: The Share controller, the Frequency controller and the Migration controller. Each of these controllers has a different scope from the system perspective, where the Share controller is applied at the core-level, Frequency controller at the cluster level and the Migration controller has a system-wide applicability. The Share controller, which is the main and the most frequently called controller, aims at determining the CPU time share $s_i$ for each task $i$ in the core. This step also assures performance fairness among the running tasks on the same core by equalizing the relative distance between the realistically attained and the target performance ($p_d$) for all active tasks.

With a larger granularity, the Frequency scaling step is applied at the cluster level to minimize the assessed relative performance distances by automatically scaling the operating frequency $f$, such that all the achievable applications’ performances ($P_a$) are as close as possible to their target performance levels ($P_{d_a}$), i.e., $P_a \approx P_{d_a}$, without affecting the previously attained fairness. It is important to note that on heterogeneous platforms where frequency controls the migration between clusters, this step will also enforce such migrations when required. To maintain balance among running applications, a Migration controller is also applied at the system level with an even sparser invocation period. The functionality of
each of these controllers is explained in the further text.

Since the proposed controlling mechanisms are aimed at run-time adaptation of the application/system execution they must be lightweight in terms of functional complexity and introduces overheads. Hence, the adopted modeling concepts do not need to capture all possible architecture details nor application complexity, but they should be insightful enough to allow adequate shared resource usage analysis and allocation. This information can then be used to ensure that an application’s performance targets are achieved. In order to provide adaptive decisions during the application run-time, the following modeling step is of the utmost importance and should be as insightful as possible while reducing the computational complexity and overheads introduced by the controller. In detail, the main objective of the proposed approach is to determine the CPU share ($s_i$) and operating frequency ($f$), such that the given performance target ($P_{da}$) is achieved for the running application $a$.

For this, the control mechanism considers performance to be proportional to both CPU share and frequency ($P \propto s$ and $P \propto f$). To capture the influence of architectural and application characteristics on the attainable performance, an application-specific parameter ($c_i$) is introduced and it is regularly updated by sampling the application performance data at every controller update. More specifically, $c_i$ represents the relative measure used to describe the performance behavior of a task $i$, such that the task’s performance can be expressed as $p_i = c_i s_i f$. In order to simultaneously approach the target performance levels ($P_{da}$) for a set of parallel tasks, the proposed mechanism automatically adjusts their CPU time shares ($s_i$) by relying on the dynamically acquired $c_i$ parameters and by setting the nice levels $n_i$ (see (3.1)) and operating frequency $f$.

### 4.4.1 Share controller

The first step of the controlling method determines the CPU share ($s_i$) for each parallel task ($i$) per core, such that their realistically achievable performance is as close as possible to the target performance. In addition, this step also assures the performance fairness among the running tasks by equalizing the normalized differences between their achievable ($p_i$) and target performance ($P_{da}$). At this stage it is important to note that performance fairness can only truly be achieved at the application level if the concurrent applications are all using the same cores. An example of a case where it is possible to
have fairness among applications is depicted in Fig. 4.7a, where two applications spawn four threads and each core shares two tasks from different applications. On the other hand, Fig. 4.7b shows an example where four dual-threaded applications are running and where fairness is only achieved among different application pairs, i.e., Application 1 + Application 2 and Application 3 + Application 4 (note the increased error on the 3+4 pair in Fig. 4.7b).

![Figure 4.7](image)

**Figure 4.7:** Cases where it is a) possible and b) unlikely to have performance fairness across all applications

Figure 4.8 provides the graphical interpretation of this idea for three different tasks. As can be observed, the previously assessed task performance (represented with stars) greatly mismatches the target performance levels (marked with lines). However, after applying the *Share controller*, the newly achieved application performance (represented with dots) is not only inline with the desired performance targets, but also the normalized difference is equalized among the applications (see the relative difference between lines and dots in Fig. 4.8).

![Figure 4.8](image)

**Figure 4.8:** Graphical interpretation of the share calculator, 3 tasks are brought to within the same error margin

In order to provide analytic tractability behind the proposed approach, the *Share controller* relies on a set of dynamically assessed task-specific performance parameters \( c_i \) and previously referred application modeling strategy (see Section 4.4). In detail, to quantify the expected task performance for a fixed operational frequency, the \( c_i \) parameters are assessed according to the most recent task sampling information, such that \( c_i = \frac{p_{curr,i}}{s_{curr,i}} \), where \( p_{curr,i} \) and \( s_{curr,i} \) represent the currently attained performance and share of task \( i \), respectively. It is worth emphasizing that both \( p_{curr,i} \) and \( s_{curr,i} \) parameters are reported by the application and measured in the system.
For a set of $N$ parallel tasks, the problem tackled by the Share controller can be formalized as follows:

$$
\begin{align*}
\min_{s_1, \ldots, s_N} & \quad J(s_1, \ldots, s_N) = \sum_{i=1}^{N} (1 - \frac{c_i s_i}{p_{d_i}})^2 \\
\text{s.t.} & \quad \sum_{i=1}^{N} s_i = 1 \\
& \quad \frac{c_i s_i}{p_{d_i}} = \frac{c_j s_j}{p_{d_j}} \text{ for } i, j \in \{1, \ldots, N\},
\end{align*}$$

(4.6)

where $s_i$, $p_{d_i}$ and $c_i$ represent the target share, the desired performance and the estimated behavior constant for task $i$, respectively. When minimizing the normalized performance difference of all running tasks, i.e., $1 - \frac{p_i}{p_{d_i}} = 1 - \frac{c_i s_i}{p_{d_i}}$, this controller relies on the normalized performance measure in order not to break the linearity and a unitary value will mean the performance target is achieved. Furthermore, the problem expressed in (4.6) is accompanied by two additional restrictions, which imply that the sum of all per-task shares $s_i$ must be equal to one (i.e., to guarantee utilization of all CPU resources) and that the normalized difference is equalized among the running tasks (which guarantees that a second linear scaling can be applied and also implicitly assures non-negativity of per-application shares).

**Share controller implementation**

For each core with more than one allocated thread, the Share controller provides the solution for the problem presented in (4.6). For this, it is firstly needed to obtain a set of expressions that can achieve a solution to this convex optimization problem. The Karush–Kuhn–Tucker (KKT) conditions [39] can be applied to this problem in order to obtain a closed expression for each thread's share ($s_i$) as follows:

$$
\begin{align*}
\lambda &= \begin{cases}
-2 N \prod_{j=1}^{N} (c_j) \left( \prod_{j=1}^{N} (c_j) + Z(c, p_{d_i}) \right) \\
2 N \prod_{j=1}^{N} (c_j) \left( \prod_{j=1}^{N} (c_j) + Z(c, p_{d_i}) \right) \\
& \quad \frac{Z(c, p_{d_i})}{X(c, p_{d_i})} \text{ for } i \neq 1
\end{cases} \\
Z(c, p) &= -\sum_{j=1}^{N} \left( \prod_{k=1, k \neq j}^{N} (c_k) p_{d_j} \right) \\
X(c, p) &= \sum_{j=1}^{i-1} \left( (N - i - 1) \frac{p_{d_j}}{N c_j} \right) + \sum_{j=i}^{N} \left( -(i - 1) \frac{p_{d_j}}{N c_j} \right)
\end{align*}

(4.7-4.11)

$N$ shares (total number of threads in the core with $i = 1, \ldots, N$) must be calculated, using as inputs the task specific performance parameters ($c_i$) and target performances ($p_{d_i}$). Although the expressions may seem complex, it is important to realize that a significant part of the computations can be re-utilized, thus reducing their complexity. Both $p_{d_i}$ and $c_i$ are thread-specific parameters (as opposed to global application parameters such as $P_{d_a}$), but both are easily obtained from the data stored in the controller’s
data structure. To get \( c_i \) one must calculate the performance and share per core, based on the time interval between Beeps (\( T_{intB_i} \)), the window size (\( W_{s_i} \)) and on the actual execution time between Beeps (\( T_{execB_i} \)), as the following expression illustrate:

\[
p_i = \frac{W_{s_i}}{T_{intB_i}}; \quad s_i = \frac{T_{execB_i}}{T_{intB_i}}; \quad c_i = \frac{p_i}{s_i}; \quad p_i = \frac{W_{s_i}}{T_{execB_i}}.
\]  \hfill (4.12)

As referred in Section 4.2.2 the overall application performance is the sum of the performances obtained in all cores. Hence to achieve a target performance per core (\( p_{da} \)) one must scale \( p_i \) by the global application performance error \( P_{da}/P_a \) where \( P_{da} \) is the target performance and \( P_a \) is the current performance of application of \( a \). For the second application class, where the application’s performance is reported by only one thread (although several of them may be contributing for it), certain adaptations must be made to the previous expressions, such that:

\[
P_a = \frac{W_{s_a}}{T_{intB_a}}; \quad s_i = \frac{T_{execB_a}}{T_{intB_a}}; \quad c_i = \frac{p_i}{s_i}; \quad p_i = \frac{P_a T_{execB_a}}{T_{execB_a}}.
\]  \hfill (4.13)

It is important to notice that in this case some of the values refer to application times (\( a \) index). In this case it is considered that all threads (possibly spanning across all system cores) equally contribute to the same beep. This means that \( T_{execB_a} \) is the sum of the execution time of all threads (from all cores) off application \( a \) inside the beep window \( W_{s_a} \). \( T_{intB_a} \) is the wall time between the first and last beep (from application \( a \)) inside the window \( W_{s_a} \).

**Nice conversion** In order to practically apply the previously calculated \( s_i \) shares to the running threads in the core, they must be first translated to the CFS nice values. However, by directly applying (3.1) (see Section 3.3.1), the translation between share and nice values may be practically infeasible. In order to derive a set of nice values (\( n_i \)) for each running thread, the following procedure is adopted:

\[
\frac{s_{i-1}}{s_i} = \frac{1.25^n_i}{1.25^{n_{i-1}}} \Leftrightarrow n_i - n_{i-1} = \frac{\log\left(\frac{s_{i-1}}{s_i}\right)}{\log(1.25)}.
\]  \hfill (4.14)

The nice conversion considers the ratio of share values across different threads, in order to calculate the \( n_i \) nice value for each thread \( i \). An additional (needed) constraint is provided by ensuring that the nice value of the highest priority task is as close as possible to the nice level of 0 (i.e., system default). Given the restriction that the nice values must belong to a predefined integer interval \([-20, 19]\), the obtained values are further rounded to the nearest integer values.

In practice, the controller first calculates the range of nice values it needs and then chooses one of the following three options: 1) if the range is smaller or equal to 19, it sets the highest priority task’s nice value to 0; 2) if the range is between 19 and 39, it sets the highest priority task’s nice value to \( 19 - \text{nice range} \); and finally 3) if the range is greater than 39 (note that this means great share discrepancies in the core), it sets the highest priority task’s nice level to -20 and scales the other values to fit on the correct interval.

