

Boosting Energy-Efficiency of Heterogeneous Embedded Systems via Game Theory

David Manuel Carvalho Pereira

Thesis to obtain the Master of Science Degree in
Electrical and Computer Engineering

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

Examination Committee

Chairperson: Doutor Gonçalo Nuno Gomes Tavares
Supervisor: Doutor Aleksandar Ilic
Members of the Committee: Doutor João Nuno De Oliveira e Silva

November, 2016

Acknowledgments

I would like to thank Doutor Aleksandar Ilic and Doutor Leonel Sousa for giving me the opportunity to perform this thesis as well as their guidance throughout its development. Furthermore, I want to thank all the support and motivation provided by my family and friends during this period. Finally, I want to thank INESC-ID and IST for all the resources available to perform this work.

Abstract

Nowadays it is possible to observe a change in our lives through the use of mobile devices like smartphones, tablets, among others. These devices are evolving at a high pace allowing users to have a more quickly and efficiently user experience. The ever-growing demand of devices with better performance and efficiency led mobile embedded systems to become heterogeneous devices with higher computational power and energy efficiency levels. However, it seems that these energy limited devices, are consuming more energy that it is required in order to meet the performance requirements, leading to discharge the battery more rapidly.

This thesis aims to study and develop a new energy-aware task scheduling approach for heterogeneous embedded systems, based on Game Theory, in order to reduce the overall energy consumption of the device. The proposed energy-aware game-theoretic scheduling approach combines an Auction based selection and the Nash Equilibrium concept from Non-Cooperative Game Theory. It develops a game, where players (processor's cores) compete with each other in order to acquire the tasks/applications by bidding the necessary energy consumption to execute them. Based on this game approach, the scheduler receives the player's bids and selects the player that placed the lowest one, which means that it can execute the task with the lowest energy consumption among all other players.

The dynamic energy-aware game-theoretic scheduling framework herein proposed has been implemented on ARM Versatile Juno r2 Development Platform, experimentally evaluated and compared with the available ARM big.LITTLE scheduling approaches. The conducted evaluation revealed that the proposed framework can achieve energy savings of up to 36%, 32% and 22% when compared with the Linaro's kernel 3.10, Global Task Scheduling and Energy-Aware Scheduling ARM big.LITTLE approaches, respectively.

Keywords

Game Theory; Global Task Scheduling; ARM big.LITTLE; Mobile Devices; Heterogeneous Embedded Systems; Energy-Efficient;

Resumo

Hoje em dia é possível observar uma alteração nas nossas vidas devido à utilização de dispositivos móveis como os smartphones, tablets, entre outros. Estes dispositivos estão a evoluir a um ritmo elevado, possibilitando aos utilizadores uma experiência de utilização cada vez mais rápida e eficiente. A procura constante de dispositivos com melhor desempenho e eficiência levou a que os sistemas embebidos móveis evoluíssem para dispositivos heterogéneos com maior capacidade computacional e níveis superiores de eficiência energética. No entanto, verifica-se que estes dispositivos com uma capacidade limitada de energia, consomem mais energia do que a necessária para satisfazer os requisitos de desempenho, fazendo com que a bateria dos mesmos descarregue mais rapidamente.

Esta tese tem como objetivo o estudo e desenvolvimento de um novo método de agendamento de tarefas em processadores heterogéneos, baseado na Teoria dos Jogos, de forma a reduzir o consumo de energia total nos mesmos. O método de agendamento de tarefas proposto combina uma abordagem de leilão com o conceito fundamental da Teoria de Jogos não cooperativa, o Equilíbrio de Nash, de forma a desenvolver um jogo onde os jogadores (cores dos processadores) competem entre si para adquirir tarefas/aplicações através da licitação do valor de consumo de energia necessário para as executar. Com base neste método, o agendador de tarefas recebe as licitações de cada jogador e atribui a tarefa ao jogador que tiver a menor licitação, o que significa, que esse jogador consegue executar a tarefa tendo um consumo de energia menor que todos os outros jogadores.

O método de agendamento de tarefas proposto nesta tese foi implementado na plataforma de desenvolvimento ARM Versatile Juno r2 de forma a avaliar experimentalmente o seu funcionamento, bem como compará-lo com os métodos de agendamento disponíveis na tecnologia ARM big.LITTLE. A avaliação realizada revelou que o método proposto pode alcançar poupanças de energia até 36%, 32% e 22% quando comparado com os métodos de agendamento da tecnologia ARM big.LITTLE, nomeadamente, o método presente no kernel 3.10 da Linaro, o agendamento global de tarefas e o agendamento com consciência energética, respetivamente.

Palavras Chave

Teoria dos Jogos; Agendamento Global de Tarefas; ARM big.LITTLE; Dispositivos Móveis; Sistemas Embebidos Heterogéneos; Eficiência Energética;

Contents

Abstract	iii
Resumo	v
List of Figures	x
List of Tables	xi
List of Acronyms	xiii
1 Introduction	1
1.1 Motivation	2
1.2 Objectives	2
1.3 Main contributions	3
1.4 Outline	3
2 ARM big.LITTLE heterogeneous platform	5
2.1 ARM big.LITTLE Versatile Express Juno r2 Architecture	6
2.1.1 ARMv8-A Instruction Set Architecture (ISA)	7
2.1.2 Cortex-A72 and Cortex-A53 Microarchitecture	7
2.2 ARM Juno r2 Performance Measure Unit	9
2.2.1 APB energy meters registers	10
2.2.2 PAPI performance counters	10
2.3 ARM Scheduling approaches	12
2.3.1 Cluster Migration	12
2.3.2 In Kernel Switching (Central Processing Unit (CPU) Migration)	13
2.3.3 Global Task Scheduling (Global Task Scheduling (GTS))	14
2.3.4 Energy-Aware Scheduling (Energy-Aware Scheduling (EAS))	15
2.3.5 Scheduling in ARM big.LITTLE	17
2.4 Dynamic Voltage and Frequency Scaling	17
2.4.1 Linux CPUFreq governors	18
2.4.2 Linux scheduling policies	18
2.5 Summary	19
3 State of the Art: Scheduling based on Game Theory	21
3.1 Game Theory	22
3.1.1 The Prisoner's Dilemma	22

3.1.2	Non-Cooperative Game Theory and Nash Equilibrium Background	23
3.1.3	Cooperative Game Theory and Nash Bargaining Solution Background	24
3.2	State of the Art: Scheduling based on Game Theory	24
3.2.1	Problem Definition	25
3.2.2	Game theoretic approaches	26
3.3	Summary	31
4	Framework	33
4.1	Framework general overview	34
4.1.1	Auction based approach	35
4.1.2	Game theoretic approach	36
4.2	Framework implementation in ARM Juno r2 board	43
4.3	Time and Power prediction for several frequencies	45
4.3.1	Task execution time	46
4.3.2	Task instantaneous power consumption	47
4.4	Summary	49
5	Experimental Evaluation	51
5.1	Experimental setup	52
5.2	Benchmarks	54
5.3	Experimental results	56
5.4	Summary	64
6	Conclusions	65
6.1	Future work	67
A	PMU events on ARMv8-A architecture	73
B	Cortex-A53 PMU events	77
C	Cortex-A72 PMU events	81
D	Available PAPI events on ARM Juno r2 platform	85
E	Description of each successfully compiled benchmark	89
F	Benchmark's experimental values	93

List of Figures

2.1	ARM Juno r2 SoC	6
2.2	ARM architecture improvements	7
2.3	Cortex-A53 micro-architecture	8
2.4	Cortex-A72 micro-architecture	8
2.5	PMU block diagram	9
2.6	PAPI architecture	11
2.7	Cluster Migration diagram	13
2.8	CPU Migration diagram	13
2.9	Global Task Scheduling diagram	14
2.10	Scheduler idle-state awareness	15
2.11	Current situation with DVFS support in Linux/GTS	16
2.12	New scheduler-driven DVFS (sched-DVFS)	16
2.13	EAS energy model	17
2.14	Linux scheduling policies, execution example	19
3.1	The Prisoner's Dilemma	23
3.2	Auction game environment	25
3.3	Task Agent's Algorithm	26
3.4	Core Agent's in round K	27
3.5	Pseudocode of the Non-Cooperative/Cooperative approach algorithm	28
3.6	Pseudocode of DVFS Algorithm part	29
3.7	NBS-EATA Algorithm	30
3.8	Unilaterally maximization function	30
3.9	Kernel iterative process	31
4.1	General structure of the proposed framework	34
4.2	Fluxogram of the auction approach and bidding process	35
4.3	Individual utility function, usage example	37
4.4	Global utility function, usage example	40
4.5	Other players utility function, usage example	41
4.6	Player's pseudocode to compute the task's bid	42
4.7	Individual utility function adaptation, usage example	44

4.8	Pseudocode of the Scheduler's algorithm	45
4.9	Comparison and variation of the CPI tendency between MEM_bound and CPU_bound applications	47
5.1	ARM Juno r2 board available connection ports	53
5.2	Comparison and variation of the CPI tendency between MEM_bound and CPU_bound applications	63
5.3	Comparison of run-time task migrations between the proposed framework and ondemand governor	63

List of Tables

2.1	APB energy meters registers	10
2.2	Cortex-A72, Cortex-A53, and MALI-T624 GPU maximum operating frequencies	18
5.1	Available ARM Juno r2 software's combinations	53
5.2	Successfully compiled benchmarks and respective configuration	56
5.3	Cycles per instruction tendency on different benchmarks.	56
5.4	Evaluation of power consumption prediction for the Blacksholes benchmark	58
5.5	Evaluation of execution time estimation for MEM_bound task	59
5.6	Evaluation of MEM_bound CPI prediction.	60
5.7	Evaluation of MEM_bound task energy consumption	60
5.8	Experimental results for each benchmark combination used to evaluate the proposed framework	61
A.1	PMU events on ARMv8-A architecture	75
B.1	Cortex-A53 PMU events	79
C.1	Cortex-A72 PMU events	83
D.1	Available PAPI events on ARM Juno r2 platform	87
E.1	Description of each successfully compiled benchmark	91
F.1	Benchmark's solo values for each frequency	95

List of Acronyms

APB	Advanced Peripheral Bus
ATMI	Analytical Model of Temperature in Microprocessors
CPI	Cycles Per Instruction
CPU	Central Processing Unit
DVFS	Dynamic Voltage and Frequency Scaling
EAS	Energy-Aware Scheduling
EDF	Earliest Deadline First
GPU	Graphics Processing Unit
GTFTES	Generalized Tit-For-Tat Energy-aware Scheduling
GTS	Global Task Scheduling
IKS	In Kernel Switching
LSK	Linaro Stable Kernel
LTK	Linaro Tracking Kernel
MP	Multi-Processing
NBS	Nash Bargaining Solution
NE	Nash Equilibrium
OPP	Operating Performance Point
PELT	Per-Entity Load Tracking
PMU	Performance Monitoring Unit
SoC	System on a Chip
WCET	Worst Case Execution Time

1

Introduction

Contents

1.1	Motivation	2
1.2	Objectives	2
1.3	Main contributions	3
1.4	Outline	3

The paradigm of achieving more computational power has led to an evolution in the current computing systems. The processors began to have multiple cores, which allow tasks to be executed in parallel. Nowadays, it can be seen systems with dual core, quad core processor or even more. Embedded systems have become also heterogeneous, with processors or cores with different characteristics.

Heterogeneous embedded systems are being used almost everywhere. An example, is the higher number of mobile devices (e.g. smartphones, tablets, etc.) produced every year. Those mobile devices are evolving every year aiming to achieve better performance, speed, power consumption and others aspects which can lead to better utilization conditions for the user. As it can be seen nowadays, smartphones are being faster and faster in order to execute more complex tasks and their performance are increasing but they are requiring higher power consumption, which leads to the batteries start to unload faster and faster.

There is a relation between performance and the power consumption of systems. For achieving higher performance levels, it is also necessary to increase power consumption. To solve this problem, it must be found the best way to achieve a good performance level, while at the same time the energy consumption is reduced to the minimum. This will lead to reduce the energy consumption of the system and also to save battery charge.

1.1 Motivation

The challenge to supporting the ever growing demand of performance keeping the energy consumptions at low acceptable levels is difficult to solve, but when good results are achieved it has real impact and can really motivate the development of better solutions in the future. It can be used not only in the current technology but also to be important for the future embedded systems.

This challenge can be partially solved by scheduling tasks to the several cores of a processor in an optimal way. In order to do so, an energy-aware scheduler must be developed. The solution herein proposed is based on game theory concepts. Game theory is "the study of mathematical models of conflict and cooperation between intelligent rational decision-makers" [1], and is mainly used in economics, political science, as well as logic, computer science, and biology. Nowadays in computer science there are many researchers that apply game theory to solve problems and challenges on different areas, [2], [3], [4], [5].

The game theory will allow to develop a solution for dynamic task scheduling based on a game approach, where the cores are the players that are competing for some resources, which are tasks/applications. In this game approach the players must have some strategies to play the "game" and the scheduler must decide the best way to schedule the tasks to the cores in order to reduce the overall energy consumption of the system.

1.2 Objectives

The first objective of this thesis is to study how game theory can be used to solve the problem of task scheduling on heterogeneous cores. Then using the concepts of game theory, other main goal

is to design an embedded solution capable of exploring all available computational resources simultaneously in heterogeneous embedded systems, such as the big.LITTLE from ARM. The solution herein proposed will be an energy-aware scheduler capable of achieving energy savings as well as acceptable performance levels, which is based on an energy-efficient game-theoretic approach that must use overall energy consumption as a performance governing metric.

1.3 Main contributions

An energy-aware game-theoretic scheduling approach is proposed in this thesis. The developed scheduler takes into account performance (event counters) and energy (meter registers) in order to characterize the task and estimate the necessary energy consumption to execute it on a specific core. Based on this information, the proposed scheduling approach uses an auction game approach together with the Nash Equilibrium concept in order to select the best core and frequency to execute the task, which leads to minimize the overall energy consumption of the device. The developed approach allowed to achieve energy savings up to 36%, 32% and 22% when compared with the Linaro's kernel 3.10, Global Task Scheduling and Energy-Aware Scheduling approaches, respectively, which are the default ARM's scheduling approaches available on the ARM Juno r2 heterogeneous embedded system.

1.4 Outline

This thesis is organized in six chapters. Chapter 2 presents the architecture of the ARM Juno r2 big.LITTLE heterogeneous platform, and the available performance monitoring units. It is also presented the ARM big.LITTLE scheduling approaches, as well as some power-managements techniques.

Chapter 3 explains the two main branches of Game Theory and the respective main concepts behind them. It is also introduced the state of art scheduling approaches based on Game Theory, and discussed where the focus must reside to develop each approach.

In Chapter 4 the proposed framework is presented. The chapter starts with a general overview of how auction approach as well as the utility function from Nash Equilibrium concept are formulated. It is also presented the necessary adjustments on the framework to be implemented in the ARM Juno r2 platform. Finally, it is explained how the instantaneous power consumption and task execution times are estimated, as well as how they are predicted to other frequencies in order to avoid exhaustive search.

Chapter 5 presents the experimental setup as well as a description of the benchmarks used to evaluate the proposed framework. It is also evaluated the power consumption and execution time estimations/predictions when compared with the experimentally measured values. Finally, it is presented a thorough evaluation of the proposed framework when several benchmark combinations are selected, in which the spent overall energy consumption of the developed energy-aware scheduler is compared with the available ARM big.LITTLE scheduling approaches.

The conclusions are then presented in Chapter 6, followed by the proposed future work.

2

ARM big.LITTLE heterogeneous platform

Contents

2.1 ARM big.LITTLE Versatile Express Juno r2 Architecture	6
2.1.1 ARMv8-A Instruction Set Architecture (ISA)	7
2.1.2 Cortex-A72 and Cortex-A53 Microarchitecture	7
2.2 ARM Juno r2 Performance Measure Unit	9
2.2.1 APB energy meters registers	10
2.2.2 PAPI performance counters	10
2.3 ARM Scheduling approaches	12
2.3.1 Cluster Migration	12
2.3.2 In Kernel Switching (Central Processing Unit (CPU) Migration)	13
2.3.3 Global Task Scheduling (Global Task Scheduling (GTS))	14
2.3.4 Energy-Aware Scheduling (Energy-Aware Scheduling (EAS))	15
2.3.5 Scheduling in ARM big.LITTLE	17
2.4 Dynamic Voltage and Frequency Scaling	17
2.4.1 Linux CPUFreq governors	18
2.4.2 Linux scheduling policies	18
2.5 Summary	19

ARM big.LITTLE technology combines high-performance and energy-efficiency in ARM CPU cores to deliver peak-performance capacity and increased parallel processing performance while having low-power consumption. The latest big.LITTLE software and platforms can increase performance by 40% in highly threaded workload and save energy consumption of CPU by 75% in low to moderate performance scenarios. ARM big.LITTLE technology enables mobile System on a Chip (SoC) to be designed for new levels of peak performance.

In this section will be presented the architecture of the ARM Versatile Express Juno r2 (V2M-Juno r2) heterogeneous platform [6] that is used in this work, the architectures of the cores are presented as well as the available performance monitoring unit. Finally, will be presented the ARM big.LITTLE scheduling approaches and power-management techniques.

2.1 ARM big.LITTLE Versatile Express Juno r2 Architecture

The compute subsystem of the ARM Versatile Express Juno r2 board is composed mainly by an high-performance dual-core Cortex-A72 (big) cluster, an energy efficient quad-core Cortex-A53 (LITTLE) cluster and a quad-core Mali-T624 Graphics Processing Unit (GPU) cluster. Each CPU cluster and GPU cluster has a Level 2 Cache that is used to share data between each core in the cluster. The integration of these three clusters is possible through the CoreLink CCI-400 Cache Coherent Interconnect. This board also has an external user memory with 8GB on-board DDR3L that works at 800MHz, a Performance Monitoring Unit (PMU), and an Advanced Peripheral Bus (APB) designed for low bandwidth control accesses to the energy meter registers. In Figure 2.1 is represented the ARM Juno r2 SoC architecture.

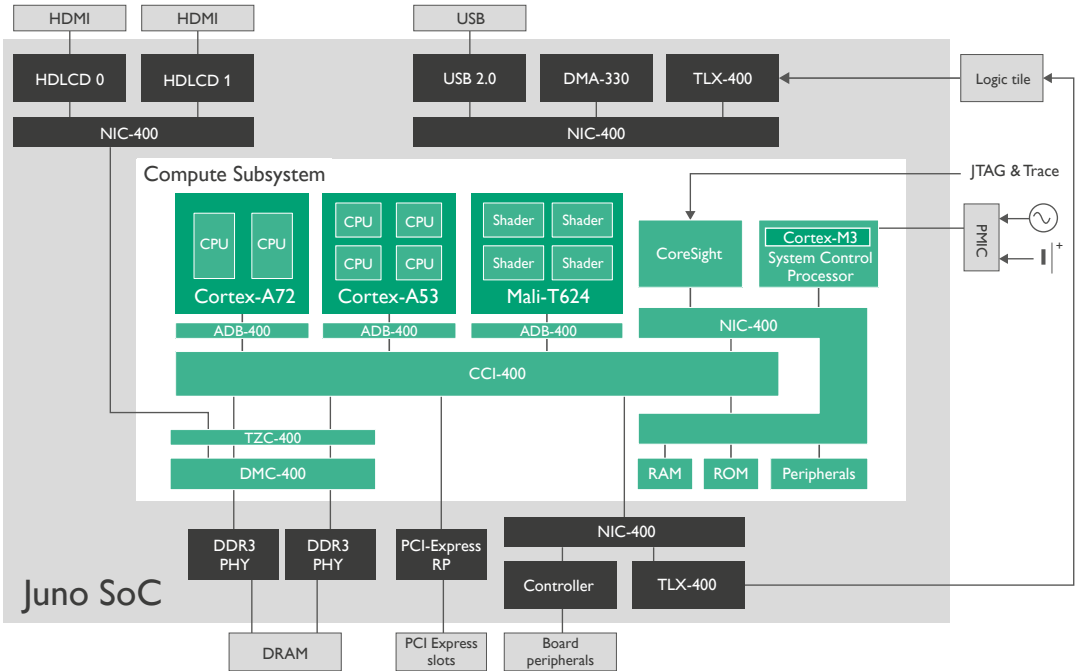


Figure 2.1: ARM Juno r2 SoC, [arm.com, 2015].

2.1.1 ARMv8-A Instruction Set Architecture (ISA)

ARMv8-A introduces 64-bit architecture support to the ARM architectures and include 64-bit general purpose registers, SP (stack pointer), PC (program counter), data processing, extended virtual addressing and NEON technology. ARMv8-A architecture has two main execution states (Figure 2.2), the 64-bit execution state AArch64 and the 32-bit execution state AArch32 [7]. On one hand, the AArch32 state supports two instruction sets:

- The A32 (or ARM) is a fixed-length (32-bit) instruction set, enhanced through the different architecture variants;
- The T32 (Thumb) instruction set provides a subset of the most commonly used 32-bit ARM instructions which have been compressed into 16-bit wide opcodes. During execution, these 16-bit instructions are decompressed to full 32-bit ARM instructions in real time without performance loss;

On the other hand, the AArch64 state supports the A64 instruction set, which is a 64-bit fixed-length instruction set that offers similar functionality to the A32 and T32 instruction sets.

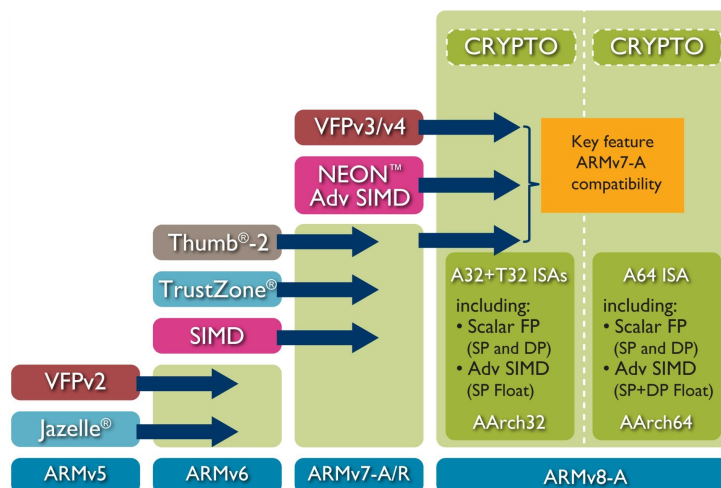


Figure 2.2: ARM architecture improvements, [arm.com, 2014].

As seen in Figure 2.2, ARM ISAs and architectures are constantly being improved in order to meet the increasing demands of high end application developers, while retaining the software's backwards compatibility.

2.1.2 Cortex-A72 and Cortex-A53 Microarchitecture

In ARM Juno r2 board, both Cortex-A72 and Cortex-A53 have the same ARMv8-A ISA, which means that applications can be executed in (or migrated to) each core without problems. ARM have developed the Cortex-A72 to have higher computational power than the Cortex-A53, which is more energy efficient than the Cortex-A72. These two processors combined provides heterogeneity to the device, high-performance and energy-efficiency when used the ARM big.LITTLE technology.

The Cortex-A53 is an in-order, dual-issue processor with a pipeline of 8 stages and has: 2 Integer ADDs units, 1 Integer MUL unit, 1 Load/Store units and 1 FP/NEON ALUs. The micro-architecture and pipeline length of this processor are the same as the Cortex-A7 [8], which is presented in Figure 2.3.