### 4.4.2 Frequency controller

The Frequency controller of the proposed framework assumes performance linearity with frequency \( p_i = c_i s_i f \). As previously referred, the Share controller guarantees the equalization of the percentual
difference between the predicted performance and the target performance for a set of running parallel tasks. However, depending on the currently obtained value for percentual difference, the Frequency controller aims at adjusting the system operational frequency \( f \), such that all applications can achieve their target performance. It is worth to note that for a set of applications whose work is not equally distributed among the cores, it cannot be guaranteed that the performance of all tasks is on the target performance (due to the available discrete combinations). However, it is possible to guarantee that all feasible sets of performances values (i.e., the application performances which are realistically attainable with the current system configuration) can never be bellow the desired target performance. Nevertheless, this still leaves room to achieve significant energy savings when dealing with programs that ask for feasible performance values. Figure 4.9 graphically represents this idea, where it can be observed that the performance targets (lines) for all three applications can be achieved by decreasing the operational frequency (see arrow in Fig. 4.9). As a result, the previously assessed performance levels (dots) are scaled, such that newly obtained performance for running applications (triangles) closely matches the desired performance targets.

The Frequency controller adjusts the operational frequency \( f \) by relying on a set of dynamically accessed application-architecture interaction parameters \( (c_i) \), the previously estimated shares \( (s_i) \) and the desired performance \( (P_{da}) \) for all running parallel applications. In detail, the calculation of the new frequency level \( (f_n) \) is conducted by considering the above-referred parameters and the previous operating frequency \( f_o \) at which they are obtained, such that:

\[
f_n = f_o \frac{P_{da}}{P_a}.
\]

\( P_a \) is the current estimated application performance, obtained by adding together all the contributions of the individual threads, i.e., \( P_a = \sum_j c_j s_j \) where \( j \) is a thread belonging to application \( a \). Typically, systems only allow setting the operational frequency from a predefined range of discrete values. As such, the solution obtained from (4.15) is further refined by rounding the newly calculated \( f_n \) frequency to the nearest value available in the system. Furthermore, when considering heterogeneous asymmetric systems composed of a set of different cores (which are typically organized in clusters), multiple application-specific performance parameters \( (c_i^{coreA}, c_i^{coreB}, \cdots) \) can be considered, one per core type, and adjusted in real-time. To allow application migration to a different core, one must evaluate the expected performance drop/increase on the other core, by assuming an application specific relation

\[
e_i^{A/B} = \frac{c_i^{coreA}}{c_i^{coreB}}
\]

between performance parameters.

In the case of single-ISA heterogeneous platforms using cluster migration, the Frequency controller
is also responsible to govern the migration. This is done by assigning a set of frequencies to one cluster (from the available discrete range), as well as another set of frequencies to another cluster. Furthermore, these frequencies may be merely virtual, i.e., the real frequencies at which the cores operate may be different. This creates an implicit performance relation among the cores, where a frequency (and consequently cluster migration) decision immediately takes into account performance changes due to cluster migration, even though this value is not tested for specific applications and represents only a global estimation. Figure 4.10 illustrates such a scaling, where a set of lower virtual frequencies (250 MHz-600 MHz) is used to instruct the execution of threads on a Cortex-A7 cluster (at a real frequency which is twice the virtual frequency). Another higher set of frequencies (800 MHz-1.6 GHz) imposes the tasks to run on the Cortex-A15, cluster at a real frequency which is the same as the virtual one.

![Figure 4.10: Range of virtual and their correspondent real frequencies for a heterogeneous platform (Exynos 5410 on the Odroid-XU+E) using cluster migration](image)

**Frequency controller implementation**

The **Frequency controller** is an essential component of the proposed framework in the sense that it is a critical element to achieve energy savings. It achieves this by lowering the system frequency whenever all applications report the performance higher that the desired target until the performance of all applications is at or above the target performance. The adopted implementation of this controller is summarized in Algorithm 4.1. The **Frequency controller** is only activated after every FREQ_WAIT (an integer) times the **Share controller** is used (line 1). This reduces the instability caused by performance misspredictions when both the **Share and Frequency controller** are simultaneously used. The practical aspects of changing this value will be further discussed in Chapter 5.

**Algorithm 4.1 Frequency update algorithm.**

1: if The shares where updated FREQ_WAIT times then
2: \( \text{scaling factor} = 0 \)
3: \( \text{old frequency} = \text{Get old frequency}() \)
4: for each application do
5: \( \text{if application performance error} > \text{scaling factor} \) then
6: \( \text{scaling factor} = \text{application performance error} \)
7: end if
8: end for
9: \( \text{Set new frequency}(\text{scaling factor} \times \text{old frequency}) \)
10: end if

The first step in the algorithm is to initialize \( \text{scaling factor} \) (line 2) and obtain the previous system
frequency \textit{old\_frequency} (line 3). The \textit{Frequency controller} then searches (lines 4-8) through every application to find the one with the greatest application\_performance\_error \((P_d/P_a)\). The choice to use the lowest performing application to calculate the new frequency means that, depending on the performance distribution of the applications, all their performances will be as close as possible or above their performance targets. Then, at line 9, the system frequency is updated by calling \textit{Set\_new\_frequency()}, which applies a new frequency of \textit{scaling\_factor*old\_frequency} to the system. Before this frequency is applied it is first rounded to the closest discrete frequency that belongs to the set of virtual system frequencies. Furthermore, if the newly calculated frequency corresponds to the previous operating frequency, the system driver will not be called to prevent unnecessary overheads.

\subsection*{4.4.3 Migration controller}

The last component of the proposed Controller is the \textit{Migration controller}. This controller is responsible for periodically checking whether threads should be migrated across cores and it aims to improve the resource allocation and application performance, such that the workload of running processes is better balanced among the cores. A simple example is illustrated on Fig. 4.11 for the case when 4 different processes are running in the system (all from different applications and with different performance targets). In detail, process 1 has 3 threads, which are assigned to core 0, 1 and 2. The other processes (\textit{i.e.}, processes 2, 3 and 4) are single threaded applications and are initially assigned to different cores to balance the number of threads per core (\textit{i.e.}, cores 3, 0 and 1 respectively). Furthermore, process 1 requires a relatively higher performance that processes 2, 3 and 4. With the distribution depicted on the left side of Fig. 4.11 the performance targets may be compromised since the resource allocation scheme causes very high utilization of cores 0 and 1, while core 3 is underused. In fact, cores 0 and 1 are not capable of simultaneously satisfying the demands of the allocated processes. The benefit of this controller can be visualized in the right side of Fig. 4.11. The proposed controller migrates all low performance threads from different applications to core 3. As a result, the utilization of all cores is more balanced and the desired performance levels are respected for all running processes.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig4_11.png}
\caption{Migration controller concept. Bars represent the required core share to achieve the target performance on a given thread (belonging to a process) and the dashed line the limit a core can provide.}
\end{figure}

One important aspect of the \textit{Migration controller} is that it may attenuate the previously referred \textit{Share controller} issue when the performance fairness can only be achieved at the level of individual application groups, depending on their mapping to the cores (see Fig. 4.12a). As can be observed, in Fig. 4.12b the migration controller might provide the allocation where the threads from all applications are directly
or indirectly coupled with each other (e.g., threads from application 3 are coupled with threads from application 1 and 4 in different cores). As a result, the other controller components (i.e., Share and Frequency controller) might have a better opportunity to derive solutions that yield better performance fairness among all running applications.

Figure 4.12: a) An example with unbalanced application groups and b) an example of how the migration controller may improve the fairness achieved

Migration controller implementation

Handling task migration is one of the most sensible aspects of the whole controller, since it provides the means to control the execution of applications with different numbers of threads. Algorithm 4.2 provides the general overview of the basic functional principle behind the Migration controller. The algorithm starts by initializing a per core resource utilization list at zero (lines 1-3). This list is later required to find the least loaded core to assign threads to. Then, a list of all the threads in a system is compiled and ordered according to thread_share*application_performance_error (line 4). Since it is assumed that performance linearly scales with share (Section 4.4), this list represents a measurement of how much resources these threads require to achieve their target performance. After this process, threads are orderly selected (line 5) from the list (by descending order of resource requirements) and assigned to the least loaded core (line 6). The core usage is then updated according to the previous estimation thread_share*application_performance_error (line 7) and the core list is reordered (line 8).

Algorithm 4.2 Thread migration algorithm.

```
for each core i do
  Initialize entries of the resource usage list used_core_resources[i] = 0
end for

Sort the applications threads accordingly to thread_share * application_performance_error

for each thread in the previous sorted list do
  Assign thread to the core with least used resources (i)
  Update used_core_resources[i] with thread_share * application_performance_error
  Sort the resource usage list (used_core_resources)
end for

Trigger the share controller
```

The core utilization list’s values (used_core_resources) may reach numbers greater than 1
asking for more than 100% utilization). To guarantee fairness and realistically attainable performances in the cores after a migration, the Share controller is triggered at all balancing events among all cores (line 10), independently of whether or not there was a recent application performance update. However, this may create serious issues in applications with different phases, e.g., if the previous element to be processed achieved high performance but the next one will achieve a low one, meaning that estimating it to be high would possibly kill the program.

To overcome this issue a safeguard was added when estimating the performance for migration: if the time interval between the last actual performance report is largest than the previous interval between reports (see T.z 1 and T.z 2 in Fig. 4.13), the performance is estimated to be lower and equivalent to what is achievable on the largest period. Mathematically this translates to \( P = \min(P_{PREV}, \frac{W_s}{T_L}) \) where \( T_L \) is the time period between the last performance report and the current migration event and \( W_s \) is the selected window size for that process. Figure 4.13 illustrates the two possible situations: 1) task \( z \) (in Core 0) would have its performance dropped (\( T_{z2} > T_{z1} \)); while 2) task \( y \) (in Core 1) would not (\( T_{y2} < T_{y1} \)). Furthermore, the controller may effectively serialize the applications if the current performance does not require parallelism (i.e., if a dual threaded application requires each thread to have less than 50% core share to achieve the target performance, both threads can be run on the same core).

4.5 Summary

This chapter presented a framework capable of controlling the application/system interaction in nowadays heterogeneous embedded systems. Through the use of several different integrated functional components, this framework captures, during run-time, the interaction between the running applications and the underlying architecture, thus attaining control over the applications’ performances and energy-efficiency through both OS-level scheduling and DVFS. The proposed framework tracks and amends essential system parameters and decisions to introduce QoS-awareness to the OS scheduler.

This framework is composed by several blocks, each accomplishing a different task. The interaction between application and system must first be captured through the use of the Beep subsystem, a functional block which tracks applications to report their share and performance. The acquired data is
then transmitted over to the Controller. The Controller represents the key functionality of the proposed framework. It provides system-wide decisions regarding temporal and spatial allocation of the running parallel tasks across the available heterogeneous computing resources, in order to achieve performance fairness and energy-efficiency.

The Controller integrates three sub-controller blocks, which are the Share, Frequency and Migration controllers. The Share controller calculates the shares for each task in a core such that their realistically achievable performance is as close as possible to the target performance. The Frequency controller complements this functionality but further extends it with the capabilities to reduce the overall system performance (if the performance targets of the running applications allow it) in order to save energy. Finally, the Migration controller aims at balancing the threads on the system such that their performance is globally attainable and evenly split among cores.
5
Proof of concept and experimental Evaluation

Contents

5.1 System behaviour simulation ............................................ 48
5.2 Controller’s triggering .................................................. 52
5.3 Benchmarks ............................................................... 54
5.4 Experimental setup ...................................................... 57
  5.4.1 Characterizing the Platform: Cortex-A7 vs Cortex-A15 .......................... 58
5.5 Experimental results ..................................................... 61
  5.5.1 Evaluation of application-wide Share and Frequency controller .................. 62
  5.5.2 Evaluation of the proposed full framework for performance and energy-aware real-time scheduling for heterogeneous embedded systems .......................... 65
5.6 Summary ................................................................. 69
In order to validate the efficiency of the proposed framework for performance and energy-aware real-time scheduling for heterogeneous embedded systems, an extensive evaluation was conducted in both a custom developed Matlab simulator and a real embedded ARM big.LITTLE platform. The main goal of the simulation-based evaluation was to assess the efficiency of the proposed methods in a stable environment, without any interference and performance/power consumption fluctuation caused by other system tasks and actions.