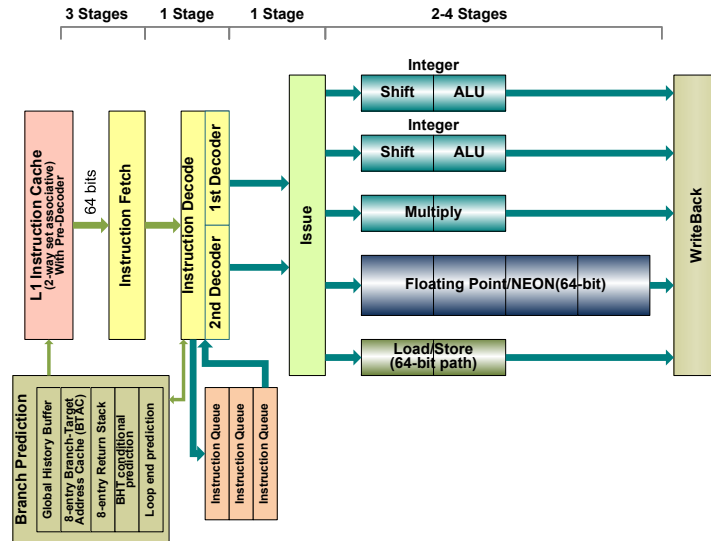


Figure 2.3: Cortex-A53 micro-architecture [Hiroshige Goto, pc.watch.impress.co.jp, 2013].

The Cortex-A72 is an out-of-order, triple-issue processor with a pipeline of 18 stages and has: 2 Integer ADDs units, 1 Integer MUL unit, 2 Load/Store units, 1 Branch unit and 2 Floating-point(FP)/NEON ALUs [9]. The block diagram of this processor is presented in Figure 2.4.

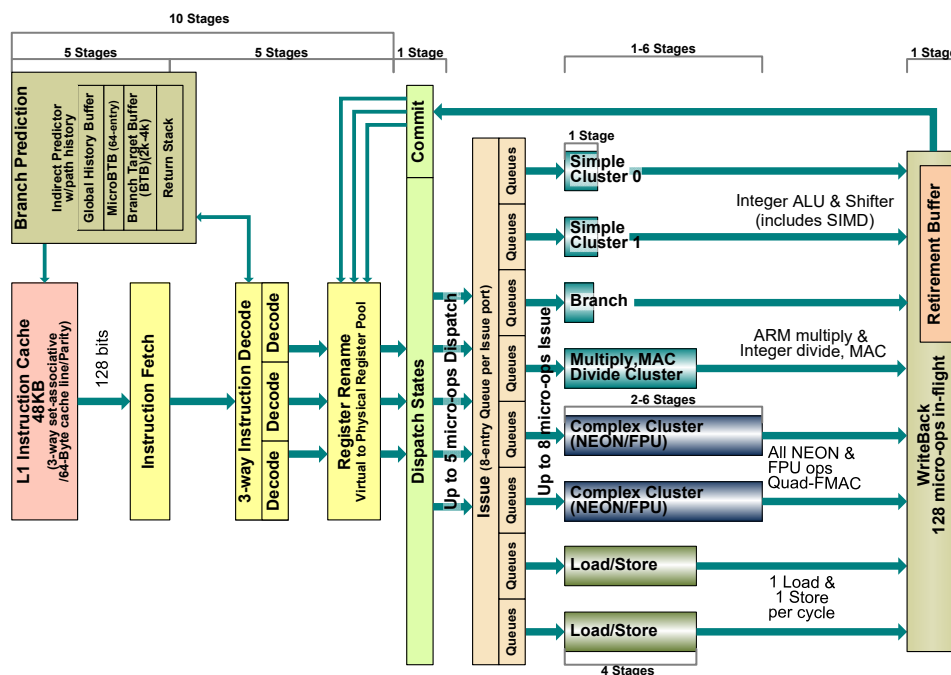


Figure 2.4: Cortex-A72 micro-architecture [Hiroshige Goto, pc.watch.impress.co.jp, 2015].

2.2 ARM Juno r2 Performance Measure Unit

Both Cortex-A53 and Cortex-A72, includes logic to gather information about performance of the processor at runtime, based on the PMUv3 architecture. This information can be used, for example, to debug or profile the developed code. The user can set up to 6 performance counters to read the desired events in order to obtain that information. Each counter can count any of the events available in ARMv8-A architecture, such as the number of correctly executed instructions, miss-predicted branches, data memory accesses, L1 data cache misses. All available events for ARMv8-A architecture are listed in Table A.1 presented in Appendix A. However, both Cortex-A53 and Cortex-A72 processors, also contains other specific events besides the ARMv8-A ones which are listed in Tables B.1 and C.1, in Appendixes B and C, respectively. Each processors's PMU also provides a dedicated cycle counter, besides the 6 performance counters. In Figure 2.5 is presented the PMU block diagram.

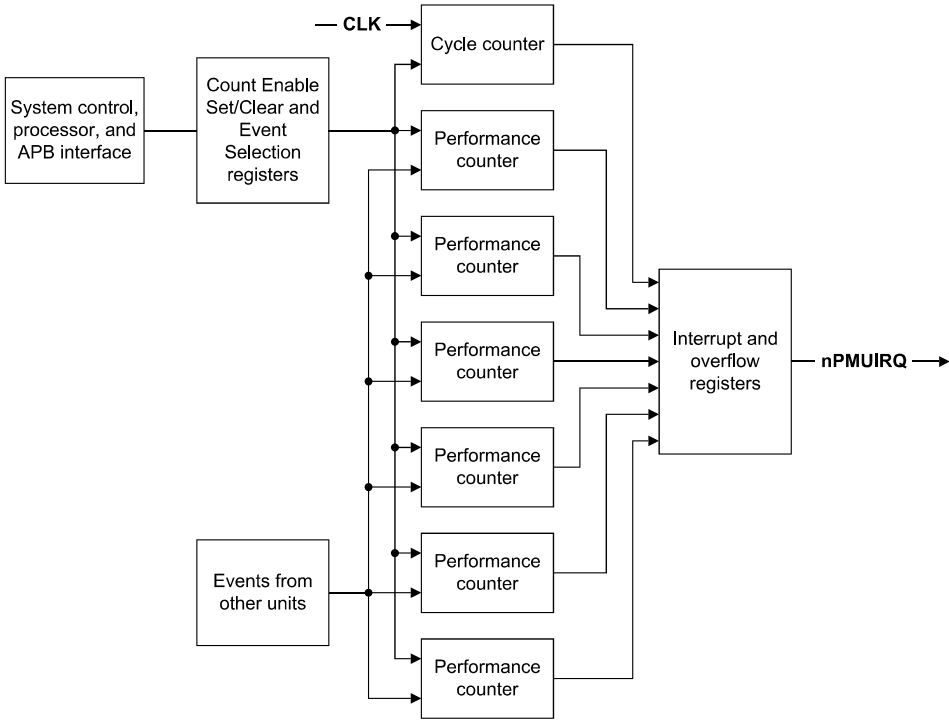


Figure 2.5: PMU block diagram.

These performance counters are not accessible directly in the user space. It is only possible to set and read them through an external debugger, by developing a kernel drive to access the registers directly or by using already existing Performance APIs such as PAPI [10] or OProfile [11]. Mainly due to pipeline effects, the event counts recorded might vary, although, this has a negligible effect unless the counters are enable for a very short period of time. Each processor asserts a signal to the system when an interruption is generated by the PMU. In this thesis, the PAPI performance event counters and the sysfs APB energy meter registers interface are used to configure the PMU.

2.2.1 APB energy meters registers

The APB energy meters registers are updated every $100\mu s$ and measures the instantaneous power consumption, cumulative energy consumption, instantaneous current consumption, and instantaneous voltage supply, of the Cortex-A72 cluster, Cortex-A53 cluster, Mali-T624 GPU cluster, and the hardware of the ARM Juno r2 SoC outside the clusters. Both Linaro Tracking Kernel (LTK) and Linaro Stable Kernel (LSK), which are available to configure the ARM Juno r2 platform, implement a standardised Hardware Monitoring (hwmon) Driver interface for reading the APB energy meter registers, which is exposed through sysfs (/sys system folder). The available interface files with the registers values as well as their absolute path are shown in Table 2.1. One can read these files by simple `fopen()` the file and `sscanf()` the value.

Table 2.1: APB energy meters registers of the System (Sys), Cortex-A72 (A72), Cortex-A53 (A53) and GPU.

		X				
		Absolute path	Sys	A72	A53	GPU
Kernel 3.10	Voltage [<i>mV</i>]	/sys/class/hmown/hmownX/in1_input	0	1	2	3
	Energy [μJ]	/sys/class/hmown/hmownX/energy1_input	4	5	6	7
	Power [<i>mW</i>]	/sys/class/hmown/hmownX/power1_input	8	9	10	11
	Current [<i>mA</i>]	/sys/class/hmown/hmownX/curr1_input	12	13	14	15
Kernel 3.18	Voltage [<i>mV</i>]	/sys/class/hwmon/hwmon0/inX_input	3	4	5	6
	Energy [μJ]	/sys/class/hwmon/hwmon0/powerX_input	1	2	3	4
	Power [<i>mW</i>]	-	-	-	-	-
	Current [<i>mA</i>]	/sys/class/hwmon/hwmon0/currX_input	1	2	3	4

In this work, the estimation of the instantaneous power consumption can be done by two ways. First, one can sample multiple instantaneous power consumptions measures, Pow_i , during a period of time and simply compute the average power consumption value. Or, one can read the initial, $Ener_i$, and final, $Ener_f$, accumulated energy consumption register value during a time period, Δt , and compute the average power consumption by normalizing for the time duration. Both ways were implemented, but it was adopted the second one. The first way has the limitation that to read the instantaneous power consumption register of Cortex-A53 Cluster it must be used one of the cores of the Cortex-A72. This must be done in order to be more accurate, because the reading of those registers causes a small increase of the instantaneous power consumption, and so, the average power consumption would be increased by that small increase. On the second way, any core can read the cumulative energy consumption register of each cluster, because the increase of instantaneous power consumption will not affect significantly the cumulative energy consumption value. And also, the register must just be read two times.

2.2.2 PAPI performance counters

The Performance API (PAPI) is an open source project for accessing hardware performance counters available in the processors. These counters are represented as a set of registers that count specific

signals related to the processors's function, which are designated by Events. Generally, these events are used to do performance analysis, for example for identifying the correlation between the code and the underlying architecture, as for example, benchmarking, debugging, monitoring and performance modeling. In high performance computing, this information is useful to detect execution bottlenecks.

PAPI can be divided into two layers of software. The user-space layer, which consists on the API and its support functions. And the kernel-space layer that directly accesses the event counters registers of the processors through a kernel extension or assembly functions. In Figure 2.6 is shown the internal design of the PAPI architecture, in which can be seen explicitly the bridge between the Portable Layer (user-space) and the Machine Specific Layer (kernel-space).

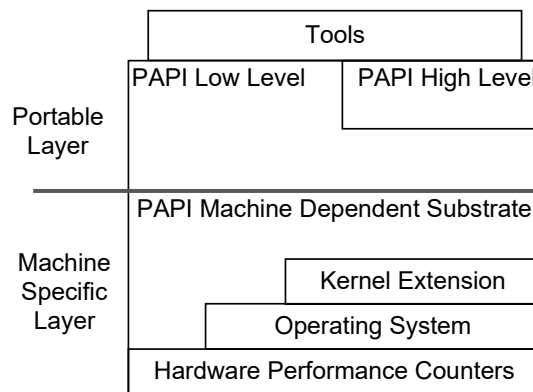


Figure 2.6: PAPI architecture.

In this work, it was used the latest PAPI 5.4.3 version. This version has support just to the ARM Cortex-A53, which means that some of the events in the Cortex-A72 may not be accessible. All the available native events counters on ARM Juno r2 board that are accessible through PAPI are presented in Table D.1 on the Appendix D.

In order to successfully read the event counters in this board, it is necessary to follow 4 steps: start PAPI's library (Algorithm 2.1); set and start the event counters with the desired events (Algorithm 2.2); read the event values to an array (Algorithm 2.3), and stop the event counters (Algorithm 2.4). In the following algorithms is present the necessary C code to execute each step.

In Algorithm 2.1 the PAPI library is initialized. The array of chars contains the names of the desired events, which in the example, are the number of instructions architecturally executed (INST_RETIRED) and the number of clock cycles (CPU_CYCLES), as seen in Algorithm 2.2. All PAPI functions used and present in Algorithms 2.1-2.4 are from the PAPI Low Level. In Algorithm 2.2, these functions are used to translate the native event names into event codes, which will be used to create the event set and configure the event counter registers. In Algorithms 2.4 and 2.3 are shown the functions to read the performance counters to an array with or without stopping the event counter registers, respectively.

Algorithm 2.1 Library initialization

```

1: #include "papi.h"
2: #define NUM_EVENTS 2
3: char *events[NUM_EVENTS];
4: PAPI_library_init( PAPI_VER_CURRENT )

```

Algorithm 2.2 Start PAPI

```
1: int EventSet = PAPI_NULL;
2: int native = 0x0;
3: events[0] = "INST_RETIRED";
4: events[1] = "CPU_CYCLES";
5: PAPI_create_eventset( &EventSet );
6: for ( i = 0; i < NUM_EVENTS; i++){
7:     PAPI_event_name_to_code( events[i], &native );
8:     PAPI_add_event( EventSet, native );
9: }
10: PAPI_start( EventSet );
```

Algorithm 2.3 Read PAPI

```
1: long long values[NUM_EVENTS];
2: PAPI_read( EventSet, values );
3: printf( "INST_RETIRED = %lld \n", values[0] );
4: printf( "CPU_CYCLES = %lld \n", values[1] );
```

Algorithm 2.4 Stop PAPI

```
1: PAPI_stop( EventSet, values );
2: printf( "INST_RETIRED = %lld \n", values[0] );
3: printf( "CPU_CYCLES = %lld \n", values[1] );
```

2.3 ARM Scheduling approaches

In the earlier ARM big.LITTLE software, only the Cluster Migration and the In Kernel Switching (IKS) scheduling approaches were available. These were the first approaches implemented in heterogeneous embedded systems. However, they could not use all the available cores at the same time. ARM have improved the big.LITTLE software and currently there are available the GTS and EAS scheduling approaches. These approaches will now be explained in detail.

2.3.1 Cluster Migration

The Cluster Migration was the first and simplest developed scheduling approach. In this approach, the identically-sized "big" and "LITTLE" clusters are arranged in a way that only one cluster can operate at a time, as shown in Figure 2.7.

Depending on the workload's (tasks/applications) demanding performance, the system can move the workload from one cluster to another. This will lead to execute the workload in the "big" Cluster at higher performance but higher power consumption or in the "LITTLE" Cluster at lower performance and higher energy-efficiency. To migrate the workload between clusters, all the relevant data is passed through the common L2 cache, the source cluster is powered off and the destination one is activated.

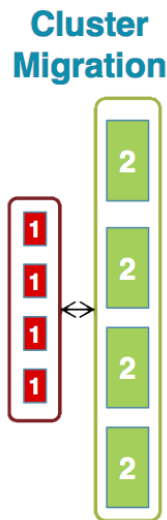


Figure 2.7: Cluster Migration diagram, [anandtech.com, 2013].

2.3.2 In Kernel Switching (CPU Migration)

The IKS scheduling approach is an improvement of the Cluster Migration approach. CPU Migration via in-kernel switcher involves pairing a "big" core with a "LITTLE" core as each pair operates as one "virtual" core. There will exist many "virtual" cores as many identical pairs in the big.LITTLE processor as shown in Figure 2.8.

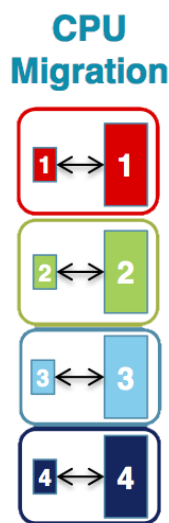


Figure 2.8: CPU Migration diagram, [anandtech.com, 2013].

The system operation in this approach is the same as in the Cluster Migration approach. In each "virtual" core, only one of the "big" or "LITTLE" core is active at a time. Each "virtual" core has an individual Dynamic Voltage and Frequency Scaling (DVFS) subsystem (explained in section 2.4), which selects the supply voltage and frequency level of the core. When the workload's demanding performance reaches a threshold value, the destination core is activated, the running state is transferred to it, the source core is deactivated and the processing continues on the destination core.

The main difference in relation to the Cluster Migration is that each pair big-LITTLE core ("virtual"

core) is visible to the scheduler. This approach is mainly used in symmetrical SoC (e.g. 4 "big" cores and 4 "LITTLE" cores). However, it can be used in non-symmetrical SoC (e.g. 2 "big" cores and 4 "LITTLE" cores) too, where multiple "LITTLE" cores can be paired with the same "big" core, but as it is a more complex arrangement it is not as precise as the Cluster Migration.

2.3.3 Global Task Scheduling (GTS)

In Cluster and CPU Migration approaches just half the cores can be activated simultaneously at the same time. The Global Task Scheduling, also known as heterogeneous Multi-Processing (MP) scheduling approach, was the first approach developed to enable the use of all physical cores at the same time, and so, it is more powerful than the previous big.LITTLE scheduling approaches. In GTS, all the "big" and "LITTLE" cores are seen as individual cores by the scheduler as shown in Figure 2.9. This approach can be used either in symmetrical or non-symmetrical SoCs.

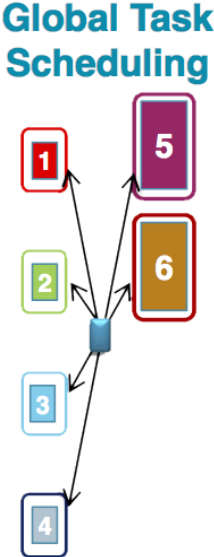


Figure 2.9: Global Task Scheduling diagram, [anandtech.com, 2013].

The GTS approach relies on a Per-Entity Load Tracking (PELT) [12] mechanism with two predefined threshold values (one for each cluster) to perform the task-to-cluster allocations. In GTS, PELT is used to provide an high-level task classification, where tasks with low computational intensity or less priority, such as background tasks, can be assign to the "LITTLE" cores while tasks with high computational intensity or high priority can be performed by the "big" cores.

The scheduler has a fine-grained control in task allocation and migration among the available cores, which leads to reduce kernel overhead, and thus, power savings can be correspondingly increased. The unused cores are idle by the operating system power management mechanisms.

The Global Task Scheduling is supported in V2M-Juno r2 ARM board and is the latest developed approach (of all three) to the current standard multi-cores systems.

2.3.4 Energy-Aware Scheduling (EAS)

The EAS is the new ARM's scheduling approach that is being investigated. This approach enables the use of all physical cores at the same time and is focused on integrating the three separate frameworks in the Linux kernel that are currently only individual subsystems on the GTS: the Linux scheduler, Linux cpuidle and Linux cpufreq. These separated subsystems have their own policy mechanisms that make decisions independently to reduce power and energy consumption. However, these decisions can conflict with each other and therefore affect drastically the device's energy consumption. Integrating them will make the scheduler more consistent with its own decisions in order to increase energy savings.

In this scheduling approach, the cpuidle subsystem has been improved to have energy-awareness. The scheduler is now aware of the idle state of the CPU's. There are mainly three idle-states [13]: Wait for Interrupt (WFI); Core power-down (C1); and Cluster power-down (C2). Each idle-state has different wake-up times, which have different associated energy consumptions. EAS minimizes the wake-up time and energy by waking up the CPU in the shallowest idle-state. In Figure 2.10 is presented an example where Cluster 1 is in idle-state C2 and the Cluster 0 has the CPU 0 in P-state (executing) and CPU 1 in idle-state C1. In this example, both cores of each Cluster have the same micro-architecture. Following the example, the scheduler receives a new task and realizes that it will not fit on CPU 0 because the current operating point is almost fully utilized. The scheduler then compares the idle-states and will assign the task to CPU 1 since it is in the shallowest idle-state, remaining the other cluster in C2 state. This is the fastest response option and the lowest energy consumption necessary.

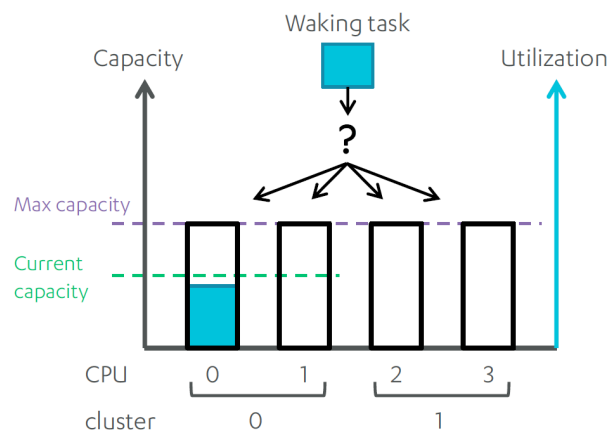


Figure 2.10: Scheduler idle-state awareness, [linaro.org, 2015].

In GTS, the core's utilization is computed based on the existing cpufreq framework, which uses a sampling-based approach to consider cpu time in idle state along with some heuristics to control the CPU Operating Performance Point (OPP). However, there are some disadvantages with this approach, which are illustrated in Figure 2.11. On one hand, in Example 1, the sampling rate is too long which leads to a slow reaction. The OPP is just actualized on the third sample. On the other hand, in Example 2, the average cpu utilization is too low that there is no reaction at all. Also, if sampling is too fast, leads OPP to change for small utilization spikes, and hence increasing the energy consumption.

This sampling-based approach was improved in EAS by using the history of the task, which is stored

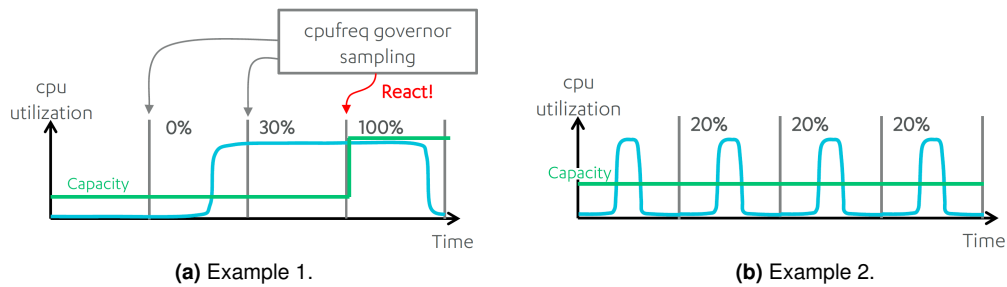


Figure 2.11: Current situation with DVFS support in Linux/GTS scheduling approach, [linaro.org, 2015].

internally as part of the scheduler in the kernel. This stored tracked load of the task is used to immediately switch to the required OPP. With the improved cpufreq subsystem, the new task is scheduled to one core and its capacity is changed immediately, as illustrated in Figure 2.12.