Based on the simulation results, the main components of the framework were calibrated and experimentally tested in a real heterogeneous asymmetric embedded system, i.e., Odroid-XU+E platform based on an ARM big.LITTLE processor with two clusters of quad-core CPUs (high performance A15 and low-power A7). To experimentally assess the functionality of the framework, the evaluation considered several multi-threaded applications from standard benchmark suites, since their behavior is well known and studied. This evaluation is also important to gain insights on the hardware platform behavior, specially regarding the performance and energy consumption of both the Cortex-A7 and Cortex-A15 cores. To accomplish this, a set of micro-benchmarks was developed to assess the access times to different memory levels and the time it takes for the system to change frequency and migrate the tasks across different clusters of cores. Furthermore, the energy consumption was tested by running the selected benchmark applications on both types of cores and by gathering the power consumption readings. This evaluation is extremely important to realistically assess the achievable benefits that the proposed framework can achieve while handling heterogeneity.

5.1 System behaviour simulation

In a real system, several external parameters may contribute for the performance of a task management framework (e.g., current system utilization by uncontrolled applications). The simulation-based evaluation is highly important to understand how the proposed controller (or its main functional parts) affect the system applications. Furthermore, this evaluation represents the exploratory study in the process of designing the real-world implementation of the framework. For this, a system simulator was built in Matlab, which focuses on demonstrating and evaluating the capabilities of the Share controller, which is the most critical controller of the proposed framework. As it is shown herein, this evaluation also provided very insightful guidelines for an efficient implementation of the Share controller. The Matlab simulator is based on the following execution concepts:

- There are two types of applications: one in which all threads report their performance (typically by splitting equal work portions among the threads), and another where a parent thread reports the performance of several parallel child threads, where it is more likely that each thread is preforming a different type of work;

- To deliver a single unit of performance, each thread (or the whole process) must spend a certain amount of time to process the input data. As a result, performance is expressed in work unit/s;

- In order to simulate the behavior of real applications, the amount of work performed by each thread
to achieve a performance unit may vary during application execution;

- In order to simulate the real behavior of the system, the share attributed to a task may slightly differ from the share requested by the controller (e.g., this situation may occur in the real system if an uncontrolled task briefly runs on the same core as the controlled task);

Building upon these, a simulation model was devised, where the first two assumptions are used to model the system and the latter two to introduce noise and realism to the simulator. The important questions this model must answer are: “Can performance fairness be achieved?” (as presented in Section 4.4.1) and “How often can the shares be updated?” (since the controller may be faster than the application reporting the performance).

The model, used to get insights on the efficiency of the Share controller, was built in Matlab to simulate the execution of tasks in a multi-core system. OSs handle different tasks by regularly switching among tasks on a run queue. This is done very fast so that it is transparent to the user, but, on each core, the system only handles one task at a time. This means that a task may have to be scheduled in and out of execution several times to accomplish a goal, i.e., it must accumulate some processing time to produce a given result (performance unit). Based on this premise, the produced units are differently reported according to their application classes. When all tasks from a certain application report performance, the exact execution intervals are accumulated on a per task basis, until the overall accumulated time is higher than or equal to the predefined amount of time to produce a single performance unit. For applications where a single task reports their performance, the time intervals are accumulated across all running tasks from that application.

Using these time concepts, a current performance value is gathered \((p_{\text{curr}})\), as well as current share on the core \((s_i)\). In addition to the previous values, which are gathered during the simulation, performance targets \((P_d)\) for the simulated applications were provided. According to this set of data, the share controller was invoked to recalculate the shares based on the approach presented in Section 4.4.

While the CPU share of an application can be read at any time interval, the current task performance \((p_{\text{curr}})\) is only periodically updated. However, the controller may be called at any time, even before the performance is updated, meaning that three different approaches are possible:

1. the controller is activated whenever any thread in the system reports its performance to simultaneously update the shares of every other thread on the system and estimate the performance accordingly;

2. the controller waits for at least one thread on a single core to report performance to simultaneously update the shares of all threads on that core and estimate the performance accordingly;

3. the controller waits for all threads on a single core to report performance to simultaneously updates the shares of the threads on that core;

Each presented technique for acquiring the performance has its benefits and drawbacks. While the first option provides a short delay between updates, possibly leading to fast performance corrections, performance may fail to stabilize, since any small performance fluctuation will have great repercussions.
on the threads whose performance is being estimated (based on an old performance reading). The second case provides a compromise where a larger delay between updates is expected, but stability is improved, since estimating only the performance of threads in a core provides a more realistic view of the global performance. However, it may be possible that the phase of one thread suddenly changes and its performance drops unexpectedly. Constantly using and old (high) performance estimation may lead to a significant performance degradation on such a thread. Finally, the third case will provide the greatest delay between share updates, but will base its decision on the most recent performance data from all threads in a core. Although it may update the shares in individual cores independently, the fact that it does not base its decision in any estimated data should provide the most reliability.

These three approaches were tested with the two previously introduced types of applications, where either all threads from an application report their performance or a parent thread is responsible for communicating the global performance. First, the case where all threads from an application report their performance was tested with the three update methods in Fig. 5.1, 5.2 and 5.3 respectively.

Figure 5.1 illustrates the case where applications where all threads report their performance follow the first approach. As depicted in Fig. 5.1a, constantly estimating the performance across threads and cores results in a highly erratic behavior. This behavior results from the highly fluctuating nice levels applied to threads in a core (see Fig. 5.1b). This instability appears due to the fact that the real performance of a thread in any core is not allowed to stabilize before a new update is enforced. Thus, the performance readings obtained from the threads are never truly accurate.

The second approach to update the cores and estimate performance is depicted in Fig. 5.2. In this case stability is obtained and the global application performance approximately follows its target (Fig. 5.2a). However, the nice levels may have significant differences across tasks in the same core (Fig. 5.2b). This difference in nice levels might be problematic on a real system since it may significantly reduce the performance of a critical thread in a core, stopping the execution of the parent process (i.e., application).

Finally, the results of using the third approach to update the cores and estimate performance are shown in Fig. 5.3. Although this approach takes the longest time to update performance (Fig. 5.3a), it achieves the most stability. The nice levels in a core are maintained for a longer period of time and do not significantly differ (Fig. 5.3b), which reduces the chance of creating an oscillatory behavior between threads' performances across cores.

<table>
<thead>
<tr>
<th>Controller</th>
<th>Application deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 HB/s</td>
</tr>
<tr>
<td>Target</td>
<td>1</td>
</tr>
<tr>
<td>Approach 1</td>
<td>0.8716</td>
</tr>
<tr>
<td>Approach 2</td>
<td>1.1279</td>
</tr>
<tr>
<td>Approach 3</td>
<td>1.0775</td>
</tr>
</tbody>
</table>

Table 5.1 summarizes the average application performances deviations (all threads reporting performance) on the last 10 s of simulation for all three approaches and the selected target performances.
Figure 5.1: a) Performance (real, R.P. and estimated, E.P.) of the 4 simulated applications (all threads reporting performance) with approach 1, and b) Nice levels of the two simulated threads on core 2.

Figure 5.2: a) Performance (real, R.P. and estimated, E.P.) of the 4 simulated applications (all threads reporting performance) with approach 2, and b) Nice levels of the three simulated threads on core 1.

Figure 5.3: a) Performance of the 4 simulated applications (all threads reporting performance) with approach 3, and b) Nice levels of the three simulated threads on core 1.
Although the average performance does not capture the amplitude of the observed oscillations, it was verified that all approaches approximately equalize the performance of the running applications. However, the first approach has the greatest performance deviations, which coupled with its erratic behavior (Fig. 5.1) immediately excluded it from being implemented on the real system.

The same three approaches where then tested with applications where only one thread reports the global performance (Fig. 5.4, 5.5 and 5.6). Since this type of application simultaneously updates the performance across all the application's threads, no dramatically erratic behavior is present (as was the case with approach 1 before). However, both approach 1 and 2 (Fig. 5.4 and 5.5 respectively) present significant nice level differences on the same core (see Fig. 5.4b and 5.5b). The implications of this on a real system is that the threads with higher nice levels (i.e., less CPU time) will effectively be killed. In applications where only one thread reports performance it is highly likely that all threads must accomplish some work before a new performance unit is ready. Killing one of such threads would practically stop the whole execution.

In applications where only one thread reports the performance, approach 3 (Fig. 5.6) showed to be once again the most stable. Although it may take more time to update the thread shares, and consequently performance (Fig. 5.6a), the nice levels (see Fig. 5.6b) do not present significant discrepancies (i.e., no thread has significantly less CPU time than another) and they do not significantly oscillate.

Table 5.2 summarizes the average application performances deviations (one thread reporting performance) on the last 10 s of simulation for all three approaches and the selected target performances. These results reaffirm that the share controller accomplishes the desired behavior, but, once again, they suffer from averaging effects. A graphical interpretation is then critical to understand which one is the best option. Since approaches 1 and 2 result in high nice levels discrepancies (Fig. 5.4b and 5.5b) and approach 3 achieved an overall stable performance (Fig. 5.6), approach 3 was deemed the most appropriate to implement in the real system.

<table>
<thead>
<tr>
<th>Controller</th>
<th>Application deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application deviation</td>
<td>1</td>
</tr>
<tr>
<td>Target</td>
<td>1 HB/s</td>
</tr>
<tr>
<td>Approach 1</td>
<td>1.0944</td>
</tr>
<tr>
<td>Approach 2</td>
<td>1.1376</td>
</tr>
<tr>
<td>Approach 3</td>
<td>1.1050</td>
</tr>
</tbody>
</table>

5.2 Controller’s triggering

Considerations about the execution point where each controller should be called were purposely left out of Section 4.4 so that they could be made after looking at the data provided by the model present in Section 5.1.
Figure 5.4: a) Performance (real, R.P. and estimated, E.P.) of the 4 simulated applications (parent thread reporting performance) with approach 1, and b) Nice levels of the two simulated threads on core 1.

Figure 5.5: a) Performance (real, R.P. and estimated, E.P.) of the 4 simulated applications (parent thread reporting performance) with approach 2, and b) Nice levels of the two simulated threads on core 1.

Figure 5.6: a) Performance (real, R.P. and estimated, E.P.) of the 4 simulated applications (parent thread reporting performance) with approach 3, and b) Nice levels of the two simulated threads on core 1.
**Share controller** The *Share controller* should be the one called more often, but, as seen on Section 5.1, different invocation approaches must be taken into account. The *Share controller* should be triggered periodically, but it should only calculate new shares when all tasks in a core have reported their performance. Experimentally, the conclusions regarding the stability observed from the model were confirmed. As such, this controller was configured to only update the shares, in each core, when all the tasks in that core have updated their performance readings.

**Frequency controller** By observing the graphs from Section 5.1, it is clear that the performance of applications does not immediately converge to its target when there is a phase change. Since a frequency change will bump the performance in a similar way, a minimum number of calls to the share controller is required before the *Frequency controller* is invoked. This will prevent possible oscillating behaviors, and is accomplished through the `FREQ_WAIT` parameter presented in Algorithm 4.1 (see Section 4.4.2).

**Migration** The *Migration controller* is only expected to be needed once the actual number of threads is altered (which automatically triggers it) or the type of work done by a thread has a radical change of phase. The controller is thus periodically called, but with a period that allows the shares to re-stabilize after a migration, in order to guarantee that the system performance can stabilize before the thread allocation is reevaluated.

### 5.3 Benchmarks

It is important to recall that the task management framework proposed in this thesis mainly targets iterative applications that produce real time output (*e.g.*, for the user) and which do not necessarily have a finite execution time (*e.g.*, a game where different computations have to be continuously performed to deliver a smooth user experience). Although it is nearly impossible to find benchmarks matching exactly this criteria, there are several benchmark test suites that target different computing environments and platforms, which provide benchmarks that correctly characterize common multi-threaded iterative workloads.

Since the evaluated embedded system runs Linux, any compatible benchmark suite can be used, as long as its programs are portable to the ARMv7-A architecture. For this evaluation, several benchmark suites were considered, such as The San Diego Vision Benchmark Suite [40] (due to its vision oriented benchmarks), Princeton Application Repository for Shared-Memory Computers (PARSEC) [41] (with a range of multi-threaded benchmarks) and the Phoronix Test Suite [42] (bundles common Linux utilities with pre set inputs as benchmarks).

Some of the most important aspects for choosing suitable benchmark suites are: i) the availability of the source code of the suite; ii) the adaptability of the source code for the QoS metrics; and iii) the availability of multi-threaded applications. In practice, it is hard to simultaneously meet all three above-referred criteria for many existing benchmark suites, even when they provide publicly available source codes. For example, The San Diego Vision Benchmark Suite lacks multi-threaded tests, while a wide-
range of tests from the Phoronix Test Suite must be individually verified to meet the needed conditions, which is impractical for the purpose of this work.

Hence, the PARSEC benchmark suite was deemed appropriate to conduct the experimental exploration of the proposed framework, since it satisfies all three criteria. The PARSEC suite mainly focuses on multi-threaded workloads, their diversity (exploring several application domains) and non-high-performance computing applications. This benchmark suite includes applications ranging from image and video processing to content similarity search servers and partial differential equation handling programs. Table 5.3 briefly presents the 13 available tests in the PARSEC suite.

<table>
<thead>
<tr>
<th>Test name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>Option pricing with Black-Scholes Partial Differential Equation (PDE)</td>
</tr>
<tr>
<td>bodytrack</td>
<td>Body tracking of a person</td>
</tr>
<tr>
<td>canneal</td>
<td>Simulated cache-aware annealing to optimize routing cost of a chip design</td>
</tr>
<tr>
<td>dedup</td>
<td>Next-generation compression with data deduplication</td>
</tr>
<tr>
<td>facesim</td>
<td>Simulates the motions of a human face</td>
</tr>
<tr>
<td>ferret</td>
<td>Content similarity search server</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>Fluid dynamics for animation purposes with Smoothed Particle Hydrodynamics (SPH) method</td>
</tr>
<tr>
<td>freqmine</td>
<td>Frequent itemset mining</td>
</tr>
<tr>
<td>raytrace</td>
<td>Real-time raytracing</td>
</tr>
<tr>
<td>streamcluster</td>
<td>Online clustering of an input stream</td>
</tr>
<tr>
<td>swaptions</td>
<td>Pricing of a portfolio of swaptions</td>
</tr>
<tr>
<td>vips</td>
<td>Image processing (Project Website)</td>
</tr>
<tr>
<td>x264</td>
<td>H.264 video encoding (Project Website)</td>
</tr>
</tbody>
</table>

From the available PARSEC benchmarks, only Blackscholes, Dedup, Facesim, Ferret, Fluidanimate, Freqmine, Streamcluster, Swaptions and x264 were successfully compiled on the tested embedded system, with slight alterations to the source code in certain cases. Furthermore, only Blackscholes, Fluidanimate, Swaptions and x264 had relatively clear main iterations that can be used as application-specific QoS metrics, while other benchmarks require in-depth knowledge of the algorithm to annotate a performance milestone.

In order to make these benchmarks report the QoS metrics with the considered performance monitoring APIs (i.e., Heartbeats and beeps API), the following changes must be added to their source code:

- Algorithm 5.1 presents the introduced changes to initialize both the Heartbeats (to report the performance whenever the controller is not enabled) and Beeps API (used by the controller). Variables heart and sbeep_fd are global variables of the type heartbeat_t and int respectively. The other variables of interest are target (to express the target performance), window_size (to indicate the length of the sliding window used to report performance) and attr.type (to inform the Beep API and the controller about the type of the application, i.e., where all threads report their performance or where only one thread is responsible for reporting the performance).

- Algorithm 5.2 shows the introduced changes at the code segments which are considered to be an important milestone. For example, a signal can be sent via the APIs upon the completion of a
significant iteration. At this stage a call to both the Heatbeats and Beeps API should be placed. The variable identifier used by Heartbeats is a control variable to understand what event triggered the update in this API.

• Finally, when the application reaches its end, this event is reported to the APIs, so that appropriate clean up measures can be taken, as presented in Algorithm 5.3.

Algorithm 5.1 Initialization of the performance reporting API:

```
struct sbeep_attr attr;

//HB INIT
rc=heartbeat_init(&heart, target, target, window_size, 10, NULL);
if (rc != 0)
    printf("Error allocating heartbeat data \%d\n", rc);

// Beep INIT
attr.type = APP_TP_ALL_BEEP;
attr.inherit = 1;
attr.on_exec = 0;
attr.leader_pid = getpid();
attr.window_size = window_size;
attr.target_performance = target;
sbeep_fd = lsbeep_open();
if (sbeep_fd == -1) {
    printf(stderr, "test: error opening sbeep driver\n");
    return -1;
}
lsbeep_register(sbeep_fd, &attr);
```

Algorithm 5.2 Signalling function of the performance reporting API:

```
//HB signal
heartbeat(&heart, identifier);
```

Algorithm 5.3 Termination of the performance reporting API:

```
//HB signal
lsbeep_close(sbeep_fd);
```

In Algorithm 5.1 lines 2-7 correspond to Heartbeats initialization functions, while the remaining lines correspond to the Beep API initialization. In Algorithms 5.2 and 5.3 lines 1-3 correspond to Heartbeats signaling and closing, respectively, while the remaining accomplish the same functions on the Beep API.

One question that arises when one gets to the signaling functions (Algorithm 5.2) is where, in the program, to put this report and what it represents. Both Swaptions and Blackholes divide the workload into chunks and give each thread one of these chunks of data to process. This raises two problems: 1) the threads may report their performance simultaneously, which would create a momentary spike in performance; and 2), if the work is not equally divided among the threads, the last one may end up having less work to process than the others, thus leading to faster iterations. This is easily attenuated by
the fact that, in the case of the Beeps API, the performance is reported by thread, while for Heartbeats it requires the use of an average window corresponding to at least the number of threads.

As for Fluidanimate and x264, both applications rely on some sort of thread synchronization to report their performance. Fluidanimate reports the performance simultaneously in all threads, while x264 reports it in a single thread.

The application performance target, in a real system, should be chosen high enough as to maintain a good quality of service from the users point of view. However, the target performance should not be set too high as to ask for extra performance that would not contribute for the experience of the user (e.g., processing animations faster than they could be displayed or seen by the user).

5.4 Experimental setup

The proposed framework was experimentally tested on the Odroid-XU+E platform, introduced in Section 2.3 and presented in Fig. 5.7. This board is powered by a Samsung Exynos 5410 SoC, which features 4 Cortex-A7 (energy efficient) and 4 Cortex-A15 (high performance) cores together with a PowerVR SGX544MP3 GPU and 2GB LPDDR3 RAM. The supported big.LITTLE modes are limited to cluster migration (Section 2.2.1), which is driven by the system DVFS. From the OS point of view, the board features a discrete range of frequencies from 250 MHz to 1.6 GHz, which represents a virtual frequency range, as previously described in Section 4.4. In reality, when the system selects a virtual frequency in the range of [250 MHz, 600 MHz] the tasks will be migrated to the A7 cluster with a real operating frequency that is twice as high ([500 MHz, 1.2 GHz]). On the other hand, if the selected virtual frequency is in the range of [800 MHz, 1.6 GHz], tasks will run on the A15 cluster at the same physical frequency.

![Figure 5.7: The Odroid-XU+E development board used](image)

The board is running a custom Ubuntu Linux OS with a 3.4.84 kernel provided by the manufacturer of the board, Hardkernel. Although the board features a heat sink and fan (which starts working when temperatures raise), a common scenario is that, at maximum frequency, when temperatures reach values as high as 100º C, the board throttles down frequency from 1.6 GHz (maximum) to 1.4 GHz. This mechanism is highly relevant, since it might interfere with the functionality of both the system’s default
controller and the proposed one. The default system is prepared to expect this thermal emergencies and maintains the system at the lower operating frequency for enough time for the thermal emergency to pass. However, the proposed framework does not currently expect such a behavior and immediately requests an increase of system frequency. This implies that the system spends more time at high operating frequencies with the proposed framework that with the default tools, meaning that in these situations, the proposed framework may consume a greater amount of energy than the default control systems.

This board also features current and voltage sensors which provide a way to gather the power consumed by the board via a driver. These values are updated at every 0.3 s. To get a better insight on the capabilities of this board, it is important to understand how different cores behave while handling certain types of tasks (e.g., when accessing to different levels of memory). Unfortunately, although performance counters are supported by the architecture, they are not accessible on this board. Hence, to characterize the architecture, a set of micro-benchmarks is specifically developed for this purpose, as described in the following text.

### 5.4.1 Characterizing the Platform: Cortex-A7 vs Cortex-A15

To understand the benefits of heterogeneity, it is important to quantify the differences between cores that share the same [ISA] particularly the A7 and A15 cores used in the heterogeneous big.LITTLE architecture (see Section 2.1.2). While in this case both cores feature a 32 KB/32 KB Instruction/Data L1 Cache, they differ in the structure of the L2 cache where the A7 features 512 KB and the A15 2 MB L2 cache. When frequency is scaled, it affects all cores and caches simultaneously, but the main memory operates at a fixed and independent frequency, thus resulting in a difference in the number of access cycles as frequency scales.

#### Memory hierarchy benchmarking

Without performance counters it is hard to exactly evaluate the delays introduced by specific micro-instructions in a program’s throughput. Nevertheless, it is possible to approximately test how different instructions behave on the different types of cores based on specifically created benchmarks. Since both types of core implement the same [ISA] and extension, the focus of this analysis is on evaluating the access times to the different memory levels. The created micro benchmarks take 3 passes (discarding the first for memory harm-up), and in each pass the same amount of data (2097152 integers) is iteratively accessed 50 times. On each of this accesses the value stored in the corresponding address is incremented. The different levels of memory are accessed by addressing consecutive positions of memory at increasing offsets. As illustrated in Fig. 5.8 to access low memory levels (e.g., L1 cache) a small number of consecutive memory addresses are accessed. To increase the memory level, the number of consecutive addresses used is increased while proportionally decreasing the number of repetitions over the same address. Thus the total number of operations is kept constant while accessing different memory levels.