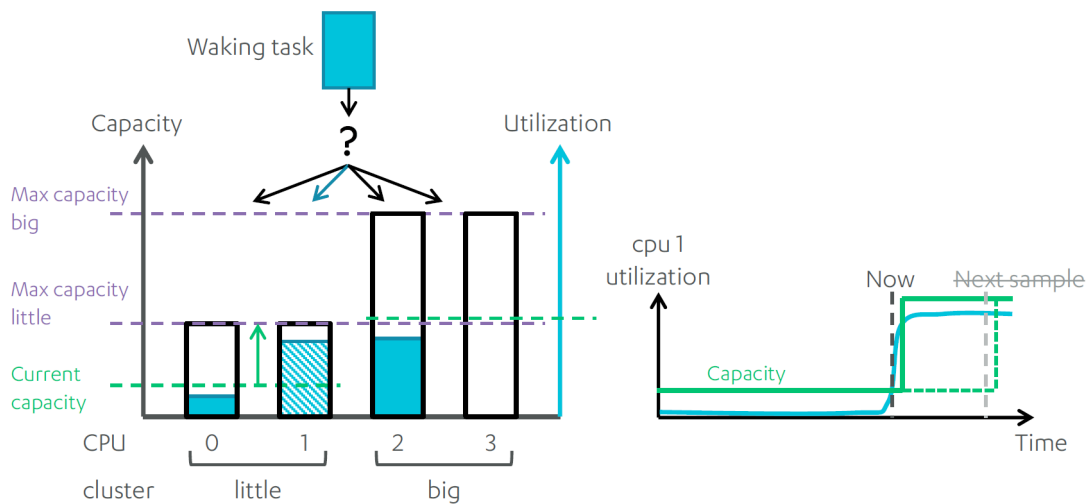


Figure 2.12: New scheduler-driven DVFS (sched-DVFS), [linaro.org, 2015].

Finally, EAS integrates an energy-aware task scheduling policy with the Complete Fair Scheduler (CFS), allowing the kernel to decide at run-time which scheduling decisions leads to lower energy consumption. This Energy-Aware policy select always the CPU that has sufficient spare capacity and provides the smallest energy impact. An example of a run-time decision is shown in Figure 2.13. Following the example, the scheduler has two possible CPUs to schedule the task, CPU 1 and CPU 3. If the task is scheduled to CPU 1, the performance state (P-state) will be moved up for both CPU 0 and CPU 1 (since both CPUs are in the same frequency domain in this example - LITTLE cluster). If the task is scheduled to CPU 3 there is no P-state change but, higher power is used than if scheduled to CPU 1. Based on this, EAS will choose the CPU that will contribute with the small energy increase. Probably, EAS will choose to schedule the task to CPU 1 because, although the small increase of power in CPU 0, it still has better power efficiency than CPU 3.

This Energy-Aware Scheduling approach is not yet released as final product on the market, however, it has been already released for users to test it on the ARM V2M-Juno r2 development platform since July of 2016.

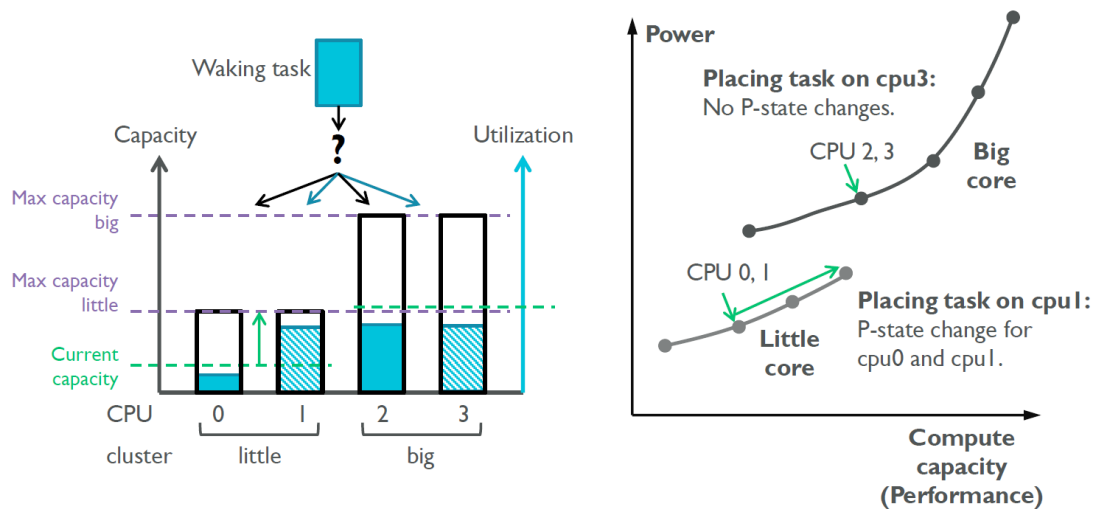


Figure 2.13: EAS energy model, [linaro.org, 2015].

2.3.5 Scheduling in ARM big.LITTLE

Among all presented approaches in this section, the Cluster Migration and IKS are the only officially accepted and main solutions [14]. The major drawback of GTS and earlier approaches is the use of separate subsystems to decide the adequate frequency and voltage levels to execute the tasks without any coordination with the scheduler. Due to this lack of energy-awareness, which may impact the quality of scheduling decisions, the GTS approach was discontinued in favor of the new EAS scheduling approach [14]. Although GTS is discontinued, many researchers work on developing new scheduling approaches based on it. For example, Muthukaruppan et al. [15] developed a methodology based on Price Theory to address energy-efficient task management in an ARM big.LITTLE heterogeneous platform. Other authors have also proposed different scheduling approaches for non-ARM architectures. Ishfaq Ahmad et al. proposed an energy-efficient scheduling approach based on Game Theory to reduce the energy consumption on distributed heterogeneous computational grids [3], and in heterogeneous and homogeneous multi-core processor architectures [2]. Guowei Wu et al. [4] proposed a game theoretic energy-aware scheduling algorithm for multi-core systems to schedule tasks in a way that the overall temperature of the processor is reduced. Some of these approaches will be studied in Chapter 3.

2.4 Dynamic Voltage and Frequency Scaling

DVFS is a commonly-used power-management technique where the processor's clock frequency can be changed, as well as the supply voltage. DVFS is used to reduce the energy consumption on processors when applied to each core (if they support it). The dynamic CPU power dissipation, P , can be mathematically represented as $P = C * V_{dd}^2 * f$, where C is the switched capacitance, V_{dd} is the supply voltage and f is the operating frequency. Reducing the core's frequency, the supply voltage can be reduced too, which leads to power consumption savings on the processor. This technique is commonly used in laptops and other mobile devices, where energy comes from a battery and thus is limited as well as in scheduling approaches that has into account applications with deadline constraints.

Using the DVFS technique on these applications, it is possible to execute them with the lowest power consumption possible while respecting its deadline, and thus, achieving energy savings.

In ARM Juno r2 platform the DVFS technique is only supported at the level of the cluster. Both Cortex-A72 and Cortex-A53 supports 3 different DVFS Operating Performance Point (OPP), while the MALI-T624 GPU only supports 2 different DVFS OPP as shown in Table 2.2.

Table 2.2: Cortex-A72, Cortex-A53, and MALI-T624 GPU maximum operating frequencies [6].

OPP	Voltage [V]	Frequency [MHz]		
		Cortex-A72	Cortex-A53	MALI-T624 GPU
Underdrive	0.8	600	450	450
Nominal drive	0.9	1000	800	600
Overdrive	1.0	1200	950	Not supported

As seen in the EAS approach, the reduction of energy consumption can be achieved through the DVFS subsystems, which will select the best core and frequency OPP to execute the task. In ARM Juno r2 platform these scheduling approaches can be used by selecting the CPUFreq governors.

2.4.1 Linux CPUFreq governors

In ARM Juno r2 board there are available four CPUFreq governors: "userspace"; "ondemand"; "interactive" and "performance". On one hand, the governors "ondemand", "interactive" and "performance" are the ones who use the ARM big.LITTLE scheduling approaches, although each one decides differently how to choose the frequency that should be used. On the other hand, the governor "userspace" allows the user to change the frequency of the clusters as well as to migrate the task between the cores.

The "ondemand" governor is one of the original and oldest governors available on the Linux kernel. Ondemand is commonly chosen by smartphones manufacturers because it provides a smooth performance for the phone by setting the cluster OPP depending on the current CPU usage. By sampling, when the CPU usage reaches the set threshold, the governor will scale the CPU frequency correspondingly.

The CPUfreq governor "interactive" sets the CPU OPP depending on its usage, similar to "ondemand". However, it is more aggressive about scaling the CPU frequency up in response to CPU-intensive activity. This governor checks the CPU load immediately after coming out of idle state, instead of doing the sampling at a specified rate.

The "performance" governor, as the name says, statically sets the cluster to the highest frequency level independently the CPU usage.

2.4.2 Linux scheduling policies

In section 2.4.1 and 2.3 were presented the scheduling approaches and CPUFreq governors that are responsible to decide in which core should be scheduled a task and how is decided the frequency to execute it. However, in each core there are also scheduling policies that could influence these decisions. If more than one task is scheduled to the same core, just one task will be executed at a time while the

remaining ones stay paused. These scheduling policies are used to choose which task should be executed and which ones should stay paused. The three main scheduling policies in ARM Juno r2 platform are: SCHED_OTHER, the standard round-robin time-sharing policy; SCHED_FIFO, a first-in first-out policy, and SCHED_RR, a round-robin policy. In Figure 2.14 are shown some examples of how these policies work, which will now be explained. In these examples, the priority of each task is shown inside the correspondingly parenthesis, i.e. A(5) means that task A has priority 5.

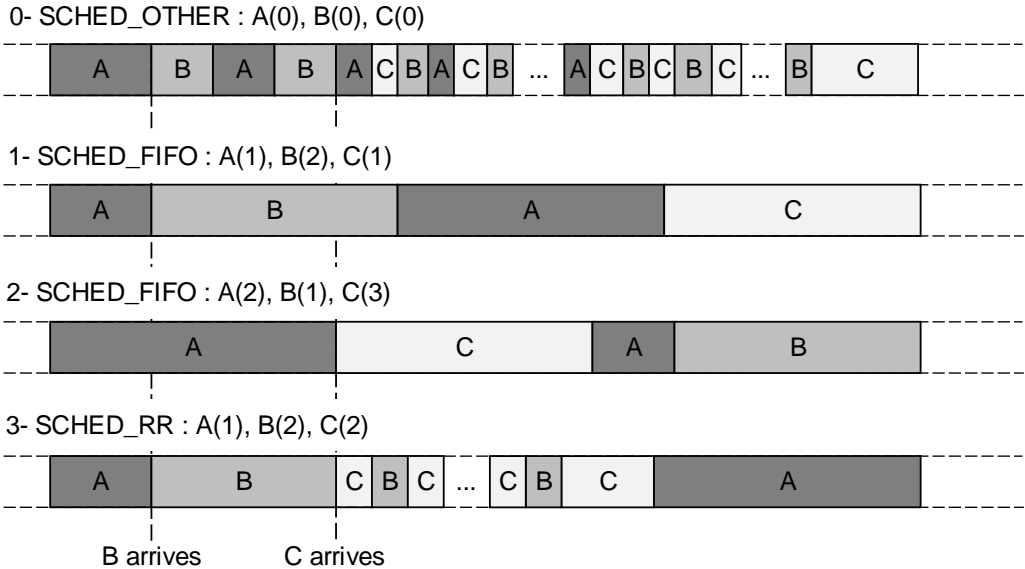


Figure 2.14: Linux scheduling policies, execution example.

SCHED_FIFO and SCHED_RR are "real-time" policies used for special time-critical applications. Tasks with these policies preempt every other task with lower priority, occupying the core until the task has been executed. Using SCHED_FIFO policy, the task with higher priority will preempt the already running task with lower priority. If both tasks have the same priority, then the new arriving task needs to wait until the previous task is finished (Examples 1 and 2). SCHED_FIFO priority value is always higher than 0, and thus it preempts always the tasks with SCHED_OTHER policy (e.g. SSH communications). SCHED_RR policy is similar to SCHED_FIFO, although when multiple tasks have the same priority, each task will run for a specific time-slice, which can be modified by the user (Example 3).

SCHED_OTHER uses a round-robin time-sharing policy. Every task with this policy have priority 0 and ideally should have approximately the same time period to be executed. If exists more than two tasks, the one to be executed is chosen based on the nice level, which is increased for each time quantum that the task is ready to run but is denied by the scheduler. Using this policy, tasks are executed in a equally-fair time-slices (Example 0).

2.5 Summary

In this chapter, the ARM big.LITTLE platform that are used in this thesis was presented. The ARM Juno r2 was analyzed as the ARMv8-A instruction set architecture and the Cortex-A72 (big Cluster) and Cortex-A53 (LITTLE Cluster) architectures. Both Clusters have the same ARMv8-A ISA, which me-

ans that applications can be executed in (or migrated to) each core without problem. The Performance Monitoring Unit present in each cluster was then presented as well as the available performance counters and energy meter registers, which could be read through the PAPI software and the hwmon driver, respectively.

It was also presented the four existing ARM big.LITTLE scheduling approaches, with a main focus on the Global Task Scheduling which, although discontinued it is being used by many researchers in their works. The new Energy-Aware Scheduling approach, which is not currently available in the market as a final product but is already available to be tested on ARM Juno r2 development platform, was also discussed.

Finally, the Dynamic Voltage and Frequency Scaling power-management technique was presented as well as the available Operating Performance Points of each cluster. It was also studied the available Linux CPUFreq governors and scheduling policies available in the ARM Juno r2 platform.

3

State of the Art: Scheduling based on Game Theory

Contents

3.1 Game Theory	22
3.1.1 The Prisoner's Dilemma	22
3.1.2 Non-Cooperative Game Theory and Nash Equilibrium Background	23
3.1.3 Cooperative Game Theory and Nash Bargaining Solution Background	24
3.2 State of the Art: Scheduling based on Game Theory	24
3.2.1 Problem Definition	25
3.2.2 Game theoretic approaches	26
3.3 Summary	31

To develop an energy-aware scheduler based on game theoretic approaches is necessary to start by studying the main concepts presented in Game Theory. In this chapter will be explained the two main branches of game theory as well as the main concepts behind them. Some of these concepts were already used by other researchers to developed their scheduling approaches. The study of these works is not only an important background on the state of the art game theoretic scheduling approaches, but also provides better understanding about the concepts and what are the principal points that must be focus to develop the proposed energy-aware game-theoretic scheduling approach.

3.1 Game Theory

Game theory is "the study of mathematical models of conflict and cooperation between intelligent rational decision-makers" [1], and is mainly used in economics, political science, as well as in logic, computer science, and biology. There are two main branches of game theory: cooperative and non-cooperative game theory.

On one hand, non-cooperative game theory deals with how individuals interact with one another to achieve their own goals. The players make decisions only by themselves seeking always for the best payoff outcome for themselves. On the other hand, cooperative game theory studies how groups of people cooperate and interact with each other to find the optimum strategy to achieve a social goal. This social goal can be maximizing the gains, minimizing losses, maximizing the probability that a specific goal can be reached, etc.

In this section will be presented some fundamental game theory concepts that can be applied to make decisions in task scheduling.

3.1.1 The Prisoner's Dilemma

The Prisoner's Dilemma is a standard example for application of game theory, which shows why two completely "rational" individuals might not cooperate, even if it appears that it is in their best interests to do so. The example consists in two persons that have done some crime and got arrested. Each prisoner is in solitary confinement with no means of speaking to or exchanging messages with the other. The two prisoners will be sentenced but the prosecutors lack sufficient evidence to convict the pair on the principal charge. They plan to sentence both to a year in prison on a lesser charge. Simultaneously, the prosecutors offer each prisoner a bargain. Each prisoner is given the opportunity either:

- to testify that his partner had committed the crime. If he do that he will go free while the partner will get 3 years in prison on the main charge;
- or to cooperate with the other by remaining silent.

If both prisoners betray each other, each of them will be sentenced with 2 years in prison. If both cooperate and remain silent each of them will be sentenced with 1 year in prison. These values are not known by the prisoners but as rational persons they already know that can exists an higher penalty

if both testify against each other than both remaining silent. To better view the problem, the prisoner's strategies as well as the sentences are represented in Figure 3.1.





	 cooperate	 defect
 cooperate	1 year 1 year	0 years 3 years
 defect	3 years 0 years	2 years 2 years

Figure 3.1: The Prisoner's Dilemma [youtube.com/ThisPlaceChannel, 2015]

As much rational as the prisoners are, by analyzing the problem they will choose to betray and testify against each other because they don't know what is the other's choice. They can just see what is best for themselves and by choosing to betray the other they will have always less penalty. This problem has a strict dominant strategy that is to betray and testify against each other but it can be seen that they can achieve just one year of penalty if they had cooperated. It must be recalled that this game is just played once.

3.1.2 Non-Cooperative Game Theory and Nash Equilibrium Background

In a non-cooperative game, the players make decisions independently based on the best payoff outcome for themselves. They do not communicate with other players in order to cooperate. The Prisoner's Dilemma already seen is an example of a non-cooperative game.

Many other examples of non-cooperative games can be seen in real world. The assigning property rights games like the rock-paper-scissors is a good example of them. These non-cooperative games are analyzed in game theory in order to understand or predict the decisions of the players. This can benefit the player decision because given what the other players have decided he can choose the best strategy which gives the best payoff to him given that specific scenario.

The Nash Equilibrium (NE) is the most fundamental concept in game theory used to analyze non-cooperative games. Considering a n-player strategic game, let $u_i(s_1, \dots, s_N)$ denote the payoff utility of Player i that is based on its strategy s_i and the strategy chosen by the other players, (s_1, \dots, s_N) . The Player i has a set of strategies and must choose one of them, $s_i \in \{S_0, S_1, S_2, \dots, S_j\}$.

A strategy s_i^* is the best strategy for Player i based on the others players strategies $(s_1, \dots, s_{(i-1)}, s_{(i+1)}, \dots, s_N)$ if the utility function is the best comparing with the other possible strategies of Player i , $u_i(s_1, \dots, s_{(i-1)}, s_i^*, s_{(i+1)}, \dots, s_N) \geq u_i(s_1, \dots, s_{(i-1)}, s_i, s_{(i+1)}, \dots, s_N)$. Given the others players strategies, Player i will choose always the best strategy which gives the best payoff for him.

The strategy set $(s_1^*, \dots, s_i^*, \dots, s_N^*)$ is a Nash Equilibrium when all players have chosen the best

strategy for themselves based on the others players strategies and have no incentive to change their strategy given what the other players are doing. So it means that the best individual payoff to all players was found.

3.1.3 Cooperative Game Theory and Nash Bargaining Solution Background

Cooperative games are built on top of non-cooperative games by rewriting the communication between the players. In this approach the players share the strategies and the player's attributes between them.

The bargaining problem is a problem of understanding how two players should cooperate when non-cooperation leads to inefficient results. Formally, bargaining problems represent situations in which multiple players with specific objectives search for a mutually agreed outcome (agreement) in order to achieve better results, although disagreements can occur.

The Nash Bargaining game is a game where two players demand a portion of some good (e.g. memory space). If the total amount requested by the players is greater than the available, each player receives nothing. If the total request is less than the available then they receive what they had demanded. These players can reject and present counteroffer when they are negotiating between them.

The Nash Bargaining Solution (NBS) was proposed by John Nash [16]. He proved that the solutions satisfying four proposed axioms are exactly the points (v_1, v_2) which maximize the function $(v_1 - d_1) \times (v_2 - d_2)$, where v_1 and v_2 represent the utility functions of Player 1 and 2, respectively. These utility functions must be greater than d_1 and d_2 , respectively, which represents the minimum acceptable amount demanded of each player from the negotiations.

3.2 State of the Art: Scheduling based on Game Theory

Temperature, energy, and performance are now at the heart of issues pertaining to sustainable computing. In multicore systems, high temperature in cores near each other can produce an "HotSpot" on the processor, which can cause instability of the processor and even hardware damage. In [4] a Generalized Tit-For-Tat Cooperative Game scheduling approach was proposed, based on a cooperative game theory concept, capable of minimize power density of the processor by achieving a more uniform thermal status on it. On a MP-SoC, [17] has proposed an approach based on Game Theory using Nash Equilibrium to adjust the frequency of each Processing Element (PE) at runtime, aiming to avoid the hot-spots and control the temperature of each PEs while maintaining the synchronization between the tasks of an application.

In a large-scale computer system, such as a computational grid, energy consumption can outweigh the procurement costs. In [18] and [3] were proposed energy-aware static scheduling algorithms for distributed heterogeneous computational grids. Those algorithms were based in cooperative ([18],[3]) and non-cooperative game theory([3]) and both are capable to perform a task-to-core mapping, including the required voltage level to execute each task by a machine, such that the entire system as a whole can benefit in terms of energy consumption.

3.2.1 Problem Definition

In [4] the proposed cooperative game approach was based on Tit-For-Tat game concept, which is a type of strategy usually applied to the repeated Prisoner's Dilemma (section 3.1.1). The Tit-For-Tat Cooperative Game is a multi-round version of the Prisoner's Dilemma where the player responds in one round with the same action that its opponent had used in the last round to achieve better results by cooperating [19]. Nash Equilibrium was used in [17] and [3] for the non-cooperative approach and Nash Bargaining Solution was used in [18] and [3] for the cooperative approach. These game scheduling approaches were developed to solve different problems but they are very similar with each other.

Based on game theory, to solve the scheduling problem, an auction system can be developed as a bargaining game. Auctions are a good way to assign tasks to cores. In an auction, the auctioneer (scheduler) present tasks to the players (cores/machines of the computational grid) in each round. The player must set a strategy and bid on the task in order to acquire it. At the end of a round, the auctioneer receives all the bids from the players and assign the task to the winner. In Figure 3.2 is represented the scheduling environment used in [4], where the Task Agent is the scheduler (auctioneer) and the Core Agent is responsible to calculate the bid and forward it to the auctioneer.

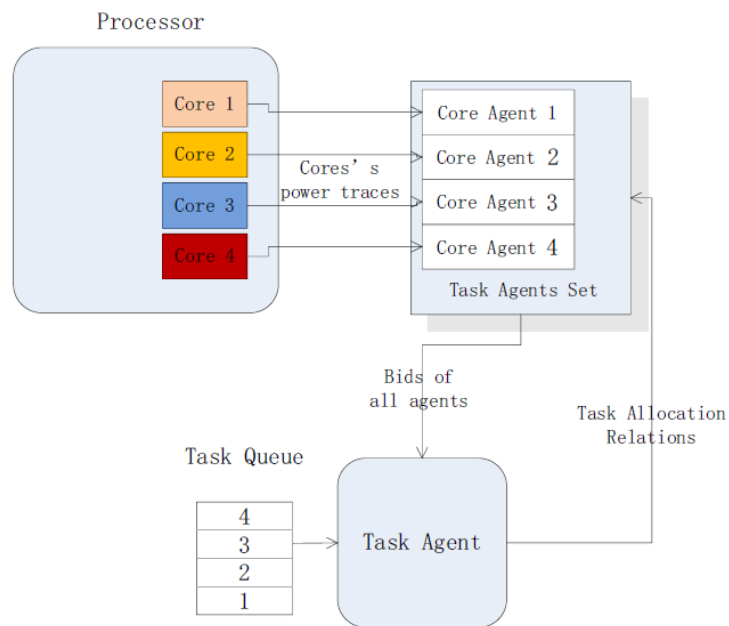


Figure 3.2: Auction game environment (GTFES [4]).

An auction n-player game should meet the concepts of game theory and is formally defined as:

- N Players. $P = \{p_1, p_2, \dots, p_N\}$;
- Each Player owns a set of strategies. $S = \{strategy_1, \dots, strategy_N\}$;
- In a round, each player p_i has a strategy S_i . The set of strategies in a round is $s = \{S_1, S_2, \dots, S_N\}$;
- Each player has a payoff function $u_i(\cdot)$ that can be based on other core's strategies and can be the same (or not) to every player.