This benchmark was coded in C, but to get a better insight into the generated Assembly instructions the Assembly output was generated. As can be seen in Algorithm 5.4 which represents the main
iteration loop in Assembly, each iteration requires 4 memory instructions, 3 loads (lines 4, 6 and 9) and 1 store (line 8) with the remaining instructions being related to the increment of the stored value and address calculation. The PAPI library [43] was used to gather the number of cycles taken to execute the relevant portion of code. The complete set of results from this benchmark evaluation is present in Appendix [5] while Table [5.4] summarizes those results by showing the obtained ranges of number of cycles at each memory level. It was verified that, in each type of core, the number of cycles to complete the micro-benchmark is consistent inside each cache level independently of the system frequency. However, when accessing the main memory, even inside the same type of core, the number of cycles to complete the memory benchmark varies. This is mainly due to the fact that, at the same frequency and for data chunks with the size close to the size of the previous memory level, some portions of data may still be cached at a lower cache level due to the pseudo-random 8-way (A7) and 16-way (A15) set-associative L2 cache. Hence, the amount of cycles required to process the same number of operations is expected to increase with the data size at the same frequency. Furthermore, the number of cycles to complete the benchmark at the level of the main memory also increases with increasing system frequencies. This can be explained by the fact that the memory frequency is fixed while the CPU one varies, requiring more cycles to pass while waiting for data.

**Algorithm 5.4** Assembly instructions per integer processed in the microbenchmark

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>ldr r3, [r7, #8]</code></td>
</tr>
<tr>
<td>2</td>
<td><code>adds r3, r3, #42</code></td>
</tr>
<tr>
<td>3</td>
<td><code>lslis r3, r3, #2</code></td>
</tr>
<tr>
<td>4</td>
<td><code>ldr r2, [r7, #32]</code></td>
</tr>
<tr>
<td>5</td>
<td><code>add r3, r3, r2</code></td>
</tr>
<tr>
<td>6</td>
<td><code>ldr r2, [r3]</code></td>
</tr>
<tr>
<td>7</td>
<td><code>adds r2, r2, #1</code></td>
</tr>
<tr>
<td>8</td>
<td><code>str r2, [r3]</code></td>
</tr>
<tr>
<td>9</td>
<td><code>ldr r3, [r7, #8]</code></td>
</tr>
<tr>
<td>10</td>
<td><code>#adds r3, r3, #43</code></td>
</tr>
<tr>
<td>11</td>
<td><code>lslis r3, r3, #43</code></td>
</tr>
</tbody>
</table>

**Table 5.4:** Summary of the number of cycles to complete memory micro-benchmark

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Number of cycles ($\times 10^6$) by memory level</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L1</td>
</tr>
<tr>
<td>A7</td>
<td>~29.6</td>
</tr>
<tr>
<td>A15</td>
<td>~21.1</td>
</tr>
<tr>
<td>Speed up</td>
<td>1.40</td>
</tr>
</tbody>
</table>

Table 5.4 also indicates the obtained speed up on this benchmark between the different core types at each memory level. This micro-benchmark shows that, even in memory bounded situations, the
A15 can achieve speed ups of up to 1.4 the speed of the A7. Since the instructions in the tested benchmark are not solely memory related (see Algorithm 5.4) it is impossible to derive the memory-specific speedup times without performance counters. However, the obtained speedups are lower than expected according to the virtual-to-real frequency scaling on this board (which considers that, at the same frequency, the A15 is twice as fast than the A7).

Energy consumption and performance during benchmark execution

To test the energy consumption and performance on both A7 and A15 clusters, a set of 4 benchmarks from the PARSEC benchmark suite was run, at the same frequency, on both clusters. The selection of benchmarks is similar to the one previously presented in Section 5.3, i.e., Fluidanimate, Swaptions, x264 and Blackscholes were used. The configurations and inputs used for these benchmarks can be seen in Table 5.5. All the benchmarks were run with 4 threads in both cores, with a fixed frequency of 1.2 GHz.

Table 5.5: Program inputs and signal location used to test the benchmark performance on the A7 and A15 cores

<table>
<thead>
<tr>
<th>Test</th>
<th>Input set</th>
<th>HB location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fluidanimate</td>
<td>PARSEC’s in_300k.fluid Frames: 150</td>
<td>Every frame</td>
</tr>
<tr>
<td>Swaptions</td>
<td>Swaptions: 128 Simulations: 400000</td>
<td>Every swaption</td>
</tr>
<tr>
<td>x264</td>
<td>sintel_trailer_2k_480p24.y4m</td>
<td>Every frame</td>
</tr>
<tr>
<td>Blackscholes</td>
<td>PARSEC’s in_10M.txt</td>
<td>Every 25000 options</td>
</tr>
</tbody>
</table>

Table 5.6 shows the average measured performance of the benchmarks on the two clusters in Beeps/s (B/s) using the Beeps subsystem (API+driver) previously described in Section 4.3. As expected, the A15 cores were capable of outperforming the A7 cores for Swaptions, x264 and Fluidanimate benchmarks where speed ups of up to 2.54 were achieved on x264 when running the program on the A15. It is interesting to notice that for Blackscholes a slight performance decrease can be observed while compared to the A7 cluster. Hence, the speed ups obtained when migrating applications across cores can present significant variations, depending on the application characteristics and how they interact with the different aspects of the micro-architecture.

Table 5.6: Uncontrolled benchmark performance on the A7 and A15 clusters

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Benchmark performance [B/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Swaptions</td>
</tr>
<tr>
<td>A7</td>
<td>0.51</td>
</tr>
<tr>
<td>A15</td>
<td>0.74</td>
</tr>
<tr>
<td>A15/A7 ratio</td>
<td>1.48</td>
</tr>
</tbody>
</table>

For the same set of tested PARSEC benchmarks, the experimentally obtained energy consumption is presented in Table 5.7. As expected, the A7 was always able to achieve energy savings when compared to the A15. These tests also reveal the average power ratio of the A7 and A15 cores while handling these benchmarks (Table 5.8). By looking at the presented values of performance (Table 5.6) and power (Table 5.8), it is clear that, for these workloads, the A15 requires 4x more power than the A7 to process the same data while its performance level is, at most, 2.54x better on average.
Table 5.7: Uncontrolled benchmark energy consumption (both clusters and memory) on the A7 and A15 clusters

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Energy [J]</th>
<th>Swaptions</th>
<th>x264</th>
<th>Fluidanimate</th>
<th>Blackscholes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A7</td>
<td>184.79</td>
<td>84.57</td>
<td>87.61</td>
<td>21.78</td>
<td></td>
</tr>
<tr>
<td>A15</td>
<td>548.02</td>
<td>144.93</td>
<td>240.50</td>
<td>92.74</td>
<td></td>
</tr>
<tr>
<td>A15/A7 ratio</td>
<td>2.97</td>
<td>1.71</td>
<td>2.75</td>
<td>4.26</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.8: Uncontrolled benchmark average power ratio (A7/A15)

<table>
<thead>
<tr>
<th>Average power ratio (A15/A7)</th>
<th>Swaptions</th>
<th>x264</th>
<th>Fluidanimate</th>
<th>Blackscholes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4.34</td>
<td>4.34</td>
<td>4.32</td>
<td>4.17</td>
</tr>
</tbody>
</table>

**Frequency and migration time**

As previously referred, in the tested ARM platform, changing the frequency level beyond a certain virtual threshold value (600 MHz), also triggers the task migrations across different core types. In order to experimentally assess the time required to apply the new frequency (DVFS) and to migrate the tasks, the testing procedure adopted herein performs the system call to change the frequency and measures the amount of time required to apply changes. The test was repeated 50 times for every possible combination of “from” and “to” frequencies and a median of the obtained values was taken. The obtained experimental results are summarized in Table 5.9.

The gathered information is organized in four categories: two that consider migration from A7 (virtual frequency ≤ 600 MHz) to A15 (virtual frequency ≥ 800 MHz) and vice versa; and two others that consider frequency scaling inside the same cluster (A7 to A7 and A15 to A15). As it can be observed, frequency changes in the same core (without triggering migration) are around 4x faster than those that trigger a cluster migration.

Table 5.9: Frequency scaling times (DVFS) on the ODROID-XU+E

<table>
<thead>
<tr>
<th>Maximum frequency scaling time [µs]</th>
<th>Same cluster</th>
<th>Different cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>A7 → A7</td>
<td>525</td>
<td>510</td>
</tr>
<tr>
<td>A15 → A15</td>
<td>1796</td>
<td>2365</td>
</tr>
</tbody>
</table>

5.5 **Experimental results**

To experimentally validate the proposed framework and controller, different combinations of the previously introduced PARSEC benchmark programs were run with varying number of threads and performance targets. Two different testing configurations of the proposed framework were devised. The first one considers frequency control and share control at the overall application level, while the second configuration implements the framework with all its components and controllers. These two framework configurations were devised in order to thoroughly evaluate the benefits of implementing the developed framework and its components in a real system from different aspects.
5.5.1 Evaluation of application-wide Share and Frequency controller

The first test configuration of the proposed framework includes the Frequency and Share controller applied at the overall parallel application level. It does not include the Migration Controller, thus it is better suited for parallel applications with a similar number of threads allocated to the same number of cores. Moreover, it relies only on the existing Heartbeats API [16], thus no thread specific information is available, only process wide shares and performance. It features the same logic and solves the same problems presented in Sections 4.4.1 and 4.4.2 but only considers the overall application performance, shares and frequency. The general layout of this framework’s configuration is presented in Fig. 5.9 Using the provided target performance, the shares that present the best compromise between applications’ performances, together with a new operating frequency, are calculated. This is done at every controller invocation, which happens at regular time intervals, i.e., 1 s in order not to throttle the system. The programs that were used to test this configuration of the framework were configured to use 4 threads each and their performance reporting intervals and inputs are presented in Table 5.10.

![Figure 5.9: Block diagram of the first controller implementation](image)

Table 5.10: Program inputs, signal location and maximum performance used to test the algorithm using only the frequency and share controllers

<table>
<thead>
<tr>
<th>Test</th>
<th>Input set</th>
<th>HB location</th>
<th>Max. Overall Av. Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fluidanimate (F)</td>
<td>PARSEC's in_300k.fluid Frames: 150</td>
<td>Every frame</td>
<td>2.5 HB/s</td>
</tr>
<tr>
<td>Swaptions (S)</td>
<td>Swaptions: 128 Simulations: 400000</td>
<td>Every swaption</td>
<td>1.0 HB/s</td>
</tr>
<tr>
<td>x264 (x)</td>
<td>sintel_trailer_2k_480p24.y4m</td>
<td>Every 5 frames</td>
<td>5.4 HB/s</td>
</tr>
<tr>
<td>Blackscholes (B)</td>
<td>PARSEC's in_10M.txt</td>
<td>Every 500000 options</td>
<td>7.7 HB/s</td>
</tr>
</tbody>
</table>

The experimental results obtained with this framework configuration are summarized in Table 5.11. As it can be concluded by analyzing the presented values, this implementation of the proposed task management framework is capable of providing substantial benefits in terms of the relative performance difference between the achieved and the targeted performance levels (compare columns 4 and 7 marked with “error” in Table 5.11). The advantage of the proposed framework was even more evident when all four benchmarks were simultaneously executed in the system. In this case, it was possible to observe a significant decrease (16 ×) in the performance relative error. In particular, it was observed that, with the controller, there was an almost exact match between the target (column 2) and observed performance levels (column 6) for the Fluidanimate, Blackscholes, and Swaptions benchmarks. There was however a
slight error for the x264 application, which was caused mostly because its performance rapidly changes during the execution (e.g., by switching between intra and inter frames) and it highly depends on the content of the input video sequence.

Table 5.11: Result summary for the controller combining application share and frequency control, with performance and energy data for the sets of tests used

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Target Performance</th>
<th>Without an active controller</th>
<th>Energy</th>
<th>With the proposed controller</th>
<th>Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fluidanimate (FA)</td>
<td>1.2</td>
<td>1.034</td>
<td>0.044</td>
<td>1116</td>
<td>1.169</td>
</tr>
<tr>
<td>Swaptions (S)</td>
<td>0.4</td>
<td>0.477</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x264 (x264)</td>
<td>3.0</td>
<td>3.397</td>
<td>0.029</td>
<td>2022</td>
<td>2.949</td>
</tr>
<tr>
<td>Fluidanimate (FA)</td>
<td>1.1</td>
<td>1.981</td>
<td>0.701</td>
<td>634</td>
<td>1.798</td>
</tr>
<tr>
<td>Swaptions (S)</td>
<td>2.1</td>
<td>1.599</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x264 (x264)</td>
<td>0.8</td>
<td>1.006</td>
<td>0.961</td>
<td>1318</td>
<td>0.790</td>
</tr>
<tr>
<td>Fluidanimate (FA)</td>
<td>0.3</td>
<td>0.447</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swaptions (S)</td>
<td>1.4</td>
<td>0.268</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x264 (x264)</td>
<td>0.6</td>
<td>0.796</td>
<td>2.801</td>
<td>2701</td>
<td>0.600</td>
</tr>
<tr>
<td>Fluidanimate (FA)</td>
<td>1.0</td>
<td>2.223</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swaptions (S)</td>
<td>0.2</td>
<td>0.372</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x264 (x264)</td>
<td>1.0</td>
<td>0.319</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In addition, the framework allowed the system to achieve significant energy savings, whenever a feasible set of target performances was requested. In fact, as presented in Table 5.11, savings of up to 49% were observed for different sets of the considered benchmarks. However, due to the thermal issues discussed in Section 5.4, the system may automatically lower down its frequency. Since the reduction in frequency is more notorious on the default scheduler, setting the performance target limits to very high values might result in an increased energy consumption. For example, in the case of B+S benchmark set in Table 5.11 with the controller enabled the system was able to sustain maximum frequency for a longer period, achieving the expected performance level at the cost of increasing its energy consumption by 16%.

Figure 5.10 shows the experimentally obtained performance levels and system frequency of Fluidanimate, Swaptions and x264 respectively (both with and without the controller). For the uncontrolled data set (left side of Fig. 5.10), Fluidanimate (FA) presents the earliest finish followed by Swaptions (S) and x264 (x264). Although the obtained per-application performance is irregular, it is clearly evident that the target performance (represented with the dotted line) is not achieved for Fluidanimate and Swaptions (real performance is higher than target) and x264 (lower than target). Also, it can be noticed that, as soon as one application completes, the remaining applications experience a performance boost which may result in delivering a performance that is even further away from the target (as is the case of Swaptions).

On the other hand, the right side of Fig. 5.10 presents the achieved performance levels for the same set of benchmarks when the runtime controller is enabled. As it can be observed, their realistically achieved average performance was much closer to target (as also reported in Table 5.11). Figure 5.11 shows the power consumption and virtual runtime frequency when executing FA+S+ x264 benchmark set. As it can be observed, in contrast to the execution without the controller (top of Fig. 5.11), reaching the target performance levels with the proposed method (bottom of Fig. 5.11) does not always require execution at the maximum frequency. This allows achieving significant energy savings, since during certain application phases the execution was even migrated from the A15 to the A7 cluster (see $f \leq 600$MHz). In
this particular case, the execution with the proposed controller was capable of attaining an energy consumption reduction of about 11.9% (on average) when directly compared with the execution that relies on the default scheduler and DVFS decisions (see Table 5.11).

Figure 5.10: Application performance during the simultaneous execution of Fluidanimate, Swaptions and x264 with the application share and frequency controllers, the dashed line represents the target while grey bars present a ±10% deviation area from target

Figure 5.11: Frequency and power consumption during the execution of Fluidanimate, Swaptions and x264 with the application share and frequency controllers

This implementation of the proposed framework, is thus capable of optimizing the run of each parallel benchmark, such that the target performance is achieved for all tests simultaneously. The presented relative performance values in Table 5.11 also confirm this behavior, where a significant reduction can be observed for different sets of three and four simultaneously run benchmarks, i.e., the reduction ranges from 0.961 to 0.101 and from 2.801 to 0.168, respectively. It is worth noting that although this framework configuration was capable of bringing the performance close to the required target, certain oscillatory behaviors were observed. This behavior was most likely caused by conflicts between the Frequency and Shares controllers. Another source of this oscillatory behavior can be found in the performance compensation mechanism that was only introduced in this configuration of the framework. Performance compensation was applied whenever a task’s performance raised above or bellow its target by increasing
or decreasing the target performance in order to reduce the cumulative performance error. However, in certain scenarios, this may lead to oscillatory behaviors due to the clashing between all the controlling elements. Hence, this feedback was deprecated in the full configuration of the framework and left as a possible consideration for future work.

5.5.2 Evaluation of the proposed full framework for performance and energy-aware real-time scheduling for heterogeneous embedded systems

The second test configuration uses the full potential of the proposed framework. It relies on the Beep API and driver, as well as all three types of controllers, i.e., Frequency, Share (at the level of the threads in a core) and Migration controller. This implementation does not impose the previous restriction (present in the implementation detailed in Section 5.5.1) about the number of application threads (and core allocations). The absence of these restrictions is in part due to the inclusion of the Migration controller, that can distribute the system load among the cores, and the use of the Beep subsystem which can accurately report task data at the level of a thread. As a result, both single- and multi-threaded applications can be handled simultaneously.

In order to prevent performances instabilities when several controllers are simultaneously invoked, their invocation time intervals must be carefully chosen. It was empirically verified that a minimum Share controller invocation period of 0.2 s allows achieving execution control of satisfactory granularity. This interval holds as long as all the threads in a core report their performance in that interval and is automatically extended otherwise (see the considerations in Section 5.2). The minimum invocation period of the Frequency controller is set to 2 s, i.e., after 10 share updates (invocations of the share controller). This invocation period is selected to prevent performance instabilities, since there must be a minimum number of share updates performed before the frequency can be scaled. Furthermore, as presented in Table 5.9, changing the frequency may take up 2.3 ms, thus invoking the frequency controller more often is not beneficial. This is because that, in addition to the Frequency controller's execution time, the other controllers may be simultaneously invoked, thus increasing the computational overhead. As for the Migration controller, it was activated whenever the total number of threads in the system changes and at every 3 s, in order for it to recognize possible phase changes in the tasks and migrate them accordingly periodically.

Several combinations of the previously identified benchmarks were used to test the proposed framework, as presented in Table 5.12. While Swaptions, Fluidanimate and Blackscholes are considered to have a stable behavior throughout their execution, x264 has a behavior heavily dictated by its input, i.e., in a general case its performance is highly unpredictable. Table 5.12 also lists the configurations of the benchmarks for the experimental evaluation, where Blackscholes is configured to report performance much more frequently than in the previous configuration (Table 5.10). The higher reporting rate for Blacksholes is used to diversify the test sets.

The experimentally obtained results with different combinations of the introduced benchmarks are presented in Table 5.13. As was the case with the first implementation (Section 5.5.1), the controller managed to reduce the Performance relative error in all tested scenarios (compare columns 4 and 7 of
Table 5.12: Program inputs, signal location and maximum performance used to test the full implementation of the framework

<table>
<thead>
<tr>
<th>Test</th>
<th>Input set</th>
<th>Beep location</th>
<th>Max. Overall Av. Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fluidanimate (F)</td>
<td>PARSEC’s in,300k.fluid Frames: 150</td>
<td>Every frame</td>
<td>2.5 B/s</td>
</tr>
<tr>
<td>Swaptions (S)</td>
<td>Swaptions: 128 Simulations: 400000</td>
<td>Every swaption</td>
<td>1.0 B/s</td>
</tr>
<tr>
<td>x264 (x)</td>
<td>sintel_trailer_2k_480p24.y4m</td>
<td>Every 2 frames</td>
<td>13.1 B/s</td>
</tr>
<tr>
<td>Blackscholes (B)</td>
<td>PARSEC’s in,10M.txt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.13. The reduction of the performance relative error ranges from values of around 1,09x (row 6 on Table 5.13) up to 122x (row 3 on Table 5.13). Furthermore, all the test sets that did not require the board to operate at maximum frequency (all the tests that do not have * nor ** in their target performance, column 2) achieved energy savings. Specifically, the test set running two instances of Swaptions (row 3) with 4 threads per instance achieved 65.4% energy savings. On the other hand, the test sets presented in rows 4, 8 and 10 of Table 5.13 required the full availability of the resources in the board, either because their performance targets were unfeasible (rows 4 and 8) or because they were at the limit performance that the system can provide (row 10). In these situations, the previously discussed thermal issues (see Section 5.4) that also affected the first implementation’s results (Section 5.5.1) contributed to the increase of energy consumption with the proposed framework, with the only exception being the case with Blackscholes and Fluidanimate in row 8.

Table 5.13: Result summary for the full implementation of the framework, with performance and energy data for different sets of tests, the performance values refer to the portion where the controller was active while energy represents the overall value until all programs complete their execution.