- The game can be composed by multiple rounds, R rounds.

The most important parts of this game are how the players may choose their strategy and what must be the bid value. Regarding the temperature problem [4], cores (players) must choose as strategy to cooperate or not to achieve the social goal, which is to avoid the “Hot Spot”. This strategy is chosen according with the strategies of the other players in the previous round by definition of Tit-For-Tat game. However, as defined, Tit-For-Tat make senses only for a two-player game but it can be generalized to an extended n-player version where players can decide their strategy according to their observations from every other players in the previous round. The bid value is related with the strategy, if the core cooperates it must place an higher bid to avoid being selected to execute the task, which will reduce its temperature. In [17], the players (processing units) choose the best frequency as strategy. This strategy is also chosen according to the other processors frequencies on the previous round. In [3], the machines of the computational grid (players) can choose as strategy, to bid or not on tasks and the value of the bid is related with the energy consumption needed to execute the task on that machine. Naturally, the machine that bids the lower value will be the winner because is the one which executes the task with the minimum energy consumption.

3.2.2 Game theoretic approaches

The authors in [4] proposed a scheduling approach based on a cooperative multi-round version of the Prisoner’s Dilemma to reduce the processor’s temperature. An auction approach is also used on top of this game-theoretic approach, where the player’s bid is directly related with the core’s actual temperature status. The proposed algorithm in [4] is divided in two parts: the Task Agent’s Algorithm and the Core Agent’s Algorithm, which are represented in Figure 3.3 and 3.4, respectively. The authors had assumed the scheduling environment as resource-rich, which means that the number of tasks to be executed in the processor is always lower than the number of cores, and so, the number of cores needed to execute the tasks is at least the same as the number of existing tasks (T).

An affinity of the core is considered in the Task Agent’s Algorithm, as presented in Figure 3.3. The task preferentially choose its previous execution core if it is on the top $|T|$ winners of the auction and is not already occupied. If the core is not in the Top $|T|$ winners, the task will be assigned to the core which had placed the lower bid (the real winner, the Top 1).

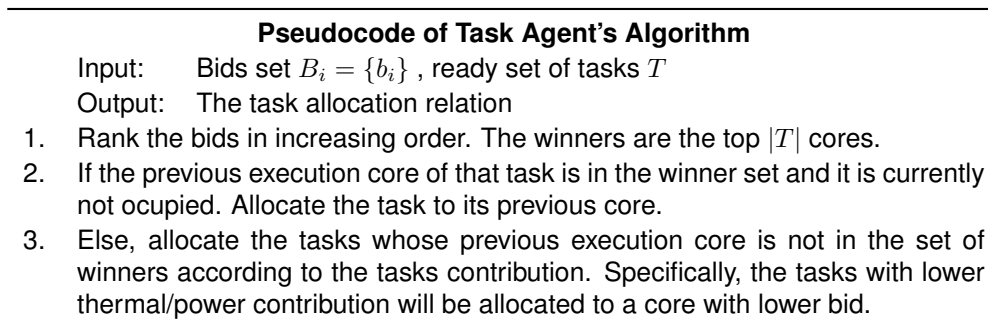


Figure 3.3: Task Agent’s Algorithm.

In the Cores Agent’s Algorithm (Figure 3.4), the core must first decide whether to cooperate or not.

This decision is based on the core's hardness factor (h_k), which can be considered the payoff function. This factor is the proportion of players who cooperate in the last round, calculated by each core. If h_k is higher than a predefined value (h_{th}^i) the player will decide to cooperate in the actual game round, otherwise, it will choose to not cooperate to achieve the global goal.

If the core choose to cooperate, the bid is based on the power status of that core (P_i) and a weighting coefficient (γ). Otherwise, the bid is based on the average execution time of the task in T (L_{avg}) and a weighting coefficient (ϕ). If the temperature of the core (S_i) is higher than a threshold (S_{th}) it will be forced to carry out a much higher bid in order to avoid being selected to execute the task. This will force the core to reduce its temperature.

To calculate the temperature of the core/processor, the authors used a temperature calculation method based on an Analytical Model of Temperature in Microprocessors (ATMI) [20]. This model needs the power consumption estimation of the core, which can be easily estimated in many processors because, as already seen, they have hardware performance counters for debugging and evaluating the performance as well as to measure power consumption.

Pseudocode of Core Agent's Algorithm

- Input: The core's hardness threshold h_{th}^i , temperature threshold S_{th} and the set of tasks T
- Output: The core's bid vector b_i .
1. Carry out the valuation of itself and calculate its thermal status S_i
 2. if $S_i > S_{th}$, the core is forced to cooperate in this round of auction. And, calculate its bid by its thermal status, that is, $b_i = \gamma \times P_i$. Then jump to step 6
 3. Calculate the hardness h_k in the previous round of the auction, which aims to decide whether to cooperate or not.
 4. if $h_k > h_{th}^i$, the core will choose to cooperate in this round of auction. And, calculate its bid by its power status, that is, $b_i = \gamma \times P_i$.
 5. if $h_k < h_{th}^i$, the core will choose to retaliate. And, calculate its bid by the average execution time of the tasks in T , that is, $b_i = \phi \times L_{avg}$.
 6. Send the bid to auctioneer (Task Agent).
-

Figure 3.4: Core Agent's in round K .

In [3], the auction approach was also implemented on two different proposed scheduling approaches. On one hand, the first scheduling approach uses a non-cooperative approach based on Nash Equilibrium, while on the other hand, the second proposed scheduling approach uses a cooperative game based on Nash Bargaining Solution. In both scheduling approaches, each task has a deadline and the entire workload is rejected if a deadline constraint is failed. The Earliest Deadline First (EDF) method was used to assign the tasks. In this method, tasks are sorted by its deadline value in ascending order, with tasks with lower deadline being the first to be scheduled. In each round of the game a single task is presented to the computational grid.

First, the machines choose their strategy, to bid the task with a value based on the necessary energy consumption to execute the task and be rewarded, or not to bid, resulting in a punishment by not participating in the auction. These outcomes enforce players to behave rationally and bid on tasks whenever possible. To properly compute the bid value, the machines (players) must know information about the

task (e.g. number of floating points operations, integer operations) and the specifications of the respective processor (i.e. DVFS interval, number of cores, etc). For this, a static Worst Case Execution Time (WCET) analysis can be performed in order to estimate the task execution time and its energy consumption.

Following the concept of auction, the auctioneer receives all the respective machine bids and will assign the task to the winner machine that has placed the lowest bid, meaning that the machine can execute that task with the lowest energy consumption among all.

In the non-cooperative approach the winning player in each round is forced to wait out of the game a constant number of rounds (F) until it can enter again in the game. This measure is used not to let the same player get all the tasks sequentially. The pseudocode of the non-cooperative approach algorithm is represented in Figure 3.5.

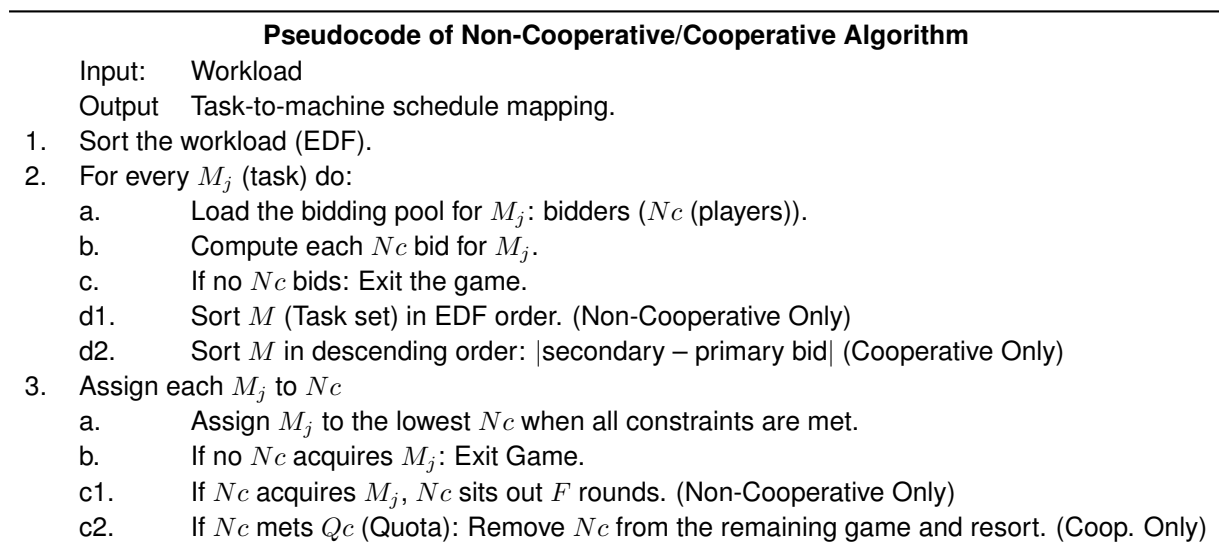


Figure 3.5: Pseudocode of the Non-Cooperative/Cooperative approach algorithm

For the cooperative approach there are two main changes in the pseudocode (on Step 2.d and 3.c) regarding the non-cooperative approach. In cooperative approach exists a quota mechanism where the players agree to acquire only a certain number of tasks (the quota constant) and a player must exit the game if it acquires that amount of tasks (Step 3.c).

In each round there are players who communicate (i.e. bid) and players who cannot communicate (i.e. not bid). The players who communicate in a round can bid on behalf of a player that cannot communicate as long as they do not deviate from that player's strategy and bidding power.

The order of assignment of tasks in this approach is different from the non-cooperative approach. It is based on a cooperative preference list where tasks are sorted in descending order of comparison between secondary and primary bids, $|\text{secondary bid} - \text{primary bid}|$ (Step 2.d). The secondary bid is the second lowest bid of all bids and the primary bid is the lowest of all. This preference lists sorts the tasks by the highest difference between bids, which means that exists an obvious winner just by looking for the difference between the bid's values. If the difference is near zero it means that the winner is not obvious and so those tasks will be the last to be assigned. If don't exists a secondary bid, which means

that, just one machine have bid on that task, this task will have an higher priority and its value to sort will be the highest bid plus one, which means that it will be one of the first tasks to get assigned.

Once all the tasks are assigned to the respective winning machines, the authors go further, regarding energy-savings, and use the DVFS technique to reduce the overall power consumption as presented on the pseudocode in Figure 3.6. The scheduler asks to the machines the lowest DVFS interval needed to execute the specific task on a core. Iterations are made in order to lower the DVFS interval on each task while checking if the respective deadlines constraints are respected. When the minimum power consumption mapping task-to-core is found the game is finished and the tasks are sent to the computational grid to be executed on the respective cores with the respective voltages.

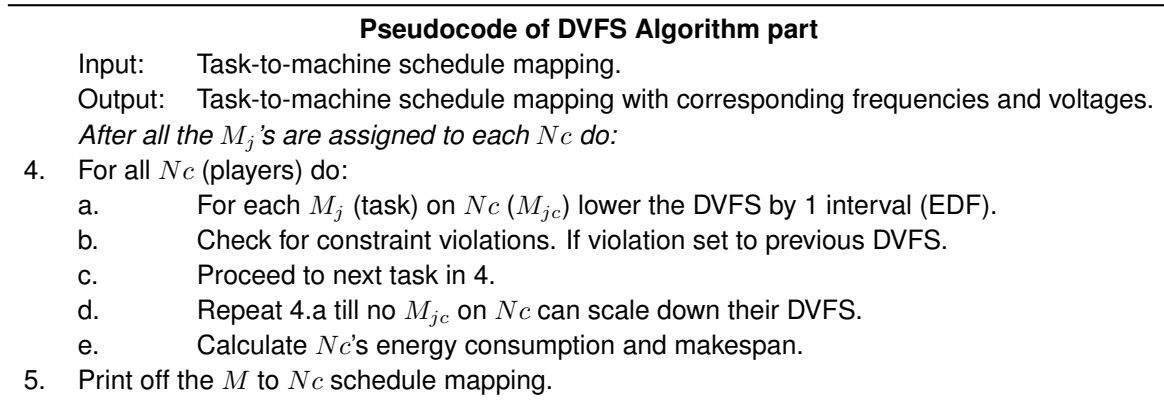


Figure 3.6: Pseudocode of DVFS Algorithm part.

In [18], the proposed algorithm was based on the concept of Nash Bargaining Solution (NBS) from cooperative game theory. The authors, through rigorous mathematical analysis proved that the proposed algorithm converge to the bargaining point. In similarity to [3], this approach also produces a task-to-machine scheduling map ensuring energy consumption and makespan optimization. The pseudocode of the algorithm is presented in Figure 3.7.

Each machine must decide as strategy, to select the lowest frequency possible to execute the task while fulfilling its deadline. The cooperation between machines is done through the centralized scheduler by using the average instantaneous power of all machines in the grid as one of the main decisions to schedule the tasks.

The scheduler starts to sort the tasks from the workload in decreasing order of their deadlines, scheduling first the tasks with higher deadlines. The machines are also sorted in decreasing order of their current instantaneous power. Then, the scheduler selects the machine that is just right above average instantaneous power of all machines in the system and with the necessary architectural requirements to execute the task (e.g. available memory space), leaving the high-powered machines for larger tasks and low-powered machines for smaller tasks. Although, if the selected machine is not capable of executing the task then the next high-powered machine will be chosen and so on until the task is scheduled. In each round, one task is scheduled, the values of each machine current instantaneous power are actualized and the machines are sorted again in decreasing order.

Pseudocode of NBS-EATA Algorithm

Input: Machines each initialized to its maximum instantaneous power using DVFS = $\{dv_{S1}, dv_{S2}, \dots, dv_{Sj}\}$

Output: A mapping that consumes minimum instantaneous power and has the minimum possible makespan.

0. Sort all tasks in decreasing order of their deadlines: $d_1 \geq d_2 \geq \dots \geq d_n$
1. For every task:
 2. Sort the machines in decreasing order of their current power consumption: $p_1 \geq p_2 \geq \dots \geq p_m$
 3. Compute the average power consumption of the system: $p_{av} = (\sum p_j)/m$
 4. Select the machine right above p_{av} . While $p_{av} \geq p_m$ do:
 - 4a. $m = m - 1$
 - 4b. $p_{av} = (p_{av} - (p_{m+1}/m + 1))(m + 1/m)$
 5. If machine meets the architecture requirements the goto Step 5a else goto Step 5c.
 - 5a. If found the smallest DVFS that satisfies the deadline of the task then goto Step 5b else goto Step 5c.
 - 5b. Assign the task t_i to machine m_m with the found DVFS, update p_m and goto Step2.
 - 5c. If it is not the last machine then $m = m - 1$ and goto Step4 else goto Step6.
6. Initialize all machines to maximum power and goto Step2.

Figure 3.7: NBS-EATA Algorithm.

In [17], to control the temperature of each homogeneous processing elements (player) and maintain the synchronization between each task running in the system, a non-cooperative approach based on Nash Equilibrium was developed. This approach, in similarity with the others works, is also focused in a utility function. However, it is based on a two optimization objective. The first part of the utility function takes into account a temperature model that is based on power consumption and thermal resistance of each PE in the system, while the second part takes into account a synchronization model based on the difference between two player's clock frequency.

The strategy of each player is to decide the best frequency to run the task, knowing that it can affect the synchronization with the others players and also their temperatures, which is decided by unilaterally finding the frequency that maximizes the player's utility function. This procedure is shown in the pseudocode presented in the Figure 3.8.

Pseudocode to find the best strategy for player i

Input: Utility function (Ui), Player's old strategy (MyStgy), Others players strategies (OthrStgy).

Output: Player's new strategy (NewStgy).

for all Player's strategy (Stgy)

if $U_i(\text{Stgy}, \text{OtherStgy}) > U_i(\text{NewStgy}, \text{OtherStgy})$

$\text{NewStgy} \leftarrow \text{Stgy}$

otherwise

$\text{NewStgy} \leftarrow \text{MyStgy}$

end

end

Figure 3.8: Unilaterally maximization function.

In order to arrive to a Nash Equilibrium, an iterative process was developed where in each round every player choose its best frequency. At the end of the round the system broadcasts the strategies taken between the players in order for them to see in the next round if they want to change the decision or not. The iterative process proceeds until there are no changes in the players decisions between two consecutive rounds. However, the authors have seen that the proposed approach did not converge to a solution in 6% of the evaluated scenarios. The system iterative process is presented in Figure 3.9.

Pseudocode of a game cycle

```

for each round of the game
  for each player  $i$ 
    NewStgy[ $i$ ]  $\leftarrow$  UnilaterallyMax( MyStgy[ $i$ ])
  end
  MyStgy vector  $\leftarrow$  NewStgy vector
end

```

Figure 3.9: Kernel iterative process.

As seen in this chapter, the auction approach was used in several works due to its simplicity, low complexity and revealed to be an interesting method to schedule the tasks iteratively, but the main focus must reside on how the bid is computed, i.e., how the player's utility function must be developed and directly related with the necessary energy consumption to execute a specific task on a specific core. It should also be noted that some presented works were focused on static scheduling approach, however this thesis will focus on creating a new dynamic scheduling approach.

3.3 Summary

This chapter introduced the main branches of game theory, the non-cooperative and cooperative game theory, as well as the most used concepts in each of them, the Nash Equilibrium and Nash Bargaining Solution, respectively. It was also presented the scheduling approaches based on game theory, in which some of them have low complexity and very intuitively to use.

The works [3] and [18] addressed the scheduling of tasks with deadline constraints, which is not common on user's applications. Both works proposed a static scheduling approach that generate task-to-machine maps with the respective executions frequencies for each task. However, this thesis will focus on developing a dynamic run-time scheduler approach.

The approach proposed in [3] focus more on the necessary energy consumption of the task to execute in a specific machine to decide in which machine the task should be scheduled. It uses an auction based approach, which is more intuitively compared to a game, where players bid the tasks against each other to obtain and execute the task. However, the use of the DVFS technique to reduce the power consumption could lead to higher energy consumption in the system, just because of the fact that lowering the frequency will increase the execution time, and thus, the product between a lower power consumption and an higher execution time can be sometimes higher than the opposite scenario. This thesis is focused on having lower energy consumption, what does not necessarily means lower power

consumption, and so, this should be taken into account.

The approach in [18] focus more on the actual instantaneous power consumption of the machines and DVFS as the principal decision points. It is used an iterative process to schedule the tasks sorted by their deadlines to the machine that is just right above the average instantaneous power of all machines in the system.

In [4], an approach based on a repeated Prisoner's Dilemma game and auction approach has been proposed. Based on previous rounds values and defined thresholds, the cores can decide if they must avoid being selected to run a task or not, leading to reduce its temperature and avoid "hot-spots" in the processor's chip. Although being an energy-aware approach, it is mainly focused in temperature and not energy consumption at all.

Relatively to [17], the definition of an utility function, based on the contributions of all players in the system, to be used in order for the player to decide the best strategy, is a good approach. However, the iterative process to select the player's decisions could not converge to the solution as the authors have seen in 6% of tested scenarios.

4

Framework

Contents

4.1 Framework general overview	34
4.1.1 Auction based approach	35
4.1.2 Game theoretic approach	36
4.2 Framework implementation in ARM Juno r2 board	43
4.3 Time and Power prediction for several frequencies	45
4.3.1 Task execution time	46
4.3.2 Task instantaneous power consumption	47
4.4 Summary	49

In this chapter, the proposed framework is presented. Firstly, a general overview of the framework will be made, where the concepts behind the proposed auction and game theoretic approaches are explained. As seen in section 2.3, these approaches must be developed in a way that integrate the separated decision subsystems with the scheduler, in order to provide energy-aware decisions, and improve the energy-efficiency of the whole system. Afterwards, it will be presented the necessary modifications on the proposed framework to be implemented in the ARM Juno r2 board. Finally, a detailed description of how the performance measures are taken as well as how the cores of the processor, as players, compute their bids, will be made. It should be noted that, this work is focused in saving energy consumption in mobile devices (e.g. smartphones), and so, it must be focused on saving energy in the whole system and not just the processors of that system.

4.1 Framework general overview

The proposed framework consists in combining an auction approach and a game theoretic approach. On one hand, the auction approach will assure a base game structure in which the players can compete with each other by bidding the tasks in order to acquire them. Basically, when a new task arrives to the scheduling queue, the scheduler must present that task to all existing players, and then is up to each player to decide its own strategy, bid the task and send its value back to the scheduler. On the other hand, it is necessary to minimize the global energy consumption in a system composed by many players. To accomplish that, Nash Equilibrium, the most fundamental concept in non-cooperative game theory, will be used. Each player will use this concept to decide its best strategy and to compute the bid. The general framework structure is shown in Figure 4.1.

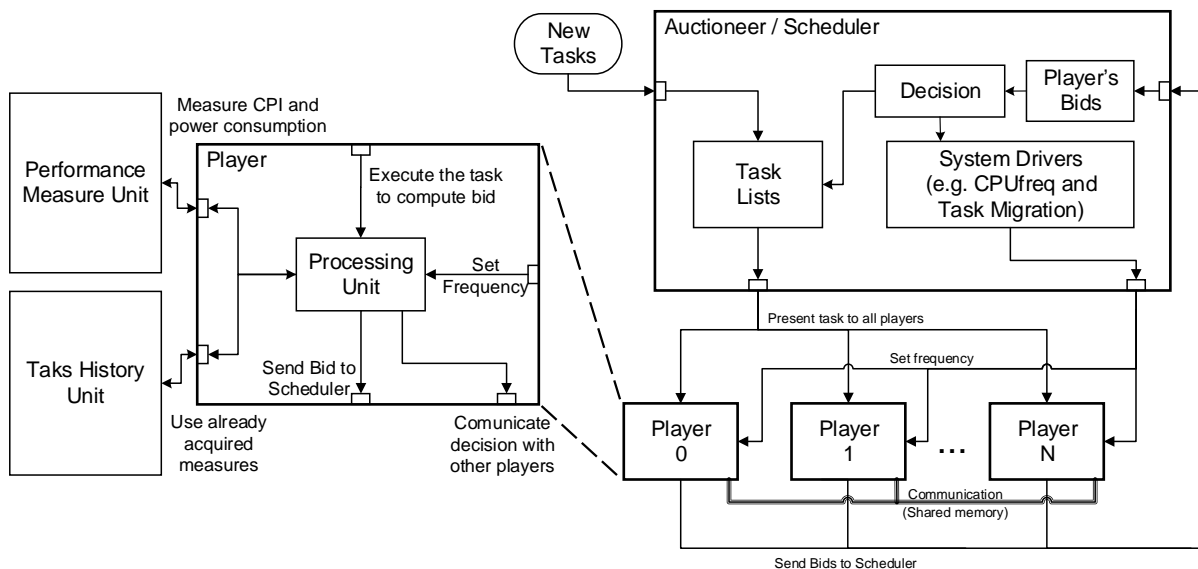


Figure 4.1: General structure of the proposed framework.

In order to minimize the overall energy consumption of the system, each player's decision/bid must be related with the necessary energy consumption to execute the task. Knowing that each task may have different characteristics, each player must use the performance measure unit to characterize the

tasks through the performance event counters and to get instantaneous power consumption measures from the energy meters registers to have energy-awareness. Other relevant information about the task, acquired from previous measures, can also be stored or accessed through the task history unit.