<table>
<thead>
<tr>
<th>Benchmark settings</th>
<th>Without an active controller</th>
<th>With the proposed controller</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Observed Performance $P_i$ [B/s]</td>
<td>Energy Consumption $E_i$ [J]</td>
</tr>
<tr>
<td>Swaptions (S) 4T</td>
<td>0.3</td>
<td>11.800</td>
</tr>
<tr>
<td>Swaptions (S) 4T</td>
<td>0.1</td>
<td>1.1</td>
</tr>
<tr>
<td>Swaptions (S) 2T</td>
<td>0.4</td>
<td>0.126</td>
</tr>
<tr>
<td>Swaptions (S) 1T</td>
<td>0.6</td>
<td>0.25</td>
</tr>
<tr>
<td>Swaptions (S) 4T</td>
<td>0.2</td>
<td>2.261</td>
</tr>
<tr>
<td>Swaptions (S) 4T</td>
<td>0.1</td>
<td>0.204</td>
</tr>
<tr>
<td>Fluidanimate (FA) 4T</td>
<td>1.0</td>
<td>0.203</td>
</tr>
<tr>
<td>Swaptions (S) 2T</td>
<td>0.16</td>
<td>38.404</td>
</tr>
<tr>
<td>Swaptions (S) 1T</td>
<td>0.2</td>
<td>0.863</td>
</tr>
<tr>
<td>Swaptions (S) 1T</td>
<td>0.25</td>
<td>0.791</td>
</tr>
<tr>
<td>Swaptions (S) 2T</td>
<td>0.25</td>
<td>0.374</td>
</tr>
<tr>
<td>Swaptions (S) 4T</td>
<td>0.25</td>
<td>0.791</td>
</tr>
<tr>
<td>Fluidanimate (FA) 4T</td>
<td>1.0</td>
<td>1.312</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>10.0</td>
<td>30.277</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>0.16</td>
<td>38.404</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>0.2</td>
<td>0.863</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>0.25</td>
<td>0.791</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>0.25</td>
<td>0.374</td>
</tr>
<tr>
<td>Fluidanimate (FA) 4T</td>
<td>1.0</td>
<td>1.312</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>10.0</td>
<td>30.277</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>0.16</td>
<td>38.404</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>0.2</td>
<td>0.863</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>0.25</td>
<td>0.791</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>0.25</td>
<td>0.374</td>
</tr>
<tr>
<td>Fluidanimate (FA) 4T</td>
<td>1.0</td>
<td>1.312</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>10.0</td>
<td>30.277</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>0.16</td>
<td>38.404</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>0.2</td>
<td>0.863</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>0.25</td>
<td>0.791</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>0.25</td>
<td>0.374</td>
</tr>
<tr>
<td>Fluidanimate (FA) 4T</td>
<td>1.0</td>
<td>1.312</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>10.0</td>
<td>30.277</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>0.16</td>
<td>38.404</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>0.2</td>
<td>0.863</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>0.25</td>
<td>0.791</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>0.25</td>
<td>0.374</td>
</tr>
<tr>
<td>Fluidanimate (FA) 4T</td>
<td>1.0</td>
<td>1.312</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>10.0</td>
<td>30.277</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>0.16</td>
<td>38.404</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>0.2</td>
<td>0.863</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>0.25</td>
<td>0.791</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>0.25</td>
<td>0.374</td>
</tr>
</tbody>
</table>

* These performance combinations are unfeasible, the benchmarks cannot achieve the target performance and the system will try to increase its frequency, meaning that due to the considerations presented in Section 5.4 more energy may be spent with the controller.

** - Although these performances are feasible, 0.25 B/s is the maximum performance achievable by Swaptions with one thread, meaning that the system will be forced to maintain maximum frequency.

In order to provide a better understanding of the behavior of the proposed controller, Fig. 5.12 presents the run-time behavior of two instances of Swaptions, both with 4 threads and feasible per-
formance targets, specifically, the first configuration (row 3) in Table 5.13. Grey bars on the performance plots (Fig. 5.12a) represent a ±10% target (marked by a dashed line) deviation. The represented performance levels are clipped as to only show the relevant (parallel) portion of the applications’ execution. Prior to the activation of the controller (left side of Fig. 5.12a) both instances of Swaptions, which offers extremely stable workloads, were above the performance target with their performance equally divided, i.e., they both achieved a value close to 0.5 B/s while the maximum performance achievable by Swaptions is 1 B/s. However, when the framework was enabled (right side of Fig. 5.12a) their performances quickly matched the performance target. The stability of this benchmark allowed it to reduce its relative performance error from 11.800 to 0.096, a reduction of 122x (see Table 5.13).

![Performance Comparison Graphs](image)

**Figure 5.12:** Runtime a) Performance and b) Frequency and Power consumption of two instances of Swaptions (4 threads) with and without the usage of the control framework. Grey bars on a) represent ±10% target (dashed line) deviation.

Figure 5.12b illustrates both the system frequency and power consumption during the execution of the benchmarks. Whenever the system frequency enters the grey zone of the graph the A7 cluster is in use, while on the white zone tasks are running on the A15 cluster. With the system’s default management tools (top of Fig. 5.12b), this test configuration runs with a frequency of 1.6 GHz on the A15 which quickly drops to 1.4 GHz due to thermal emergency. However, with the proposed controller (bottom of Fig. 5.12b), most of the execution period is spend on the A7 cluster. Even tough the execution time approximately doubled due to the selected low performance target (see Time axis of Fig. 5.12b), the accumulated power consumption (i.e., energy) was 65.4% lower (see Table 5.13). On the last execution stages with the framework (after 400 s in the bottom of Fig. 5.12b) the system frequency was increased due to the fact that, at this point, only one application was still running on the system, which disables the framework.

In order to further experimentally assess the efficiency of the proposed framework, even more challenging execution scenarios were considered, where different combinations of benchmarks and thread numbers are used, including the unpredictable x264 encoder. Figure 5.13 illustrates a case where 3 different benchmarks (i.e., Blackscholes, Swaptions and Fluidanimate) were called with 2 threads each (corresponding to row 9 of Table 5.13). As in the case of Fig. 5.12 the grey bars in the perfor-
mance graphs (Fig. 5.13a) represent a ±10% deviation from target (dashed line) while the grey area in Fig. 5.13b corresponds to A7-mapped frequencies.

The test configuration of Fig. 5.13 presented an interesting case study since the framework was controlling 6 threads, meaning that 2 of the cores were dedicated to the execution of only one thread. However, the system smoothly handled this challenge (Fig. 5.13a) and all applications managed to execute at or above their target performance. Since different types of workloads were present in the system, it was not possible to exactly match the specified targets (right side of Fig. 5.13a) meaning that both Blackscholes and Fluidanimate had to increase their performance in order for Swaptions to achieve its target. Nevertheless, this presented an improvement in performance (see left side of Fig. 5.13a, with all applications over-performing) of 32x (see Table 5.13). Blackscholes often presented spikes in its execution which were attributed to the simultaneous reporting of performance across threads. Although graphically this may appear to show irregular performance, it was verified that most of the reports are on the lower contour of the line. Furthermore, while Heartbeats (used on the left side of Fig. 5.13a, without the controller) records every performance communication, Beeps only samples (and stores) the performance at every 0.2 s (minimum Share controller invocation time), which, depending on the application, may happen less often than the performance report.

With this configuration, while using the system’s default controllers, the system frequency often oscillated between 1.6 and 1.4 GHz (see top of Fig. 5.13b) which is a sign of thermal emergency. However, the controller once again achieved significant energy savings, 38.5%, by performing part of the applications’ execution on the A7 cluster (see bottom of Fig. 5.13b).

Figure 5.14 illustrates a new case where Blackscholes, 2 instances of Swaptions and Fluidanimate were simultaneously ran with 2, 4, 1 and 4 threads respectively (row 10 of Table 5.13). The main difference between this scenario and the previously presented ones lies in the fact that one of the test set members (Swaptions, with one thread) requires the utilization of a system core at maximum frequency. This is due to the fact that the selected performance target for this benchmark corresponds to the maximum achievable performance by one thread of Swaptions in this system. While the overall performance error was decreased by 2.5x and all applications were able to match their targets (compare the uncontrolled, left, and controlled, right, side of Fig. 5.14a), the energy consumption increased by 10.6%, as presented in Table 5.13. Since the selected performance requires the maximum system frequency to be selected, the system was susceptible to enter a thermal emergency scenario even while using the proposed framework. In fact, the controller fought this scenario as can be seen from the fast frequency transitions on the bottom graph of Fig. 5.14b. Due to this, the system spent more time at maximum frequency with than without the proposed framework, and consequently spend more energy.

A final test configuration worth analyzing is a scenario where all benchmarks (Blackscholes, Swaptions, Fluidanimate and x264) were simultaneously run with 4 threads each (last row of Table 5.13). The particularity of this test configuration is that it used the x264 encoder, a benchmark that is irregular in nature due to its input dependencies. Figure 5.15 depicts the run time behavior of this configuration. One thing that is immediately noticeable is that the achieved performance with the proposed control framework oscillates more than in previous cases (see the right side of Fig. 5.15a). This is a consequence
of the unpredictability of x264 whose performance is often over estimated (notice the consecutive drops in the bottom right graph of Fig. 5.15). Nevertheless, even in this scenario, the framework was able to improve overall performance (compared to the default tools) by 4x while achieving 55.1% energy savings (see last row of Table 5.13). These improvements were possible due to a fairer performance distribution among tasks and the reduction of the system frequency (and power consumption) during the execution of the benchmark set (see bottom graph in Fig. 5.15b).

Overall, and according to the results presented in Table 5.13, the proposed framework was able to significantly reduce the performance relative error, up to 122x, and energy consumption by up to 65% (in the first presented case) when compared to the default system control methodologies. It should be noted that the achieved energy savings are tightly coupled with the specified performance targets, but they would not be possible without the introduction of the proposed framework in the system. It is also worth mentioning that, as presented in Fig. 5.16, the framework overhead to the system while controlling 18 threads from 4 applications was measured to be of only 0.3%. Since the required computations depend on the number of registered threads, this load is expected to increase either if the system frequency drops or the number of processed threads increases. The 18 threads were spawn by the 4 available benchmarks, all called with 4 threads. x264 creates two additional threads (a total of 6) and hence the total of 18 captured by the controller in this case.

5.6 Summary

This chapter presented a thorough analysis of the proposed performance and energy-aware real-time scheduling for heterogeneous embedded systems framework. It started with the analysis of the proposed framework on an artificial modeled environment, followed by an analysis of the required benchmarks and experimental platform. After that, two controller implementations were presented: i) one with only the Share and Frequency controller, limited to controlling only multi-threaded applications with their threads bounded to the same number of cores; and another ii) that implements the full proposed framework, capable of handling both single-threaded and multi-threaded applications simultaneously.

The modeled system yielded valuable insights, as to the stability of the controller and the most appropriate moment to trigger a share update; it turned out to be to update the shares of the threads in a core whenever all of them had reported new performance values. Afterwards, the reasons behind the selection of the PARSEC benchmark suite were discussed together with the list of used benchmarks from that suite (i.e., Blackscholes, Fluidanimate, Swaptions and x264). The required procedure to modify these benchmarks in order for them to report performance in real time was also explained.

The experimental platform, an Odroid-XU+E board, was then introduced together with the results from some custom built micro-benchmarks. With these micro-benchmarks, it is possible to identify the differences between the A7 and A15 CPU clusters of the board. These tests focused on assessing and characterizing the memory hierarchy, frequency change (DVFS) and migration times, and benchmark performance and energy consumption on different clusters of cores.

Finally, the obtained experimental results for the two different implementations, with only the share
and frequency controller and the full framework, were presented. It was demonstrated that the full implementation of the framework successfully managed to reduce the performance relative error of parallel applications. According to the selected performance targets, the framework correctly handled the balance of applications with different number of threads and different behaviors while achieving up to 122x reductions on the performance relative error and 65% energy savings once compared with the system default behavior. Furthermore the controller load to the system was measured to amount to 0.3%, a value that is not expected to become a significant overhead.
Figure 5.13: Runtime a) Performance and b) Frequency and Power consumption during the execution of Blacksc- holes, Swaptions and Fluidanimate (2 threads each) with and without the usage of the control framework. Grey bars on a) represent ±10% target (dashed line) deviation.
Figure 5.14: Runtime a) Performance and b) Frequency and Power consumption during the execution of Blacksc- holes (2 threads), Swaptions (one instance with 4 and another one with 1 thread) and Fluidanimate (4 threads) with and without the usage of the control framework. Grey bars on a) represent ±10% target (dashed line) deviation.
Figure 5.15: Runtime a) Performance and b) Frequency and Power consumption during the execution of Blacksheoles, Swaptions, Fluidanimate and x264 (4 threads each) with and without the usage of the control framework. Grey bars on a) represent $\pm 10\%$ target (dashed line) deviation.

Figure 5.16: Load of the proposed framework while handling 4 apps with 18 threads: Swaptions, Fluidanimate, Blacksheoles and x264, all called with 4 threads, x264 spawns two additional threads.
Conclusions
This thesis presents a state of the art performance and energy-aware real-time scheduling for heterogeneous embedded systems framework. It proposed a new method for dynamic scheduling on real-time heterogeneous embedded systems and developed a framework capable of maintaining performance fairness by core and scaling the available system resources in order to follow performance targets of multiple single-threaded and multi-threaded applications without spending unneeded energy. The necessity for such a controller is evidenced by the proliferation of the use of smartphones and tablets and the need to achieve higher performances on these devices with limited energy budgets. One consequence of this growth is that heterogeneous architectures are evermore common in the embedded architectures market, bringing complex scheduling issues to these platforms.

Most of the current scheduling solutions focus on scheduling tasks on desktops or high performance computing systems, meaning that a significant portion of the restrictions found on embedded systems are not often considered. The existing studies targeted at heterogeneous embedded platforms lack the functionality to capture the performance of individual application threads, and are thus not capable of following QoS references for multi-threaded applications. The proposed task management framework, is the result of a thorough state of the art study on scheduling techniques and heterogeneous embedded architectures, specifically big.LITTLE.

The developed scheduling framework relies on online performance monitoring to explicitly capture the run-time behavior of multiple parallel applications running on the underlying architecture. Based on the monitored task-specific parameters, the proposed method provides the decisions regarding the allocation of shared system resources, such that the target performance levels are achieved, as well as to guarantee the performance fairness among the running tasks through a Share controller. The proposed method also relies on DVFS to manage the system energy-efficiency levels and to further augment the scope in which performance can be scaled such that energy consumption savings are achieved. Since the target platform relies on the virtual frequency set by the DVFS driver to control its operation point, the proposed method further manages the system heterogeneity without the need to perform any kernel level modifications. Finally, the framework implements a migration mechanism, which is used to balance threads across cores and thus provide the simultaneous control over single-threaded and multi-threaded applications.

The conducted evaluation revealed that the proposed performance and energy-aware real-time scheduling for heterogeneous embedded systems framework is able to accomplish the defined objectives by providing a fairer execution of tasks across the system while saving energy. These tests relied on the Beeps subsystem, a purpose built performance reporting tool, inserted into four programs from the PARSEC benchmark suite, Blackscholes, Fluidanimate, Swaptions and x264. Running these programs in different combinations (and different numbers of threads) yielded a range of scenarios that successfully verified the framework behavior.

Depending on the selected targets, results showed energy savings of up to 65% as well as significant reductions on the performance relative error, up to 122x. These results are biased by the selected performance targets, but this is precisely where the strength of this approach lies. The framework is solely responsible to provide the means to save energy and better distribute performance, but it is the
responsibility of the application programmer (or user) to ask for reasonable performance levels, e.g., a video game that is rendering video in real time can probably have its performance capped somewhere between 30 and 60 fps.

In summary, the proposed framework is able to greatly improve the resources’ distribution in a complex heterogeneous system, achieving the specified target performances and saving energy while running diverse applications (single- and multi-threaded) without a prior offline evaluation. There are however some points which can be identified as open questions, still worth studying, these are discussed bellow, in Future work.

6.1 Future work

Several issues were left open during the development of the proposed performance and energy-aware real-time scheduling for heterogeneous embedded systems framework.

One of such issues is the selection of the periods used to call the different controllers. The tests were run with fixed periods which were empirically determined, but, as was discussed, there are conditions that should be met before a new call of the controllers is possible. Due to the dependencies the Share controller must respect before updating the shares assigned to tasks in a core, the best option is probably to configure the controller to use a period that is somehow related to the current running applications’ performances.

There is also no compensation method to increase the performance temporarily if it has been too low for some portion of time (or vice-versa). If an application requires a strict average performance level this issue becomes significant, but unfortunately it is also the source of even further instabilities on the controller since it will be one more element on the feedback loop. A more prominent issue which may be tackled first is to guarantee (or at least further improve) performance fairness across applications (as opposed to enforce it solely per core). The Distributed Weighted Round-Robin presented in [27, 28] may be adapted (even for heterogeneous platforms) to consider performance as a metric instead of a task’s weight, but this would come at the cost of portability requiring some further system integration.

Applications which require synchronism between threads require a careful examination since a low performing thread may bottleneck the whole execution. These situations are not accounted for in the model and either the migration controller or the programs themselves should be prepared to foresee and avoid such situations.

Finally, thermal throttling situations may be improved since the current implementation tries to contradict the system behavior in extreme situations, which exists for safety reasons. The controller should be made aware of thermal emergency scenarios and react accordingly by forcefully throttling the performance until that emergency has passed. This would reduce the additional energy consumption seen when a target performance is not feasible under the framework-controlled system.

As for controller implementation and functionality expansion, new architectures are starting to allow per core frequency control, meaning that frequency could be managed at the level of the core without dragging tasks outside a specific core to different performance levels.
References


Countable events on the ARMv7-A ISA
Table A.1: Countable events through performance counters on the ARMv7-A ISA \[21\]

<table>
<thead>
<tr>
<th>Number</th>
<th>Event counted</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Software increment of the Software Increment Register</td>
</tr>
<tr>
<td>0x01</td>
<td>Instruction fetch that causes a Level 1 instruction cache refill</td>
</tr>
<tr>
<td>0x02</td>
<td>Instruction fetch that causes a Level 1 instruction TLB refill</td>
</tr>
<tr>
<td>0x03</td>
<td>Memory Read or Write operation that causes a Level 1 instruction TLB refill</td>
</tr>
<tr>
<td>0x04</td>
<td>Memory Read or Write operation that causes a Level 1 data cache access</td>
</tr>
<tr>
<td>0x05</td>
<td>Memory Read or Write operation that causes a Level 1 data TLB refill</td>
</tr>
<tr>
<td>0x06</td>
<td>Memory-reading instruction executed</td>
</tr>
<tr>
<td>0x07</td>
<td>Memory-writing instruction executed</td>
</tr>
<tr>
<td>0x09</td>
<td>Exception taken</td>
</tr>
<tr>
<td>0x0A</td>
<td>Exception return executed</td>
</tr>
<tr>
<td>0x0B</td>
<td>Instruction that writes to the Context ID register</td>
</tr>
<tr>
<td>0x0C</td>
<td>Software change of program counter</td>
</tr>
<tr>
<td>0x0D</td>
<td>Immediate branch instruction executed</td>
</tr>
<tr>
<td>0x0F</td>
<td>Unaligned load or store</td>
</tr>
<tr>
<td>0x10</td>
<td>Branch mispredicted or not predicted</td>
</tr>
<tr>
<td>0x11</td>
<td>Cycle count; the register is incremented on every cycle</td>
</tr>
<tr>
<td>0x12</td>
<td>Predictable branch speculatively executed</td>
</tr>
<tr>
<td>0x13</td>
<td>Data memory access</td>
</tr>
<tr>
<td>0x14</td>
<td>Level 1 instruction cache access</td>
</tr>
<tr>
<td>0x15</td>
<td>Level 1 data cache write-back</td>
</tr>
<tr>
<td>0x16</td>
<td>Level 1 data cache write-back</td>
</tr>
<tr>
<td>0x17</td>
<td>Level 2 data cache refill</td>
</tr>
<tr>
<td>0x18</td>
<td>Level 2 data cache write-back</td>
</tr>
<tr>
<td>0x19</td>
<td>Bus access</td>
</tr>
<tr>
<td>0x1A</td>
<td>Local memory error</td>
</tr>
<tr>
<td>0x1B</td>
<td>Instruction speculatively executed</td>
</tr>
<tr>
<td>0x1C</td>
<td>Instruction write to TTBR</td>
</tr>
<tr>
<td>0x1D</td>
<td>Bus cycle</td>
</tr>
<tr>
<td>0x1E-0x3F</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
Cycles to complete memory micro-benchmark on the Cortex-A7 and Cortex-A15
<table>
<thead>
<tr>
<th>Cluster</th>
<th>Real Frequency</th>
<th>Number of cycles by data size</th>
<th>8 KB</th>
<th>16 KB</th>
<th>128 KB</th>
<th>256 KB</th>
<th>512 KB</th>
<th>1 MB</th>
<th>4 MB</th>
<th>8 MB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L1</td>
<td>L2</td>
<td>Main memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A7</td>
<td>500 MHz</td>
<td>29735000</td>
<td>29756000</td>
<td>30155250</td>
<td>30211250</td>
<td>31354250</td>
<td>32628250</td>
<td>33086750</td>
<td>33083000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>700 MHz</td>
<td>29690150</td>
<td>29688050</td>
<td>30069550</td>
<td>30165100</td>
<td>31686200</td>
<td>33338550</td>
<td>33918150</td>
<td>33899950</td>
<td></td>
</tr>
<tr>
<td></td>
<td>900 MHz</td>
<td>29607750</td>
<td>29597400</td>
<td>29999700</td>
<td>30073950</td>
<td>31878000</td>
<td>33924150</td>
<td>34641450</td>
<td>34623450</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0 GHz</td>
<td>29601000</td>
<td>29585000</td>
<td>29985500</td>
<td>30065000</td>
<td>31994000</td>
<td>34187500</td>
<td>34953500</td>
<td>34929500</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.2 GHz</td>
<td>29579400</td>
<td>29572200</td>
<td>29988000</td>
<td>30063000</td>
<td>32254800</td>
<td>34820400</td>
<td>35740200</td>
<td>35740200</td>
<td></td>
</tr>
<tr>
<td>A15</td>
<td>800 MHz</td>
<td>21147600</td>
<td>21148000</td>
<td>22998800</td>
<td>23000000</td>
<td>23000000</td>
<td>23000000</td>
<td>24662400</td>
<td>24835200</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0 GHz</td>
<td>21121000</td>
<td>21123000</td>
<td>22985000</td>
<td>22986000</td>
<td>22985000</td>
<td>22991500</td>
<td>24906000</td>
<td>24985500</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.2 GHz</td>
<td>21122400</td>
<td>21183600</td>
<td>22975200</td>
<td>22974000</td>
<td>22972800</td>
<td>22986000</td>
<td>25203600</td>
<td>25395600</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.4 GHz</td>
<td>21105700</td>
<td>21163800</td>
<td>22957200</td>
<td>22972600</td>
<td>22974000</td>
<td>22988000</td>
<td>25589900</td>
<td>25959500</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.6 GHz</td>
<td>21111200</td>
<td>21169600</td>
<td>22961600</td>
<td>22958400</td>
<td>22960000</td>
<td>22969600</td>
<td>26041600</td>
<td>26480800</td>
<td></td>
</tr>
</tbody>
</table>