Taking into account the energy-awareness of the proposed scheduler and the game theoretic approaches, sub-optimal solutions can be found by individually selecting the best frequency to execute the task in each core and then globally selecting the best core to execute that task.

4.1.1 Auction based approach

The auction approach, as seen in section 3.2.2, was used in some researchers' works [3][4]. This approach assures that players compete with each other by bidding the tasks in order to acquire them. In each round of the game, one auction is realized, which means that one task is scheduled per round. The auction of each task gives to each player an opportunity to participate and be able to execute the task. In Figure 4.2 is shown the flow diagram of the auction approach implemented by the scheduler. It should be noted that the players' bidding process shown in the right side of the figure can be controlled by the scheduler in a serialized or parallel way. It will be the architecture of the system that will fix the way how the bidding process should be implemented. Once the player receives the task, it will execute the task for a short amount of time in order to acquire the necessary performance and energy measures. The player should also have access to the others players information in order to decide its own best strategy using the game theoretic approach. Then, when all players have computed and sent their bids to the scheduler, the winner core will be selected, which will receive and execute the task.

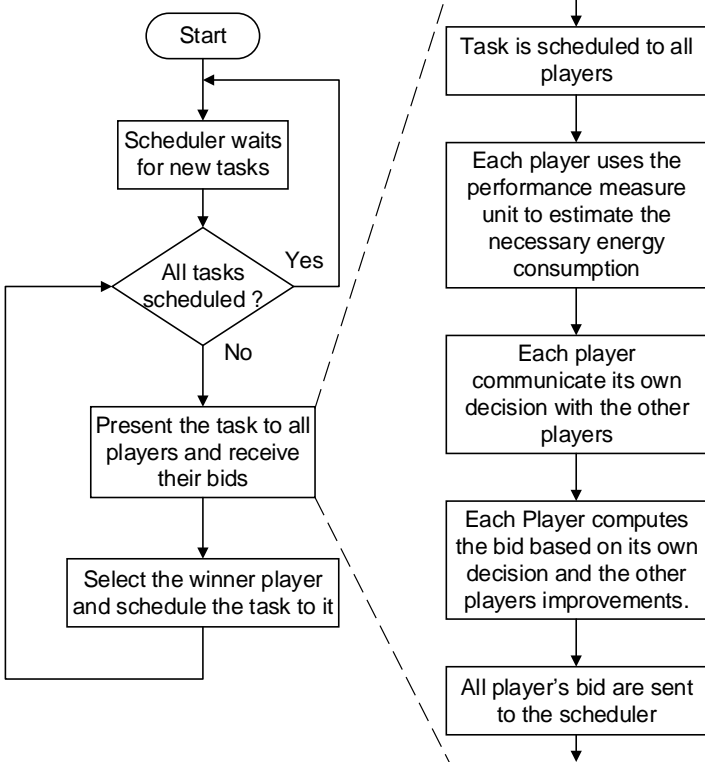


Figure 4.2: Fluxogram of the auction approach and bidding process.

4.1.2 Game theoretic approach

The Nash Equilibrium concept will be used in this game theoretic approach. As seen in section 3.1.2 and in some works [3][4][17], the utility function is the principal mechanism used by the players to select their own strategy. Using the auction approach together with Nash Equilibrium, it is expected that the players place a bid, for the task to execute, which is based on the decision that maximizes the player's utility function, and also, on the principal goal, to minimize the overall energy consumption of the system. Having said that, it is expected that the player utility function is directly related with the overall energy consumption of the device, and so, in each auction, the winner will be the player who bids the lowest value, which means that it can execute the task with the lowest energy consumption among all players. By scheduling one task in each auction to the core that can execute it with the lowest possible variation in the overall system's energy consumption, this scheduling approach is able to find a local sub-optimal solution.

As already mentioned in this chapter, the proposed approach must consider the energy consumption of the whole system and not just of the processors. This must be present because this work is focused on mobile devices, such as smartphones, which have a lot of hardware that could have higher energy consumption than just the processors. Mobile devices are energy limited devices, and thus, the main goal herein is to save energy of their battery as much as possible. On a compute system exists mainly two subsystems that consume energy, the processing units (e.g cores of a CPU and GPU) and the remaining units (e.g. memory, system drivers and peripherals). Therefore, the overall player utility function (equation 4.1) must be divided in 2 parts: the selected core (u_{player}) utility function, which is divided in two sections to cover the increase of energy consumption in the selected player and its impact on other players ($u_{individual}$), as well as the impact on the remaining units (u_{global}); and the other cores ($u_{other_players}$) utility function, which aims to find if it is possible to change their strategy in order to improve energy savings having into account the strategy of the selected core. All these functions are shown in equations 4.2 - 4.5 and will be explained in the next sections.

In these equations, S_{ij} is the strategy set of player i , and j is one of the M_i available strategies, $S_{ij} = \{s_{i1}, s_{i2}, \dots, s_{iM_i}\}$. In this game approach exists N players, $i \in \{1, 2, \dots, N\}$, and $s_{i'}$ represents the best strategy selected by player i in the last round/auction; s_{i^*} represents the best strategy of player i , which is found through the selected player utility function (Equation 4.2).

$$u_i = u_{player}(s_1', s_2', \dots, S_{ij}, \dots, s_{N'}) + u_{other_players}(S_{1j}, \dots, S_{(i-1)j}, S_{(i+1)j}, \dots, S_{Nj}) \quad (4.1)$$

$$u_{player}(s_1', s_2', \dots, S_{ij}, \dots, s_{N'}) = \min_j [(u_{individual}(s_{ij}) + u_{global}(s_1', s_2', \dots, s_{ij}, \dots, s_{N'}))] \quad (4.2)$$

$$u_{individual}(s_{ij}) = \Delta Energy_{player}(s_{ij}, s_{i'}) + \sum_{w=1, w \neq i}^N \Delta Energy_{player}(s_w') \quad (4.3)$$

$$u_{global}(s_1', s_2', \dots, s_{ij}, \dots, s_{N'}) = \Delta Energy_{system}(s_1', s_2', \dots, s_{ij}, \dots, s_{N'}) \quad (4.4)$$

$$u_{other_players}(S_{1j}, \dots, S_{(i-1)j}, S_{(i+1)j}, \dots, S_{Nj}) = \sum_{w, w \neq i}^N \min_j (\Delta Energy_{other_player}(s_{i^*}, s_{wj}, s_{w'})) \quad (4.5)$$

Individual utility function

Equation 4.2 represents the utility function of the player i that will compute the bid for the received task. In this case, the set of actions or strategies of player i , S_{ij} , will be the set of available frequencies in the DVFS subsystem, $S_{ij} = \{s_{i1}, s_{i2}, \dots, s_{iM_i}\}$, being s_{i1} the lowest frequency available. This individual player utility function is focused on unilaterally minimizing the variation of energy consumption on player i , when it is selected to execute the task, by selecting the best frequency to achieve that.

Energy consumption is the product between execution time and power consumption, so, it is necessary to compute an estimation of the tasks execution times and the player (core) current power consumption. In this work, it is assumed that is already known or has been collected some information about the task a priori, mainly the total number of instructions of the task, in order to estimate the task execution time. By measuring the task performance through the performance counters is possible to know its current Cycles Per Instruction (CPI) ratio. Knowing the current CPI ratio, frequency and the total number instruction it is possible to estimate the task execution time, as it will be further explained in section 4.3.1. To obtain the current instantaneous power consumption value, it will be necessary to read the APB energy meters registers of each processing unit, which will also be further explained in section 2.2.

In systems with many tasks, executing at the same time, it is necessary to know in which cores are the tasks being executed, as well as, their actual frequencies and power consumptions. In this proposed approach, all this information is put together and stored in memory. To simplify the reading, this structure will be referenced as "task execution map". In Figure 4.3 is shown an example of a task execution map in order to describe how the selected player chooses the best frequency to execute the received task.

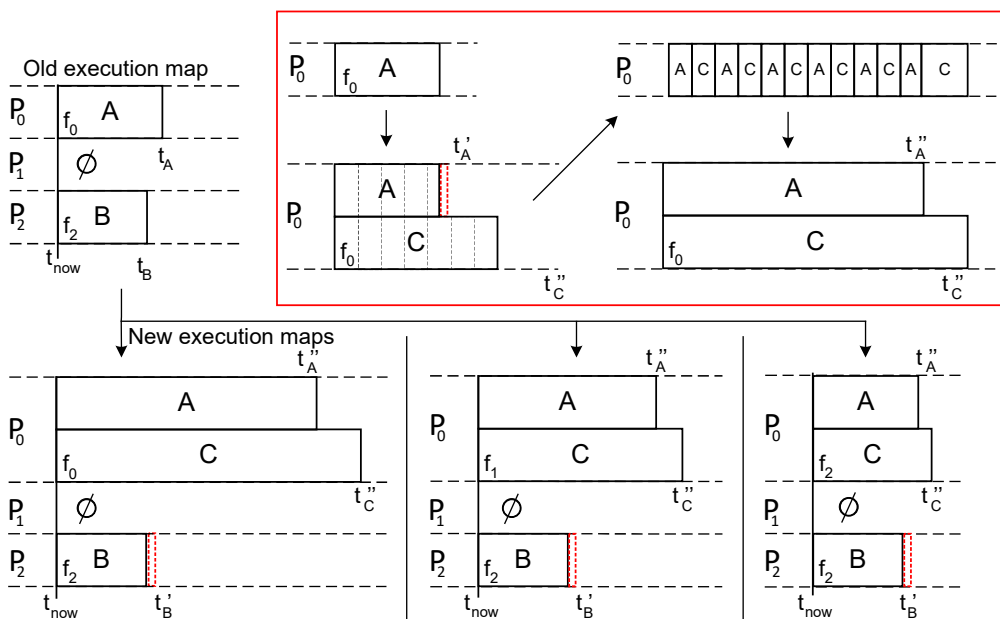


Figure 4.3: Individual utility function, usage example.

In this example, player (core) 0, P_0 , and player 2, P_2 , are already executing task A and B, which will finish approximately at t_A and t_B , respectively. P_1 is empty, and thus, in idle mode. However, when task C is scheduled to an occupied or empty core, the execution time of existing tasks A and B, will increase to $t_{A'}$ and $t_{B'}$, respectively, due to conflicts between the tasks inside the shared caches and memory, i.e. cache-misses that originate data fetch from memory to the cache, which overwrites the data used by the other task. Besides that, if task C is scheduled to the same core that is executing task A then the execution time of task A will increase from $t_{A'}$ to $t_{A''}$ due to the round-robin time-sharing policy, SCHED_OTHER, as already seen in 2.4.1. This increase is shown in the red square present on Figure 4.3. This is a good example to show that an energy-aware approach must be aware of the availability of each core before scheduling the task, because when task C is scheduled to the same core that is executing task A, the amount of context switching between tasks is quite intensive due to the SCHED_OTHER policy, and thus, it will probably be less energy-efficient than if task C was scheduled to an empty core, which in this case is P_1 .

Following the example, to properly compute the necessary energy consumption to execute task C on core 0, it must be first introduced the following $\Delta Energy$ and the $Energy_{map}$ functions:

$$\begin{aligned} \Delta Energy &= Energy_{new_map} - Energy_{old_map} \\ Energy_{map} &= \sum_{i=1}^K Power_{Task_combination(i)} \times \Delta Time_{Task_combination(i)} \end{aligned} \quad (4.6)$$

In Equation 4.6, K represents the number of task combinations in the execution map of the selected player. As seen in Figure 4.3, K is 1 when core 0 is just executing task A (old execution map), and is 2 when core 0 is executing task A and C (new execution map) because there are two time slices with different combinations, first, the time slice from t_{now} to $t_{A'}$ with the task combination A and C and then from $t_{A''}$ to $t_{C''}$ with just task C.

Following the example, the energy consumption of the new execution map of core 0 is $Energy_{new_map} = Power_{AC} * (t_{A'} - t_{now}) + Power_C * (t_{C''} - t_{A''})$, while previous, the energy consumption of old execution map was just the energy consumption of task A solo, $Energy_{old_map} = Power_A * (t_A - t_{now})$. In this utility function, it is also taking into account the increase of energy consumption in the other players due to the impact of core 0 decisions, and so, these energy consumption variations must be also added in the individual utility function, as seen in the second term of Equation 4.3. It should be noted that each player must have the ability to access and read its own instantaneous power consumption value in order to compute the $\Delta Energy$.

In these calculations, it can be seen that for each task execution map it must be known several task execution times and powers consumption values. In this example, $Power_{AC}$ can be measured immediately after task C is scheduled to core 0, while $Power_A$ was already been measured when task A was scheduled in the previous round. However, $Power_C$ is not known but can be assumed that its value is already known from previous executions of task C and is stored on the task history unit or it can be measured by pausing task A in order to measure the instantaneous power consumption of task C solo. $Power_C$ can also be measured when task C is being executed in its own time slice on

the SCHED.OTHER policy. The execution times can be approximately estimated through the CPI, frequency and total number of instructions as will be explained in section 4.3.1. The new execution maps should be stored on the memory in order to be used in the next auctions and also because they will be used in the global utility function.

With these functions is possible to compute the variation of energy consumption of the core when a new task is scheduled on it. Although, the energy consumption depends on the core's frequency, there will be as many execution maps as frequency levels available on the core. The new frequency (new map) that assures the lowest variation of energy consumption compared to the old frequency (old map), will be the individual decision of the player.

Global utility function

The energy consumption of the mobile device is not only due to the processor, as previously discussed, but also other components consume energy consumption. For simplicity, this set of components will be referenced as "system", which refers to the existing hardware of the ARM Juno r2 SoC outside the processors (clusters), mainly the DRAM memory, buses, power-management subsystems (e.g. DVFS) and other peripherals. On some devices this energy consumption can be even higher than the processor, and so, it must be considered to the scheduling decisions.

It would be interesting to expand the meaning of "system" to all components in the mobile device, as for example the device's display or sound speakers, and to establish a connection between the task, for example "play music", and its respective power consumption in the processor as well as in the audio speakers. This should give full energy-awareness to the scheduler, but unfortunately, is far from being done because it would imply the manufacturers to integrate more power sensors hardware in their components and also to establish compatibility with the existing SoC.

In ARM Juno r2 board and generally in mobile devices, the variation of power consumption in the "system" is mainly due to the DRAM memory accesses, because it is one of the components whose usage is more dependent on the executing tasks. However, this variation can be slightly insignificant when compared to the static power consumption that the remaining components in the "system" have.

The global utility function operates similarly to the individual utility function. The difference is that all the players must now be seen as an unique player. This modification is required because the overall instantaneous power consumption of the "system" is only represented by one power consumption sensor. And so, the task combinations must be aggregated at the level of the whole system and not at the level of the player. In Figure 4.4 is shown an example to better understand the global utility function. As mentioned before, for each player, there exists as many execution maps as available frequencies. However, for simplify, in this example, it is just shown two new task execution maps when the received task C is scheduled to core 1 and core 2 at frequency 0 and 2, respectively. The $\Delta Energy$ and the $Energy_{map}$ functions used in this utility function are the same that are used in the individual utility function, Equation 4.6. However, in this utility function, the power considered will be the "system" instantaneous power consumption and not the power consumption of each individual player.

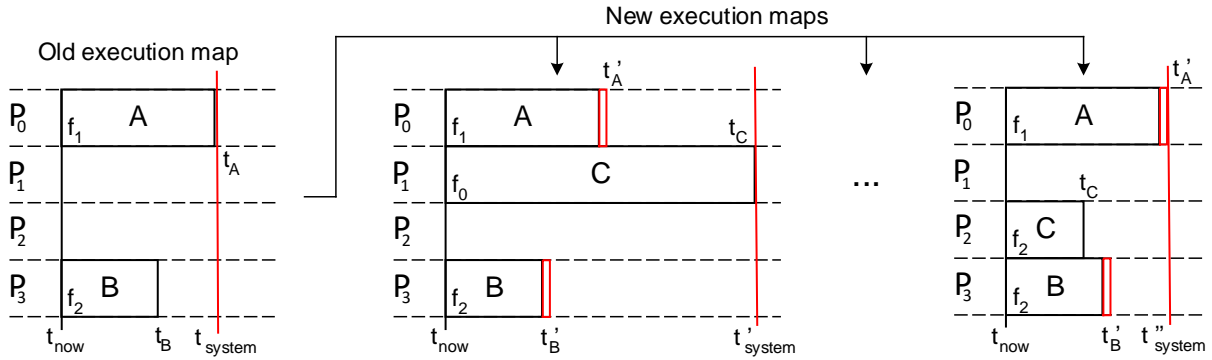


Figure 4.4: Global utility function, usage example.

Following the example, the "system" energy consumption of the new execution map when the task C is scheduled to core 1 with the lowest frequency is $Energy_{new_map} = Power_{ABC} * (t_{B'} - t_{now}) + Power_{AC} * (t_{A'} - t_{B'}) + Power_C * (t_C - t_{A'})$, while previously, the energy consumption of old execution map was, $Energy_{old_map} = Power_{AB} * (t_B - t_{now}) + Power_A * (t_A - t_B)$. In this example, $Power_{ABC}$ is measured immediately after task C is scheduled to core 1. $Power_{AC}$ is not known but can be measured by pausing the other tasks for a short amount of time. $Power_C$ is also not known but it can be used the same approach used for $Power_{AC}$ or assume that the value is already known, it was stored in the task history unit. To overcome this problem in future auctions, the unknown instantaneous powers consumptions must be stored in the task history unit. However, as already discussed, the "system" power consumption can be approximately constant, and this would let the $Power_{AC}$ be approximately equal to $Power_{ABC}$ and not needed to be measured. However, this would be a pessimistic approach and the respective associated errors could influence the overall decision. It should be also noted that if task C was scheduled to core 2 at frequency 2, the only unknown task combination power consumption is the new ABC, because AB and A were already been measured on previous auctions.

As seen in Figure 4.3 and 4.4, different frequencies lead to different execution maps, and hence, different energy consumptions. The relevant information of each execution map should be stored on the memory in order to be used in the next auctions. In this example, it should be noted that depending on the frequency and core selected to schedule the task, the t_{system} varies, and so, in some execution maps, the energy consumption of the "system" can be more relevant than in other execution maps.

Joining the individual utility function with the global (system) utility function it is possible to predict the increase of energy consumption inside the processor and in the "system" when scheduling a task to a specific core. For each frequency available on the core, the one which implies lower variation of the overall (core + system) energy consumption will be selected as the best decision for that player. However, it must be also seen if the remaining players should change (or not) its previous decisions based on the actual selected player decision, which is taken into account in the Other players utility function that will be presented in the following section.

Other players utility function

As seen in the last two utility functions, the selected core choses the best frequency based on the variation of energy consumption on the processor as well as on the "system". This is done because the

scheduler should have an overall energy-awareness and not only at the level of the processor. However, the approximately constant power consumption of the "system" can be so higher when compared with the processor/core, that this last can be irrelevant when compared to the "system", which can lead to select higher frequencies on the core in order to reduce the task execution time and hence reduce the dominant "system" energy consumption. Having said that, once we have taken into consideration this overall energy consumption of the device, we can then look if it is possible to save energy consumption in the remaining cores by selecting their best frequencies from the individual utility function, as illustrated in Figure 4.5.

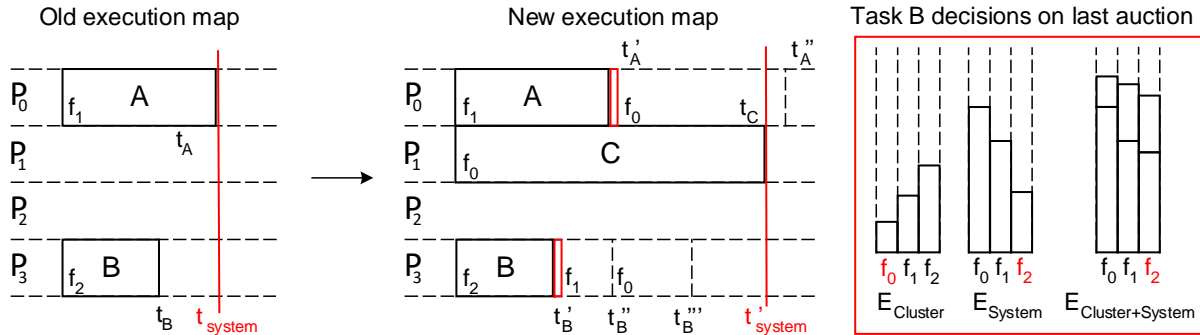


Figure 4.5: Other players utility function, usage example.

As already seen, in the selected core utility function (u_{player}) the best frequency for that player is chosen and the corresponding execution map is stored on memory. This new execution map will be the base for this utility function. The highest execution time of all players, is designated the time system, t_{system} , which is marked red on the example. The goal of this approach is to see if the other players can lower their frequency in order to achieve energy savings while they are just allowed to select frequencies whose execution time is lower than the t_{system} .

Following the example, task B was scheduled in the last auction to the core 3 with the highest frequency (see $E_{Cluster+System}$), which was not the one that corresponds to best individual energy savings (see $E_{Cluster}$) as can be seen in the energy values represented in the red square (Figure 4.5). It should be noted that when each task is scheduled to a core, its individually values of necessary energy consumption to execute on that core for all available frequencies are stored in memory in order to be used in this utility function. For task B, the best individually energy consumption would be achieved when selecting the lowest frequency. However, f_0 was not selected in the last auction because the "system" has higher energy consumption at that frequency, and thus, the selected frequency was f_2 , which provides lower overall energy consumption in the device. By using the selected core utility function (u_{player}), the best frequency selected to schedule task C to core 1 was the lowest one, which corresponds to a change from t_{system} to t'_{system} . Looking to each execution time for each frequency, one can see that the core 3 can achieve energy savings by selecting the frequency 0, which corresponds to an execution time t'''_B . However, for task A, frequency 0 will not be selected because the task execution time will be higher than the t'_{system} .

By using these three utility function, one can conclude that, in each auction, the new task can be scheduled to the core that offers the lowest variation of energy consumption on the device. With this

game approach, each player will place a bid regarding its own decisions and the other players decisions, which will lead to find local sub-optimal energy savings. In Figure 4.6 is shown the pseudocode of the player algorithm based on the proposed game theoretic approach.

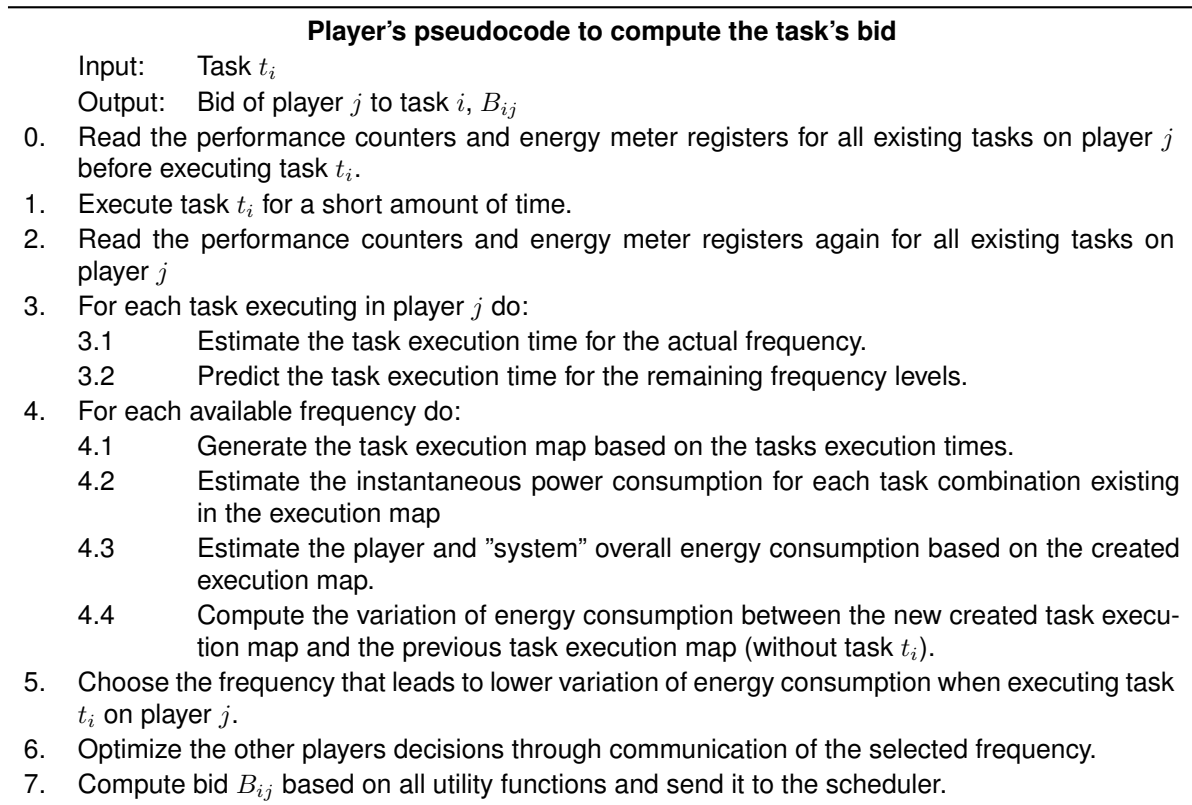


Figure 4.6: Player's pseudocode to compute the task's bid.

As can be seen in the player algorithm (Steps 0-2), the player starts by receiving the task and characterize it through the performance counters, which will be explained in section 2.2. The player afterwards, uses this information to estimate the tasks execution times and instantaneous power consumptions (Steps 3-4). However, this estimation is done just for the current operating frequency and not to all the available frequencies, which would turn this scheduling approach to be similar to exhaustive search and to have higher energy consumption due to excessive utilization of the DVFS drivers. To overcome this problem, we can predict the instantaneous power consumption and execution time values without changing the frequency, as it will be explained in section 4.3. Returning to the algorithm, the player then selects its best strategy that contributes with the lowest variation of energy consumption to the overall device, and communicate it to the other players. These will see if it is possible to improve their strategies to achieve energy savings, and communicate back to the selected player (Step 5-6). Finally, based on all these energy consumption values, the selected player computes the final bid and sends it to the scheduler (Step 7). Once all players have sent their bids to the scheduler, the player with the lowest bid will be the winner and the task will be schedule to it as well as the respective frequency changes.

4.2 Framework implementation in ARM Juno r2 board

As already seen in section 2.1, the compute subsystem of ARM Juno r2 board is composed mainly by a dual-core Cortex-A72 cluster, a quad-core Cortex-A53 and a quad-core Mali-T624 GPU cluster. This board only has energy meters at the level of the cluster and not at the level of the individual core, therefore, the players in this game approach will be the clusters and not the individual cores, because it can be just known the sum of instantaneous power consumption of all cores on the cluster, and not on each individually one. However, one of the clusters, the GPU, was not been used because the Mali Drivers and OpenGL ES (OpenGL for Embedded Systems) are not supported in the current Linaro OpenEmbedded filesystem (see section 5.3). And so, the players presented in this approach are: the big cluster composed by two core and the LITTLE cluster composed by four cores. In this implementation, it will be adopted the notion of "players' representative", which are the clusters and the "sub-players", which are the cores. Basically, the idea is that the player represents a coalition of sub-players. This player must adopt an non-cooperative game approach in its relation with the other clusters, but, at the same time, it must exists some cooperation between the cores inside that cluster, because when the frequency is changed in the cluster it will change the frequency of all cores of that cluster. This limitation will lead to a modification in the $u_{individual}$ utility function in the way of how the task execution map is used.

To better describe this modification, it will be shown in Figure 4.7 an example of how the execution map will be used to be computed the energy consumption. Following the example, it can be seen that before the modification it exists 6 cores, which corresponds to 6 players, and after the modification it exists 2 players, one that has 2 cores and the other that has 4 cores. Before the modification, one can see that only the selected core 3 changes its frequency to compute all the possible task execution maps, while in the other cores just have to be considered the increase of energy consumption due to the conflicts with the new task E. After the modification, all possible frequency changes in the selected core 3 of the LITTLE cluster will change the whole execution map of that cluster. In this approach it is assured cooperation between the sub-players by selecting the new best frequency for all cores that corresponds to the lowest energy consumption on the cluster, and then it is also assured a non-cooperative approach between the players, by using the auction approach in order to compete individually against each other.

The tasks to use in this approach are real benchmark applications, as it will be explained in section 5.2. These tasks can have different execution phases until they finish. For example, they can start to behave like a memory bounded task, which means that they are mainly dependent on the memory's frequency and not on the processor's frequency, because they do many memory accesses and they must wait many cycles to obtain the stored data. And then, they can behave like compute bound tasks, where they just use the caches of the processor, which are much faster than the general DRAM memory. This unpredictable behavior, i.e. different phases, can affect the estimation of the task execution time, as it will be explained in section 4.3.1. To overcome this problem, the scheduler must do the estimations frequently in order to detect the different task phases. In this approach, when a new task needs to be scheduled, it is done a new estimation of all tasks execution times in order to compute the energy

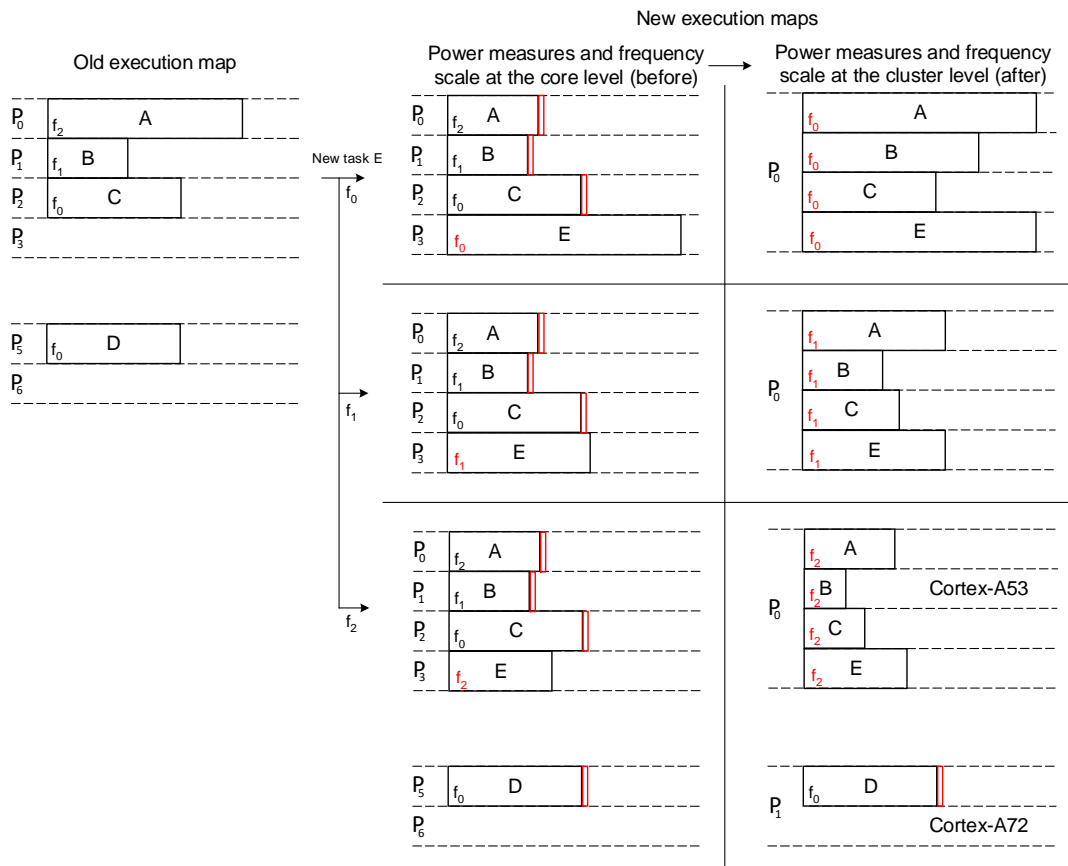


Figure 4.7: Individual utility function adaptation, usage example.

consumption more accurately. It is also done a reschedule of the already executing tasks when one of the tasks has finished. This contributes to actualize the estimations of executions times as well as to give other opportunity for other cores to acquire those tasks. The pseudocode for the proposed scheduler is shown in Figure 4.8. The detection of when some task have finished is done through a flag, corresponding to Step 6 of the algorithm.

In the developed algorithm three task lists are maintained: the task_running list, the task_waiting list and the task_paused list. In the task_running list are present the tasks that were scheduled and are being executed on the cores. In the task_waiting list are present the new tasks that arrived to the scheduler and were not yet scheduled. The scheduler starts by picking one task from the task_waiting list and uses the proposed scheduling approach to schedule it. First, the scheduler opts to schedule only to the cores that are empty in order to avoid exhaustive search on all existing cores (Steps 2.3-2.5). However, if there is no available cores, the scheduler has no choice but finding the best of all existing cores (Steps 4). There is also the option to not schedule the task and wait until some task finishes, which could lead to lower energy consumption than scheduling the task to an occupied core (Step 3). If eventually this could be the decision, then the task will be transfered from the task_waiting_list to the paused_list and will stay there until some core becomes available (Step 5). As already mentioned, once one task finishes it is done a reschedule of all executing tasks. To do so, the tasks in the running_list are inserted in the top of the waiting_list followed by the tasks in the paused_list and the remaining tasks already present in the

waiting_list. And then, the scheduler executes the tasks reschedule (Step 6.).

Pseudocode of the Scheduler's algorithm.

- Input: Task t_i
Output: Schedule of task t_i to core c_j with freq f_w
0. Scheduler waits until new tasks appear.
 1. Scheduler enqueues the new tasks in the waiting list.
 2. If there are tasks on waiting list, proceed, otherwise go to Step 0.
 - 2.1. Check all players available, i.e., the ones with at least one core unoccupied.
 - 2.2. If there are no available players then go to Step 3.
 - 2.3. Send the task for each available player and wait to receive all player's bids.
 - 2.4. Select the player with the lowest bid and schedule the task t_i to it.
 - 2.5. Change the players' frequencies according to the bid of the winning player. Go to Step 2.
 3. Send the task to the player who will be available sooner and compute the bid as scheduling the task just when the player becomes available, i.e., to not scheduling the task now and wait until some player is available.
 4. Send the task and compute the bid for each core in each player.
 5. If it is better to not schedule the task now, then send the task to the pause_list and wait until some player becomes available. Otherwise, schedule the task to the winner core and proceed to Step 2.
-
6. If some task finishes, insert the already executing tasks on the top of the waiting_list followed by the tasks in the paused_list and proceed to Step 2 (reschedule).
-

Figure 4.8: Pseudocode of the Scheduler's algorithm.

It also should be noted that the bid process could be parallelized, however, that would let to increase the energy consumption. In order to do that, it would be necessary to schedule copies of the new task to each core and let them to compute the bids. This is doable, but it must be taking into account that the current instantaneous power consumption instead of be increased proportionately to one task, it will be increase by the number of copies, and also, there will be more stress and conflicts on the caches and the system. For those reasons it was opted to do the bidding process in a serialized mode.

4.3 Time and Power prediction for several frequencies

In the proposed approach it is necessary to compute the energy consumption for the different execution maps. Each frequency will lead to a different execution map due to different execution times of each task. There are two ways to compute the bid for each frequency. On one hand, it is possible to measure the power consumption and performance counters for one frequency, and then, wait until the DVFS driver changes the frequency and proceed the readings again. This way is called exhaustive search because one must change and wait for each frequency scaling and respective readings and becomes impractical when there are many available frequencies. On the other hand, it would be interesting to pass this physically exhaustive search to the compute domain by making predictions based on previous measures. In the following sections will be explained how to predict an execution time or an instantaneous power consumption value for other frequencies without physically changing it.

4.3.1 Task execution time

In order to estimate the task execution time, it will be necessary two performance event counters for each task. The necessary performance events are the number of instructions architecturally executed and the number of CPU cycles, which can be obtained through PAPI, as already seen in the section 2.2.2. However, the total number of instruction of the respective task must be already known. This value must be stored in the task history unit in order to be possible to make an estimation of the task execution time. Starting by knowing the time to execute just one instruction, $\Delta t_{instruction}$, and then by multiplying that time with the total number of instructions, $\#Total_instructions$, it is possible to estimate the task execution time for the current frequency. As show in equation 4.7, the $\Delta t_{instruction}$ can be calculated by knowing the time duration of the necessary number of cycles to execute, in average, one instruction, which is the time correspondent of one cycle, $1/frequency$, multiplied by the number cycles per instruction ratio, CPI . Once known all the values, is then used the Equation 4.8 to estimate the task execution time.

$$\begin{aligned}
 Execution_time &= \Delta t_{instruction} \times \#Total_instructions \\
 \Delta t_{instruction} &= \frac{\#cycles}{\#instructions} \times \Delta t_{cycle} = CPI \times \Delta t_{cycle} \\
 \Delta t_{cycle} &= \frac{1}{frequency}
 \end{aligned} \tag{4.7}$$

$$Execution_time = \frac{CPI}{frequency} \times \#Total_instructions \tag{4.8}$$

This estimation assumes that the task has always the same behavior when it is being executed. However, as mentioned in section 4.2, tasks can have different execution phases and can be more compute bound in one phase and more memory bound in another. In this estimation, the current measure of the CPI is what represents the tendency of task phases, and for that reason, it must be measured always when a new execution map is created to compute a bid, which occurs when a new task is about to be scheduled or when the scheduler effectuates a reschedule when one task finishes. Generally, memory bounded tasks have higher CPI than compute bounded tasks, because in average it must wait more cycles to load or store data then to compute.

To predict the CPI for other frequencies it will be used the reference CPI values measured when the task is running solo on the device (CPI solo value) and also the actual CPI measured value. In the task history unit, it is assumed to be saved one CPI solo value of that task for each available frequency.

Based on two developed tasks, that will be presented in the experimental results, section 5.3, it was seen that in this board, the compute bounded tasks have the same CPI in each frequency, which is true because CPU bounded tasks are mainly dependent on the frequency of the core. And it was also seen, that memory bounded tasks have different CPI values on different frequencies; they are more dependent on the memory's frequency than the frequency of the core, and so, they wait more cycles to get the data from the memory, which is operating at a constant and different frequency than the cores. The tendency in the variation of CPI values are used to characterize the behavior of the task, and so it must be preserved when it is necessary to predict the CPI to other frequencies. To do so, it will be

assumed that the relation between the new CPI measure and the CPI solo value is the same in each different frequency, and so based on the CPI measure for the actual frequency and the CPI solo values stored for the same frequency and the other frequency to predict, it will be possible to predict the CPI value to that frequency, as shown in Equation 4.9.

$$\frac{CPI_{f_0}}{CPI_{solo_{f_0}}} = \frac{CPI_{f_1}}{CPI_{solo_{f_1}}} = \frac{CPI_{f_2}}{CPI_{solo_{f_2}}} \quad (4.9)$$

$$CPI_{f_x} = \frac{CPI_{f_y}}{CPI_{solo_{f_y}}} \times CPI_{solo_{f_x}}$$

Figure 4.9 shows an example of the CPI value variation in a memory bounded and a compute bounded tasks for different frequencies when they are being executed solo and with the other on the same core.

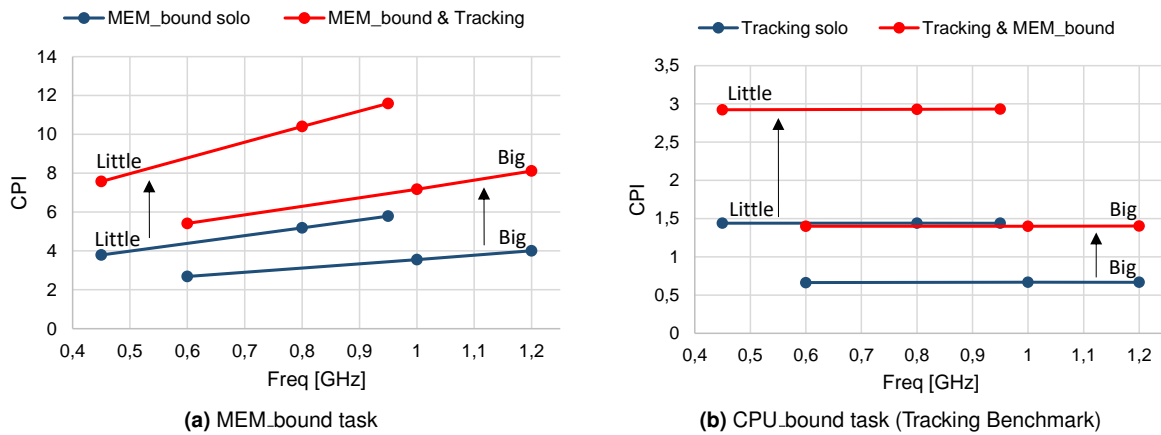


Figure 4.9: Comparison and variation of the CPI tendency between MEM_bound and CPU_bound applications.

As seen in the example, the blue and red lines represent the CPI tendency when the task is being executed solo and together, respectively, on the same core. It can be seen that the CPI increases when the two tasks are executed in the same core. This impact is mainly due to the "equal" time-sharing SCHED_OTHER policy but also due to the context-switching/conflicts between each other in the shared L1 and L2 caches. If other task was also scheduled to another core in the same cluster, we would also see a CPI increase, but just due to the conflicts in the shared L2 cache. It can be seen that the CPI tendency is approximately preserved when the task is being executed solo or with another tasks. And so, by predicting the task CPI value for the remaining available frequencies, it is possible to estimate the task execution times without physically doing exhaustive search.

4.3.2 Task instantaneous power consumption

The cumulative energy consumption meter register present in the performance measure unit will be used to predict the instantaneous power consumption of one task to different frequencies. As already mentioned in section 2.2.1, the instantaneous power consumption of the cluster can be obtained by dividing the difference between two cumulative energy measures by the time that passed, as shown in Equation 4.10. The time duration can be obtained by computing the difference between the two time readings obtained through a PAPI function (PAPI_get_real_usec).

$$Power_consumption = \frac{Energy_after - Energy_before}{\Delta Time} \quad (4.10)$$

$$\begin{aligned} Power_dissipation &= Power_Dynamic + Power_Static \\ &= Power_Processing + Power_Leakage \end{aligned} \quad (4.11)$$

Power dissipation in CMOS circuits can be defined by the processing power dissipation and the transistors static power leakage (Equation 4.11). Static power leakage is present at a micro-level in the transistors and is based on the small currents that are flowing between the differently doped parts of the transistor. This leakage currents tends to relatively increase when the size of the transistors is reduced and also when the temperature increases. However, it still represents the minor part of the power dissipation. *Power_Processing* represents the switching power dissipation due to charging and discharging the output capacitances when the processor is executing tasks. As referenced in [21], this dynamic power consumption accounts for more than 60-70% of the total power dissipation of a processor, and therefore, it seems reasonable to ignore the static power consumption of the processor. The approximated power consumption of the processor can be now estimated based on the power consuming transitions of all the transistors when the processor is executing tasks (*Power_Dynamic*), which is shown in equation 4.12.

$$Power_consumption = \alpha C \times f_{clk} \times V_{dd}^2 \quad (4.12)$$

In this equation, α represents the activity factor, i.e., the fraction of the circuit that is switching, C is the switched capacitance, f_{clk} is the frequency of the cluster and V_{dd} is the supply voltage of the cluster. In the switching transition, if the capacitance C is charged and discharged by f_{clk} frequency and peak supply voltage V_{dd} , then the charge moved per cycle is $C \times V_{dd}$ and the charge moved per second is $C \times f_{clk} \times V_{dd}$. Since the charge is delivered at voltage V_{dd} , then the power consumption is $C \times f_{clk} \times V_{dd}^2$. However, the existing processors on the market, generally do not have available the data needed to accurately compute the activity factor of the processor, and so, in this work, it will be also assumed that the power consumption can be computed by using the equation 4.13 [21]. In this equation is also shown, how it can be predicted the power consumption for frequency 0 by knowing the actual power consumption for frequency 1 and the voltage supply values for each frequency.

$$\begin{aligned} P &\propto f_{clk} \times V_{dd}^2 \\ \frac{P_{f_0}}{P_{f_1}} &= \frac{f_0 \times V_{f_0}^2}{f_1 \times V_{f_1}^2} \end{aligned} \quad (4.13)$$

It should be noted that these power estimations have errors associated due to the approximations. However, these power estimations are just used if there is no previous "real" measures for that task combination, saved in the task history unit. This unit stores both "real" power measures and power predictions of each task combination. The unit is composed by three lists: one for the power estimation of task combinations in the Cortex A-72; another for the Cortex A-53, and other for the "system". The power consumption of the system is not dependent on the frequency of the CPU and hence there is no need to predict to the other frequencies, because it is approximately constant. As already mentioned,

when one task finishes it is done a reschedule of the executing tasks. When this occurs, the tasks must be paused in order for the cluster power consumption return to the value when no task was scheduled. This must be done because the reading of power consumption for the first task to be rescheduled must be only its own power consumption and not the overall cluster power consumption of all executing tasks. The tasks must have integrate code to pause it whenever the scheduler sends the reschedule signal, which will be explained in section 5.2.

4.4 Summary

In this chapter the proposed framework was presented. First, a general overview of the framework was made, where the concepts behind the proposed auction and game theoretic approaches were presented. It was seen that the Auction approach assures a non-cooperative game between the players and Nash Equilibrium approach assures that the player selects the frequency that leads to the best outcome for that player based on its utility function.

Afterwards, some adaptations of the general framework to the ARM Juno r2 board were made. The limitation of that the power consumption measures were available just at the level of the cluster, has imposed cooperation between the cores (sub-player) inside the cluster (player) and non-cooperation between the clusters through the auction approach.

Finally, was explained how the task execution time as well as the task instantaneous power consumption are estimated and predicted by using the performance measure unit.

5

Experimental Evaluation

Contents

5.1	Experimental setup	52
5.2	Benchmarks	54
5.3	Experimental results	56
5.4	Summary	64

In order to evaluate the proposed energy-aware game-theoretic scheduling approach on mobile devices, it was used the ARM Juno r2 board. It is a heterogeneous processing unit composed by a dual-core Cortex-A72 and quad-core Cortex-A53. These ARM processor architectures are dominating the market of embedded systems and are now being used on the new high-end smartphones. The ARM Juno r2 board supports the ARM's big.LITTLE technology which provides power-consumption optimization and high computational performance.

In this chapter will be presented the necessary experimental setup to evaluate the proposed framework against the Linux kernel, the discontinued ARM's GTS and the new EAS scheduling approaches. Furthermore, it will be presented the benchmarks used to evaluate the proposed framework. As it will be seen, these benchmarks will need to be slightly modified in order to be used by the framework. Afterwards, the task power consumption and execution time predictions for different frequencies are experimentally evaluated. And finally, the proposed framework is evaluated with different benchmark workloads and the obtained experimental energy savings are presented.

5.1 Experimental setup

The proposed energy-aware scheduling approach was experimentally tested on the ARM Juno r2 board, which has already been introduced in section 2.1. This board can run different software stacks developed by both ARM and Linaro, which include the board firmware, system control processor firmware, ARM Trusted Firmware, OP-TEE Trusted Execution Environment, and then either an EDK2-based UEFI environment, or a Linaro kernel and filesystem (Android / BusyBox / OpenEmbedded (OE)) booted via U-Boot. To evaluate the proposed scheduler, the ARM Juno r2 board will be configured with a Linaro kernel and a OpenEmbedded filesystem following the instructions present in ARM's "Using Linaro's deliverables on Juno" web document [22].

The configuration is based on two main softwares, the Juno's Flash and the Filesystem. The Juno's Flash is the software that contains the Linaro's kernel and must be inserted on Juno's Multi Media Card (MMC) in order for the board to boot and load that kernel. The MMC is accessed through the usb configuration port (see Figure 5.1). When it is booting up, the board will search for a filesystem. This, must be burnt on a USB stick with a minimum capacity of 4GB, and must be inserted in one of the USB 2.0 ports. In this work it was used a Samsung 850 Pro SSD that was already available for that board. Once the board's boot is done, it reaches a prompt, where you can start using the system.

There are two available Linaro kernels, the Linaro Stable Kernel (LSK) and the Linaro Tracking Kernel (LTK). However, only the LSK incorporates the big.LITTLE MP patchset produced to support scheduling on heterogeneous multi processor systems. And so, the LSK is the one that supports the GTS and EAS scheduling approaches.

The filesystem chosen to configure the board was the OE LAMP, because it is a complete filesystem and has more available tools/drivers than the OE Minimal, Android or Busybox filesystems. The OpenEmbedded filesystem is composed by the files and folders of a Linux operating system, and thus, it is very similar to Ubuntu, for example. Each version of the available OE filesystems contains a /boot

folder, in which is present the kernel that has the same kernel headers as the filesystem. However, this kernel is not used by the board when it is used the LSK in the Juno's Flash, but it can be loaded when an Unified Extensible Firmware Interface (UEFI) Flash is used. The UEFI Flash contains no kernel and it can be used to load the kernel from the USB boot folder. The software stack combinations used to setup the board and to evaluate the proposed framework are shown in Table 5.1.

Table 5.1: Available ARM Juno r2 software's combinations.

Flash	Filesystem	Kernel	PAPI	Energy meters
UEFI 16.04	OE LAMP 15.09	3.10.0-1-linaro-lt-vexpress64	Yes	Yes
LSK 16.05	OE LAMP 15.09	3.18.31 (GTS)	No	Yes
LSK 16.06	OE LAMP 15.09	3.18.34 (EAS)	No	Yes

The fact that the filesystem do not contain the correct kernel headers for the LSK 16.05 and 16.06 kernels, some of the modules are not loaded when the board is booting up. Due to this, it was seen that PAPI could not be successfully installed on those LSK configurations. In ARMConnected Community [23] is referenced that the LSK kernels have no support to read the energy meters registers, which is only supported in the LTK kernels. However, some basic experiments were done in both LSK and LTK kernels and it was seen that the instantaneous power consumption values obtained were the same, which reveals that LSK also supports the reading of energy meters registers. This was communicated to support@arm.com and was later confirmed that in fact the LSK also supports the energy readings.

The proposed framework will be tested in the Linaro kernel present in the /boot folder of the OE LAMP 15.09 filesystem by using the UEFI 16.04 flash and will be compared with the GTS and EAS scheduling approaches present in LSK 16.05 and LSK 16.06 kernels respectively. It should be noted that each configuration has a different kernel.

In Figure 5.1 is present the ARM Juno r2 board where are shown the used ports. When the ARM Juno r2 board is working with the UEFI 16.04 flash, the Ethernet port that must be used is the one at the front panel. The rear panel Ethernet port is used by the LSK flashes.



Figure 5.1: ARM Juno r2 board available connection ports.

5.2 Benchmarks

Benchmark applications are used to test and evaluate the proposed scheduling approach. The selection of these benchmarks must be done based on following existing limitations:

- In order for the benchmark to be scheduled, it must be visible by the scheduler as a thread. This can be done by inserting the benchmark C code inside the proposed scheduler code, as a function, and compile it, or by using a system call to execute an already compiled binary of that benchmark;
- PAPI only supports performance measurements at the level of the thread. When a thread is created, it inherits no PAPI events or information from the calling thread. Each thread must create, manipulate and read its own counters [10];
- When some executing benchmark finishes, the scheduler sends a signal to pause the remaining executing threads, as seen in section 4.3.2, in order to reschedule them. The benchmarks must include a C code to be paused when detected the reschedule signal. This code is shown in Algorithm 5.2 and must be inserted within a main loop of the benchmark so that it can have a quick response. The scheduler's algorithms to pause and unpaue the threads are shown in Algorithm 5.1 and 5.3 respectively.

Algorithm 5.1 Scheduler pause_thread function

```
1: void pause_thread_X(int thread_number){
2:   printf("pausing thread %d\n", thread_number);
3:   pthread_mutex_lock(&threads_array[thread_number].mutex_t);
4:   threads_array[thread_number].pause_flag = 1;
5:   pthread_mutex_unlock(&threads_array[thread_number].mutex_t);
6: }
```

Algorithm 5.2 Thread pause code

```
1: (Benchmark code ...)
2: pthread_mutex_lock(&this_thread->mutex_t);
3: while this_thread->pause_flag == 1 do
4:   pthread_cond_wait(&this_thread->cond_t, &this_Thread-> mutex_t);
5: end while
6: pthread_mutex_unlock(&this_thread->mutex_t);
7: (Benchmark code ...)
```

Algorithm 5.3 Scheduler unpaue_thread function

```
1: void unpaue_thread_X(int thread_number){
2:   pthread_mutex_lock(&threads_array[thread_number].mutex_t);
3:   threads_array[thread_number].pause_flag = 0;
4:   pthread_cond_signal(&threads_array[thread_number].cond_t);
5:   pthread_mutex_unlock(&threads_array[thread_number].mutex_t);
6: }
```

Having said that, it must be implemented in the source code of each benchmark the necessary PAPI functions to create an event set and start the reading of counters as well as the code to pause the thread when the scheduler sends the reschedule signal.

Multi-threaded benchmarks require in-depth knowledge of the algorithm in order to know where must be inserted the C code modifications, specifically where the threads are being created. Each benchmark can be composed by many C code files and some of them requires installation of additional libraries, which became impractical to find where the C code must be inserted. For those reasons, the developed framework will just focus on single-threaded benchmarks.

These however, also requires some knowledge of the algorithm, specially to know where must be inserted the C code to pause the thread. The benchmarks must have a quick response to the scheduler pause signal, and to accomplish that, the C code must be inserted within a main short loop. The PAPI functions to create an event set and start the reading of counters can be easily inserted at the beginning of the benchmark main function.

To evaluate the developed scheduling approach, the following benchmark suites were considered: the Princeton Application Repository for Shared-Memory Computers (PARSEC) [24], which contains 10 applications from many different areas such as computer vision, video encoding, financial analytics, animation physics and image processing; the Standard Performance Evaluation Corporation (SPEC) CPU 2006 [25], which contains 13 CPU-intensive applications, and The San Diego Vision Benchmark Suite (SD-VBS) [26], which contains 9 diverse vision applications, such as image processing, image analysis, motion and tracking, with the respective input sets. It was also used the OpenBlas library [27], which contains linear algebra functions.

It has been tried to individually compile and configure each one of these benchmark applications to execute with an adequate input set. In order to simulate user applications with a duration between 7 to 10 seconds in the lowest frequency of Cortex-A53, and also, with a quick response to the scheduler pause signal, the benchmarks can be configured in two ways. On one hand, it can be selected the smallest input set for the benchmark and execute it many times or, on the other hand, the benchmark can be executed with a large input set just once. The second way requires in-depth knowledge of the application because it must be seen which is the main loop of the application or where it takes most of the time executing in order to insert the pause code there to achieve a quick response when the signal is received. The first way is more appealing because the benchmark response to the pause signal will be related with the time duration to execute the smallest input set. For some benchmarks that were successfully compiled, the pause code was inserted in the main loop as well as inside other functions of the benchmark. In Table 5.2 are presented the benchmarks that were successfully compiled and that met the requirement of having a quick response to the pause signal, as well as their respective input set and the number of repetitions. A brief description of each one of these benchmark applications is presented in Table E.1 on the Appendix E. The most delay on the response to the pause signal in all successfully compiled benchmarks was 10ms and occurs when that application is being executed at the lowest frequency on the Cortex-A53. Therefore, the scheduler must wait approximately 10ms after sending the signal to start reading the power consumptions correctly.

Table 5.2: Successfully compiled benchmarks and respective configuration.

SD-VBS			OpenBLAS		
Benchmark name	Input set	Repetitions	Benchmark name	Input set	Repetitions
Disparity	test	8000	Sgemm	random values	2000
Mser	test	3400	Sgemv	random values	2000
Stitch	test	5000	Sscal	random values	200000
Texture Synthesis	test	2000	Saxpy	random values	80000
Tracking	test	500	Sdot	random values	80000
PARSEC			SPEC CPU2006		
Benchmark name	Input set	Repetitions	Benchmark name	Input set	Repetitions
Blacksholes	in.4.txt	5000	Bzip2	sample4.ref	2500

5.3 Experimental results

Different benchmark applications were selected to evaluate experimentally the proposed framework. Each benchmark was executed individually to evaluate the average number of cycles per instruction, total number of instructions and the average power consumption of each cluster and of the system. These values will be referenced as the "task solo values" and were obtained for each one of the three available frequencies in each cluster. The solo values were stored in the task history unit and are used to predict the power consumption and execution time of the task to other frequencies without changing it, as already mentioned in section 4.3. The experimentally measured solo values for each benchmark are present in Table F.1 on the Appendix F. In table 5.3 are just shown the CPI solo values of some selected benchmarks for both clusters. The remaining benchmarks in each respective suite have similar CPI values to the selected ones as can be seen on the Appendix F. All the values obtained experimentally in this chapter were obtained by computing the median for 10 measures in order to have consistent results. To simplify the reading, the frequencies f_0 , f_1 and f_2 shown in Table 5.3 for Cortex-A53 corresponds to 450 MHz, 800 MHz and 950 MHz, respectively, and for Cortex-A72 corresponds to 600 MHz, 1 GHz and 1.2 GHz, respectively.

Table 5.3: Cycles per instruction tendency on different benchmarks.

Benchmark	Cortex-A72 (big)			Cortex-A53 (LITTLE)		
	f_0	f_1	f_2	f_0	f_1	f_2
Disparity	0.898	0.897	0.897	1.337	1.336	1.337
Tracking	0.664	0.667	0.667	1.440	1.439	1.439
Blacksholes	1.206	1.028	1.207	1.515	1.523	1.523
Sgemm	0.624	0.624	0.624	1.340	1.344	1.343
Sgemv	1.011	1.009	1.012	1.301	1.301	1.301
Bzip2	0.538	0.539	0.538	1.109	1.109	1.109
CPU_bound	1.169	1.169	1.169	1.124	1.124	1.124
MEM_bound	2.686	3.552	4.009	3.793	5.191	5.792

As can be seen in Table 5.3, the CPI value of one task executing on a LITTLE core is always higher

than 1, which is due to the fact that the Cortex-A53 is an in-order processor. On the other hand, the CPI values in a big cluster core can be lower than 1 because the Cortex-A72 is an out-of-order processor, as already mentioned in section 2.1.2, and so, the execution of task instructions can be more parallel on Cortex-A72 than on Cortex-A53.

It is possible to see in Table 5.3 that each benchmark from the available benchmark suites has an approximated constant CPI value for each frequency, which allows to conclude that those benchmarks have a compute bound behavior, as already explained in section 4.3.1. To prove this, it was created one pure compute bound task, CPU_bound, as well as one memory bound task, MEM_bound, which C codes are shown in Algorithm 5.4 and 5.5 respectively.

Algorithm 5.4 CPU_bound task C code to ARMv8 architecture

```
1: int i;
2: int add = 0;
3: int arg1 = 100;
4: for (i = 0; i < 25000000; i++) do
5:     asm (
6:         "ADD %[result], %[a], %[b]"
7:         : [result] "=r" (add)
8:         : [a] "r" (arg1), [b] "r" (add)
9:         );
10: end for
```

Algorithm 5.5 MEM_bound task

```
1: int loop = 2500000;
2: int rand_val, i;
3: srand(21);
4:
5: int *num1 = (int *)malloc(20000000 * sizeof(int));
6: ...
7: int *num8 = (int *)malloc(20000000 * sizeof(int));
8:
9: for (i = 0; i < 20000000; i++) do
10:     num1[i] = rand();
11:     ...
12:     num8[i] = rand();
13: end for
14:
15: while loop != 0 do
16:     num6[rand_val] = num1[rand_val];
17:     num3[rand_val] = num2[rand_val];
18:     num5[rand_val] = num7[rand_val];
19:     num4[rand_val] = num8[rand_val];
20:
21:     num6[rand_val+100000] = num1[rand_val+100000];
22:     num3[rand_val+100000] = num2[rand_val+100000];
23:     num5[rand_val+100000] = num7[rand_val+100000];
24:     num4[rand_val+100000] = num8[rand_val+100000];
25:
26:     loop - -;
27: end while
28:
29: free(num1);
30: ...
31: free(num8);
```

On one hand, the CPU_bound task represents a loop composed by an ADD assembly function, where one value, *add*, is accumulating the other, *arg1*. During the loop execution, the task only needs to access the registers, L1 and L2 cache because once the two values are read from the DRAM memory and stored to the cache, it is not needed to do more memory accesses, and so, the task is mainly dependent on the CPU frequency than the DRAM memory frequency. On the other hand, the MEM_bound task needs much memory space to save all its values. The overall necessary memory allocation for the $8 \times 20.000.000$ integers is approximately 610 Megabytes, assuming that an integer is represented by 4 bytes. Knowing that the Cortex-A72 has a 2 MB L2 cache and Cortex-A53 has a 1 MB L2 cache, one can conclude that in each loop iteration the task will need to access the DRAM memory to read and store the values, which represents a pure memory bound behavior. The solo CPI values obtained for the CPU_bound and MEM_bound tasks are also shown in Table 5.3 and prove that these two types of tasks assume different CPI tendencies when the frequency is changed. In the MEM_bound task, the CPI increases with the CPU frequency because it takes more CPU cycles to execute the same instruction, once the DRAM memory frequency does not change.

Based on the solo values present in the Table F.1 on the Appendix F, it is possible to see how the power consumption and execution time predictions are calculated. First, the prediction of power consumption is based on equation 5.1.

$$\frac{P_{f_0}}{P_{f_1}} = \frac{f_0 \times V_{f_0}^2}{f_1 \times V_{f_1}^2} \quad (5.1)$$

The measured supply voltages of each cluster are 0.833 V, 0.914 V and 1.014 V, which represents V_{f_0} , V_{f_1} and V_{f_2} respectively. These values are similar to the ones referenced in the manual, as seen in section 2.3.5.

Given the measure of the task power consumption at the current frequency, it is expected that the equation 5.1 could estimate approximately the power consumption of the same task for other frequencies. In Table 5.4 are shown the errors between the predictions and the solo power consumption measured value of Blacksholes benchmark for each frequency.

Table 5.4: Evaluation of power consumption prediction for the Blacksholes benchmark.

		Cortex-A72 (big)			Cortex-A53 (LITTLE)		
		f_0	f_1	f_2	f_0	f_1	f_2
Power [mW]	f_0	220,74	442,93	654,19	83,84	179,45	262,28
	f_1	207,41	416,19	614,69	73,41	157,13	229,65
	f_2	210,71	422,80	624,45	74,27	158,97	232,35
Relative errors (%)	f_0	0,00	6,43	4,76	0,00	14,21	12,88
	f_1	-6,04	0,00	-1,56	-12,44	0,00	-1,16
	f_2	-4,55	1,59	0,00	-11,41	1,17	0,00

In each row of the table it is marked as gray the solo power consumption value for that frequency and the other two values on that row are the estimated predictions for the other frequencies. One can see that depending on the actual frequency, the predictions have associated errors between 2% and 15%

when compared with the measured values. However, this is a simple technique to predict the power consumption, which can be improved in future work in order to reduce the estimation errors.

To estimate the task execution time it is used the equation 5.2.

$$Execution_time = \frac{CPI}{frequency} \times (\#Total_instructions - \#Instructions_already_executed) \quad (5.2)$$

This equation uses the CPI measured value for the current frequency as well as the total number of instructions of a task, which is stored in the task history unit, and the measured number of instructions already executed. The equation estimates the remaining execution time of that task. To evaluate this, it was used the solo CPI values of MEM_bound task to estimate the execution times for each frequency, which were then compared with the real execution time measured. In Table 5.5 are shown the relative errors between the execution times estimations and the measured execution times for the MEM_bound task. As shown in Table 5.5, one can see that the associated errors between the estimated and measured execution times are approximated 1%.

Table 5.5: Evaluation of execution time estimation for MEM_bound task.

	Cortex-A72 (big)			Cortex-A53 (LITTLE)		
	f_0	f_1	f_2	f_0	f_1	f_2
CPI measured	2,686	3,552	4,009	3,793	5,191	5,792
Execution time estimated [ms]	2,843	2,256	2,122	5,353	4,121	3,872
Execution time measured [ms]	2,872	2,278	2,142	5,408	4,159	3,906
Relative error (%)	1,010	0,975	0,943	1,027	0,922	0,878

As already seen in section 4.3.1, the CPI solo values are used to predict the CPI value for other frequencies while preserving the CPI tendency between different frequencies. The CPI solo values of MEM_bound task were already presented in Table 5.3, and will now be used to predict its CPI values for other frequencies when both MEM_bound task and Tracking benchmark are executed in the same core. When this happens, it is expected a CPI increase in both tasks since just one can be executed at the same time slice due to the SCHED_OTHER policy. To predict the CPI values for other frequencies it must be computed first the relation between the MEM_bound CPI measure when the two task are being executed at current frequency and the respective MEM_bound CPI solo value for that frequency. Based on this relation, and assuming that it is the same to others frequencies (see Equation 4.9), it is possible to predict the CPI value for one frequency by already knowing the CPI solo value for that frequency. In Table 5.6 are shown the relative errors between the CPI value predictions and the measured CPI values for each frequency.

Similarly to Table 5.4, in each row of the table it is marked as gray the measured MEM_bound CPI value for that frequency when both MEM_bound and Tracking tasks are executing in the same core. The other two values on that row are the estimated predictions for the other frequencies. One can see that the CPI relations between the CPI measured and solo for one frequency is approximately equal to the other frequencies, which proves that the CPI tendency is preserved. And it can also be seen that the

Table 5.6: Evaluation of MEM_bound CPI prediction.

		Cortex-A72 (big)			Cortex-A53 (LITTLE)		
		f_0	f_1	f_2	f_0	f_1	f_2
MEM_bound CPI solo (1)		2,686	3,552	4,009	3,793	5,191	5,792
MEM_bound CPI (MEM+Tracking) (2)	f_0	5,417	7,164	8,086	7,575	10,368	11,568
	f_1	5,421	7,168	8,091	7,599	10,401	11,604
	f_2	5,435	7,188	8,113	7,591	10,391	11,593
CPI Relation $\left(\frac{(2)}{(1)}\right)$		2,017	2,018	2,024	1,997	2,004	2,001
Relative errors (%)	f_0	0,000	-0,058	-0,334	0,000	-0,312	-0,214
	f_1	0,058	0,000	-0,276	0,313	0,000	0,098
	f_2	0,335	0,276	0,000	0,214	-0,098	0,000

observed predictions errors are lower than 1% when both MEM_bound and Tracking benchmarks are executing in the same core.

Once evaluated the tasks power consumption and execution time predictions functions, it will now be used combinations of benchmark applications to evaluate the proposed scheduling approach. The MEM_bound task is the first task used to evaluate the decision of the proposed scheduler and to compare it with the Linaro's kernel 3.10 decision. In Table 5.7 are shown some of the MEM_bound solo values used to estimate the overall energy consumption on the board when executing the MEM_bound task.

Table 5.7: Evaluation of MEM_bound task energy consumption.

	Cortex-A72 (big)			Cortex-A53 (LITTLE)		
	f_0	f_1	f_2	f_0	f_1	f_2
Cluster power consumption [mW]	228,9	404,4	589,3	78,5	139,4	202,7
System power consumption [mW]	833,2	844,8	848,5	805,9	819,2	816,4
Execution time [ms]	2,9	2,3	2,1	5,4	4,2	3,9
Cluster energy consumption (1) [mJ]	657,5	921,2	1262,0	424,3	579,6	791,8
System energy consumption (2) [mJ]	2392,7	1924,4	1817,1	4358,5	3406,9	3188,8
(1) + (2) energy consumption [mJ]	3050,2	2845,6	3079,0	4782,8	3986,5	3980,7

As can be seen in Table 5.7, the best frequency to execute the MEM_bound task, having just into account the energy consumption of the cluster, is the lowest frequency of Cortex-A53, which makes sense because the task is not mainly dependent on the CPU frequency, and so, it can be reduced to achieve lower power consumption in the CPU during its execution. However, it can be seen that the lowest energy consumption by the system is achieved by selecting the highest frequency of Cortex-A72. In this case the system power is much higher than the core power, and so, it will be dominant in the decision. Although is known the best individually frequencies to select at the cluster level and at the system level, it must be selected the frequency that minimizes the overall energy consumption in the device. In this case, to achieve that, the MEM_bound task must be scheduled to a Cortex-A72 core and the frequency f_1 , 1 GHz, must be selected. As already mentioned in section 4.2, the GPU was not used. During the task executions, the GPU stays in Idle mode with a constant power consumption of approximately 78 mW, and for those reasons, it was not taken into account in the decisions.

In the experimental results, it could be seen that the proposed scheduler decide to execute the MEM_bound task on a Cortex-A72 core with the frequency f_1 . The Linaro's kernel 3.10, with the ondemand governor selected, decides to execute the task at f_0 for half the execution time and at f_2 for the remaining time in the same core. And, with the interactive governor selected the scheduler decides to execute the task at f_2 during the total execution time, also in the same core. One can see that neither the ondemand and interactive governors selected the best frequency. As can be seen in Table 5.8, the proposed scheduler achieves 8% and 11% energy savings, when compared it with the ondemand and interactive governors, respectively.

The proposed scheduler was also evaluated for different benchmark combinations. In Table 5.8 are shown the energy consumption as well as the execution time for different benchmark combinations. It is also presented the achieved energy savings between the proposed approach and the Linaro, GTS and EAS scheduling approaches present in different kernels. To provide a good evaluation of the proposed framework, it will be tested different scenarios, where the number of tasks to schedule is higher, equal and lower than the number of available cores. The MEM_bound task will be inserted in some benchmark combinations in order to not have only CPU_bounded tasks on the workload.

Table 5.8: Experimental results for each benchmark combination used to evaluate the proposed framework. The energy consumption values represents the overall energy consumed by the ARM Juno r2 board until all tasks completes their execution.

Benchmarks	Proposed Scheduler			Linaro Kernel 3.10 - UEFI 16.04			LSK 16.05 (GTS)			LSK 16.06 (EAS)		
	Energy Consumption [mJ] E_0	Execution Time [s]		Energy Consumption [mJ] E_1	Execution Time [s]	Savings (%) $\frac{E_1 - E_0}{E_1} \times 100$	Energy Consumption [mJ] E_1	Execution Time [s]	Savings (%) $\frac{E_1 - E_0}{E_1} \times 100$	Energy Consumption [mJ] E_1	Execution Time [s]	Savings (%) $\frac{E_1 - E_0}{E_1} \times 100$
1 - MEM_bound												
userspace	2837	3,213	ondemand	3116	3,463	8,955	2983	3,428	4,904	3004	3,391	5,568
			interactive	3196	3,128	11,243	2747	2,942	-3,246	2711	2,948	-4,623
2 - Tracking												
userspace	2237	2,503	ondemand	3496	4,514	36,004	3319	4,308	32,582	2447	3,018	8,570
			interactive	2290	2,542	2,282	2206	2,484	-1,429	2198	2,488	-1,772
3 - MEM_bound, MEM_bound, and CPU_bound												
userspace	5801	5,202	ondemand	7560	5,43	23,27	7034	5,104	17,535	5646	4,392	-2,747
			interactive	7986	4,93	27,37	7103	4,624	18,330	5384	3,757	-7,733
4 - MEM_bound, Tracking, Mser, Sgemm												
userspace	6411	5,000	ondemand	7505	5,026	14,579	6650	4,597	3,604	6998	4,589	8,391
			interactive	7429	4,313	13,711	6727	3,868	4,706	7058	4,137	9,173
5 - Blacksholes, Sdot, Bzip2, Saxpy, Texture_Synthesis												
userspace	11936	9,270	ondemand	16133	9,976	26,016	13017	7,907	8,301	15448	8,857	22,733
			interactive	16591	8,952	28,058	12000	6,528	0,532	14083	7,584	15,244
6 - Saxpy, Blacksholes, Texture_Synthesis, Stitch, Sscal, Disparity												
userspace	12813	8,671	ondemand	19969	12,250	35,836	15899	9,837	19,409	16491	9,829	22,302
			interactive	16133	8,606	20,582	14837	8,003	13,646	15595	8,514	17,841
7 - Bzip2, Blacksholes, Sgemm, Stitch, Mser, Disparity, Tracking, Texture_Synthesis, Sgemv												
userspace	14094	9,202	ondemand	17790	13,957	20,777	15928	9,955	11,513	15923	10,435	11,485
			interactive	17828	9,251	20,945	16582	8,527	15,005	16482	8,511	14,490

The proposed scheduler controls the frequency scaling and task migration from the user-space through an algorithm programmed in C language. It also uses the APB interface and PAPI to read the energy meter registers and the performance counters, respectively, from the user-space. All these user-space driven procedures, used to gather the necessary performance information in order for the scheduler to decide, have higher associated overheads than if it was possible to access these registers directly through the kernel. And so, the user-space proposed scheduler have higher overheads than the other scheduling approaches. In order for the proposed scheduler algorithm to have no influence in the

task executions, and also, to be as fast as possible like a kernel, the scheduler algorithm was executed in a dedicated Cortex-A53 core. In the experimental results, there are only three Cortex-A53 cores and two Cortex-A72 cores available to execute tasks. The remaining Cortex-A53 core is used to execute the scheduler algorithm on the proposed framework and is shut down in the remaining scheduling approaches of the other kernels. By adopting this, it is achieved fairness in the experimental results between the different scheduling approaches. However, it should be noted that the proposed framework will account with the power consumption of that dedicated core to run the proposed scheduler, which does not happen in other approaches. This can be improved in future work by passing the proposed scheduling functions to inside the kernel, and thus, using all the available cores to execute the tasks.

As can be see in Table 5.8, the proposed framework can achieve energy savings up to 36%, 32% and 22% when compared with the ARM's Linaro, GTS and EAS scheduling approaches. However, it can be unfair to compare the proposed scheduler, evaluated in the Linaro's kernel version 3.10, with the GTS and EAS scheduling approaches, which were evaluated in different kernels versions, 3.18.31 and 3.18.34, respectively. These kernels can have such improvements that are not present in the 3.10 kernel, and so, if one task is selected to be executed on both Linaro's and EAS approaches it can have different energy consumptions. This can be proved by looking at the MEM_bound task experimental results when the interactive governor was selected in both Linaro's and EAS kernel. As already mentioned in section 2.4.1, the interactive governor is more aggressive than the ondemand governor when is up to scaling the CPU frequency up in response to intensive computational activity. And it was seen, during the experimental evaluation, that in both Linaro's and EAS kernels when the same interactive governor is selected, the MEM_bound tasks is always being executed on one Cortex-A72 core at the highest frequency until it finishes. And so, in this case, the 15% energy savings observed between the Linaro's and EAS scheduling approaches suggests that other subsystems rather than the frequency scaling and task migration were improved, and thus, it can be unfair to compare the experimental results obtained for the proposed framework with the EAS and GTS scheduling approaches. Future work can focus on developing an performance unit framework to set and access directly the performance counters in order to not use PAPI and execute the developed scheduler on the same kernel of each respective scheduling approaches.

In benchmark combination number 7, the number of benchmarks is higher than the number of cores ($9 > 5$). In this case, the proposed scheduler consumes more energy to decide to which core the task must be scheduled than when the number of tasks is lower or equal than the number of cores, because every core of each player must receive the task to evaluate and bid the necessary energy consumption to execute it, as already seen in section 4.2.

In order to better understand the scheduling decisions of the proposed scheduler, Figure 5.2 shows the overall instantaneous power consumption (Cortex-A53 + Cortex-A72 + system) and frequency levels obtained during the execution of the benchmark combination MEM_bound, Tracking, Mser and Sgemm (benchmark combination number 4 on Table 5.8), for the proposed scheduler as well as for the ondemand governor selected on Linaro's kernel 3.10.

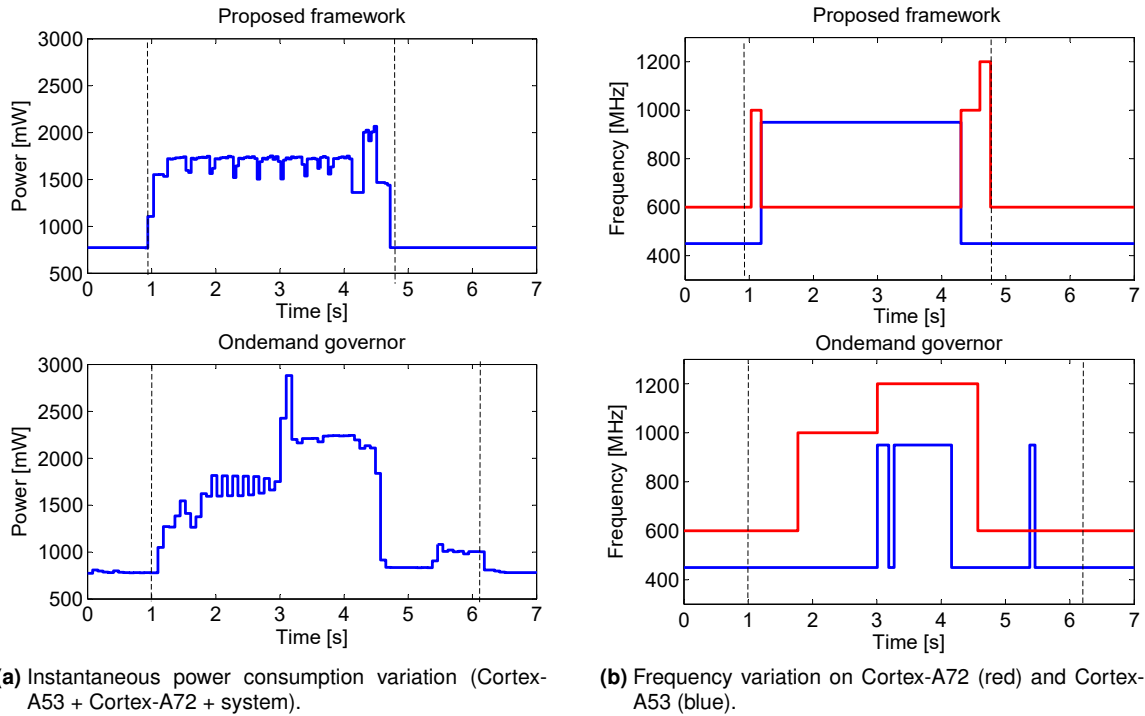


Figure 5.2: Comparison and variation of the CPI tendency between MEM_bound and CPU_bound applications.

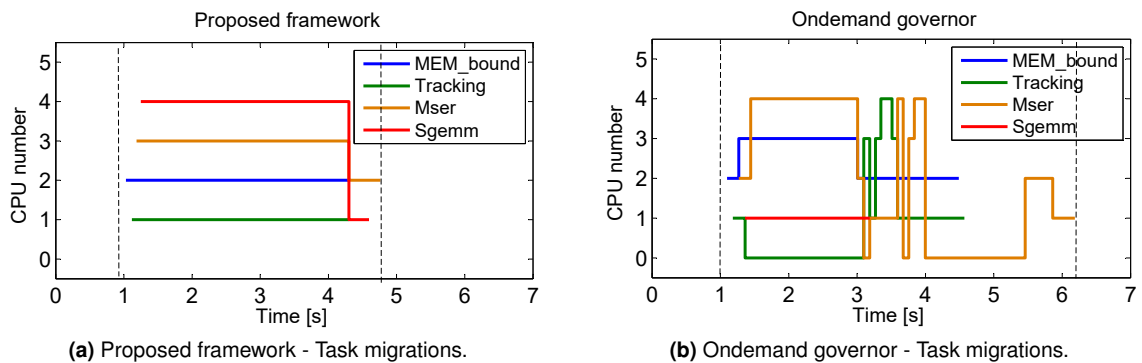


Figure 5.3: Comparison of run-time task migrations between the proposed framework and ondemand governor. Cortex-A72 CPUs: 1, 2; Cortex-A53 CPUs: 0, 3, 4 and 5.

On one hand, in this case, it can be seen that the Linaro's scheduling approach does more frequency scaling than the proposed scheduler. This approach uses the Cortex-A72 during more time and at higher frequencies than the proposed scheduler, which leads to higher power consumption as shown in Figure 5.2.a. The Linaro's scheduling approach, with the ondemand governor selected, takes more time to execute the same workload than the proposed scheduler, which can be related with the bad decisions of task migrations as can be seen in Figure 5.3.b. It can be seen that around the 3 seconds, there are more than one task being executed in the same core, which could triggered the frequency scaling and respective power consumption peak. On the other hand, the developed energy-aware game-theoretic scheduling approach executes the tasks on the same core until that task, or some other finishes, in order to perform the task rescheduling, which can be seen in the 4.5 seconds, when the Mser and Tracking

benchmarks were finished. The proposed scheduler tends to select the best frequency for every task combination executing at the time.

According to the experimental results presented in Table 5.8, the proposed framework was able to reduce the energy consumption when compared with the actual scheduling approaches developed by Linaro and ARM, which is based on the ARM big.LITTLE technology. It can be seen in the experimental results that when the interactive governor is selected, especially in the benchmark combinations number 1 and 2, which have just one task, the energy savings obtained are much lower than when the ondemand governor is selected. This is due to the interactive governor select the highest CPU frequency more often during the execution, which can be beneficent in those cases because the system power consumption is much higher than the Clusters power consumption, and so, by reducing the execution time, the dominant power consumption is also reduced. However, this does not mean that for all possible scenarios, the energy consumption of the system will be always the dominant one. It should be noted that in this proposed energy-aware game-theoretic scheduling approach some issues were left open, which can be improved in future work.

5.4 Summary

In this chapter the proposed energy-aware game-theoretic scheduling approach was experimentally evaluated. First, the available kernels and filesystems combinations used to setup the ARM Juno r2 board were presented, as well as their respective scheduling approaches, i.e., Linux, GTS and EAS.

Then, the benchmarks used to evaluate the proposed framework were also discussed. These benchmarks were modified in order to overcome the limitations faced, such as the necessity of a fast response to pause the benchmark when some other thread finishes, and also the initialization of PAPI counters in the beginning of each benchmark. It was presented the C language code for some of these modifications.

Afterwards, the custom developed MEM_bound and CPU_bound tasks were introduced in order to prove and explain the CPI tendency. It was also presented the solo values measured for each benchmark. Furthermore, it was experimentally evaluated the power consumption and execution time estimations, as well as, the respective relative errors between the estimated predictions and the measured values. It was observed that the execution time prediction have associated errors of 2% while power consumption predictions have higher associated errors of 15%.

Finally, the proposed framework was evaluated with the MEM_bound task in order to see how it decides the best frequency to use. Other benchmark combinations were then used to evaluate the proposed scheduler and the energy savings obtained for each benchmark configuration were presented. It was also shown in detail, the instantaneous power consumption, frequency scaling and the task migrations observed during the execution of one benchmark configuration, where could be compared the behavior between the Linaro's ondemand governor approach and the proposed scheduler.

6

Conclusions

Contents

6.1 Future work	67
---------------------------	----

Nowadays, there is an increasing demand of higher performance and low energy consumption mobile devices. Such devices have a battery, and hence, they are energy limited. In order to achieve lower energy consumption, these devices must have an energy-aware scheduler to decide in which processing unit should the tasks be scheduled. The main objective of this thesis was to develop an energy-aware game-theoretic scheduling approach for heterogeneous embedded systems such as the big.LITTLE from ARM, in which the overall energy consumption is the governing metric. The proposed framework uses an auction game approach as well as the Nash Equilibrium concept, from non-cooperative game theory, in order to design an energy-aware scheduler.

The existing ARM scheduling approaches were introduced in this thesis, a study focused on the discontinued Global Task Scheduling (GTS) approach and the new Energy-Aware Scheduling (EAS) approach that is currently being developed by ARM. On one hand, it was seen that the GTS approach was the first ARM's scheduling technique to use all "big" and "LITTLE" cores available in the system at the same time. However, this approach was discontinued because it relies on separated subsystems, such as the DFVS and CPUidle, that decide the adequate frequency and voltage levels to execute the tasks without any coordination with the scheduler. On the other hand, the new EAS approach, is focused on integrating these subsystems with the scheduler to give the necessary energy-awareness to the system.

Afterwards, the most fundamental concepts of game theory were introduced as well as some different game-theoretic approaches developed by other researchers. The state of the art study of game theory scheduling approaches led to understand the main advantages of each approach, and where the development should focus, in order to achieve the proposed objectives. Based on this study, it was seen that, in the low complexity and intuitively auction game based approach, the focus must be on how the bid is composed and computed, while in the Nash Equilibrium approach, the focus must reside on the formulation of the player's utility function that must be related to energy consumption.

The developed energy-aware framework was then presented. The player's bid is computed based on the utility function, which takes into account the system overall variation of energy consumption when the task is scheduled to a specific core at selected frequency. In order to compute this variation of energy consumption, two tasks execution maps were created, which are computed based on the power consumption and execution time estimations for each task that is being executed. Based on the difference of the overall energy consumption between the "before" and "after inserting the task on the core" task execution map, it is decided to which core the task must be scheduled and which should be the selected frequency that imposes the minimum variation of energy consumption in the system. It was also developed a way to predict the task instantaneous power consumption and execution time for each available frequency without physically changing it, in order to compute the bids of each task execution map.

Furthermore, the developed framework was then experimentally evaluated on the ARM Juno r2 development platform. It was conducted an evaluation of the task power consumption and execution time predictions, which revealed that both predictions are able to estimate approximated values with associated errors of 15% and 2%, respectively, which reveals that they should be improved in future work

in order to reduce the associated errors. The proposed framework was then evaluated with different benchmark workloads. The conducted evaluation revealed that the proposed framework can achieve energy savings of up to 36%, 32% and 22% when compared with the Linaro's kernel 3.10, GTS and EAS scheduling approaches, respectively.

In summary, the proposed energy-aware game-theoretic scheduling approach is capable of exploiting the performance counters and energy meter registers of the system to gather information about the task in order to characterize it, and hence, by following the game-theoretic concepts, to select the best frequency and core to schedule the task, which corresponds to the minimum variation of the overall energy consumption in the device.

6.1 Future work

In this thesis, there were several issues left open that suggests possible improvements. In order to create an energy-aware game-theoretic scheduling approach, which uses the overall energy consumption as a performance governing metric, it is necessary that the player's bid must be related with energy consumption. However, it is difficult to know the necessary energy consumption to execute a task when it is not known a priori information about the task, as for example the number of total instructions in order to estimate the task execution time. Another seen issue was the lack of task behavior/phase-awareness, which has to be taken into account when some task is scheduled or finishes by re-measuring the current CPI value. Future work can focus on improve task characterizations in order to detect the task phases and reschedule them. To tackle this two issues, it could be assumed that the task will execute in the next predicted short amount of time (number of total instructions not need to be known a priori) and with that it would be possible to estimate the energy consumption of that small portion of the task (which has the same task phase) with lower associated errors.

Further studies should also focus on a better characterization of the task through the performance counters as well as the board's static and dynamic power consumption in order to improve the task power consumption predictions for other frequencies. It would be interesting to have energy-awareness at the level of the instructions and each processing unit (e.g. Integer ALU, Floating point and multiply unit) to then predict more accurately the power consumption for other frequencies. It should be also develop a framework to read directly the performance counter registers through the kernel. This would provide lower overheads than PAPI software, and would let to evaluate the developed scheduler on every Linaro's kernel used on ARM Juno r2 board, in which PAPI has not to be installed.

Furthermore, the proposed user-space scheduling approach functions could be implemented inside the kernel in order to reduce the associated overheads to pause the threads as well as to measure the CPI and instantaneous power consumption values. As the proposed scheduler has been developed, each benchmark is one thread of the scheduler process, and after sending the pause signal, it must wait a fixed time of 10 ms in order for all threads to detect the signal and pause. This wait time can be removed when using the kernel because the benchmarks will be considered as threads in different processes and each one can then be easily paused.

Finally, it can be expected that new architectures will allow frequency scaling and energy meter measures at the level of the core, which could increase the individual energy-awareness for each core and achieve lower energy consumption by scaling just the frequency of the individual core and not at the whole cluster.

References

- [1] Roger B. Myerson. Game Theory: Analysis of Conflict. Harvard University Press, 1nd edition, 1997. ISBN:0-674-34115-5.
- [2] I. Ahmad, S. Ranka, and S.U. Khan. Using game theory for scheduling tasks on multi-core processors for simultaneous optimization of performance and energy. IEEE International Symposium on Parallel and Distributed Processing, pages 1–6, 2008.
- [3] Nickolas Bielik and Ishfaq Ahmad. Cooperative versus non-cooperative game theoretical techniques for Energy Aware Task scheduling. 2012 International Green Computing Conference (IGCC), pages 1–6, 2012.
- [4] Guowei Wu, Zichuan Xu, Qiufen Xia, and Jiankang Ren. An energy-aware multi-core scheduler based on generalized tit-for-tat cooperative Game. Journal of Computers, 7(1):106–115, 2012.
- [5] Muhammad Shafique, Lars Bauer, Waheed Ahmed, and Jörg Henkel. Minority-Game-based Resource Allocation for Run-Time Reconfigurable Multi-Core Processors. pages 1–6, 2011.
- [6] ARM. ARM Versatile Express Juno r2 Development Platform (V2M-Juno r2) Technical Reference Manual, November 2015.
- [7] ARM. ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile, 2013.
- [8] ARM Cortex-A53 Architecture, <http://www.anandtech.com/show/8718/the-samsung-galaxy-note-4-exynos-review/3>, Web accessed: 22 of September of 2016.
- [9] ARM Cortex-A72 CPU, <http://pc.watch.impress.co.jp/docs/column/kaigai/699491.html>, Web accessed: 22 of September of 2016.
- [10] Performance Application Programming Interface (PAPI), <https://icl.cs.utk.edu/projects/papi/wiki/Threads>, Web accessed: 17 of April of 2016.
- [11] OProfile - A System Profiler for Linux, <http://oprofile.sourceforge.net/news/>, Web accessed: 17 of April of 2016.
- [12] Morten Rasmussen. Using task load tracking to improve kernel scheduler load balancing. linux foundation collaboration summit. 2013.
- [13] BKK16-317: How to generate power models for EAS and IPA, <http://connect.linaro.org/resource/bkk16/bkk16-317/>, Web accessed: 15 of September of 2016.

- [14] Francisco Gaspar et al. A Framework for Application Guided Task Management on Heterogeneous Embedded Systems. ACM Transactions on Architecture and Code Optimization (TACO), Vol. 12, Article 42, 2015.
- [15] Thannirmalai Somu Muthukaruppan, Anuj Pathania, and Tulika Mitra. Price Theory Based Power Management for Heterogeneous Multi-Cores. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 161–176, 2014.
- [16] John Nash. In "The Bargaining Problem", volume 18, pages 155–162. *Econometrica*, 1950. JSTOR 1907266.
- [17] Diego Puschini, Fabien Clermidy, C E A Leti Minatec, Pascal Benoit, Gilles Sassatelli, and Lionel Torres. Temperature-Aware Distributed Run-Time Optimization on MP-SoC using Game Theory. pages 375–380, 2008.
- [18] Samee Ullah Khan and Ishfaq Ahmad. A Cooperative Game Theoretical Technique for Joint Optimization of Energy Consumption and Response Time in Computational Grids. 20(3):346–360, 2009.
- [19] Mikhael Shor. Tit for Tat, Dictionary of Game Theory Terms, Game Theory .net, <http://www.gametheory.net/dictionary/TitforTat.html>, Web accessed: 06 of December of 2015.
- [20] P Michaud. (2009, ATMI manual. Available: <http://www.irisa.fr/alf/>.
- [21] Joel Wilkins, Ishfaq Ahmad, Hafiz Fahad Sheikh, Shujaat Faheem Khan, and Saeed Rajput. Optimizing Performance and Energy in Computational Grids using Non- Cooperative Game Theory. 2010.
- [22] Using Linaro's deliverables on Juno, <https://community.arm.com/docs/DOC-10804>, Web accessed: 02 of March of 2016.
- [23] Tutorial: Energy monitoring on the Juno (Revision 6), <https://community.arm.com/docs/DOC-9321>, Web accessed: 28 of September of 2016.
- [24] Christian Bienia. Benchmarking Modern Multiprocessors. PhD thesis, Princeton University, January 2011.
- [25] Standard Performance Evaluation Corporation (SPEC) CPU™ 2006, <https://www.spec.org/cpu2006/>, Web accessed: 3 of August of 2016.
- [26] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. SD-VBS : The San Diego Vision Benchmark Suite.
- [27] Opensource Basic Linear Algebra Subprograms (OpenBLAS), <http://www.openblas.net/>, Web accessed: 21 of April of 2016.

- [28] ARM Cortex-A53 MPCore Processor Technical Reference Manual, <https://developer.arm.com/products/processors/cortex-a/cortex-a53>, Web accessed: 02 of March of 2016.
- [29] ARM Cortex-A72 MPCore Processor Technical Reference Manual, <https://developer.arm.com/products/processors/cortex-a/cortex-a72>, Web accessed: 02 of March of 2016.



PMU events on ARMv8-A architecture

Table A.1: PMU events on ARMv8-A architecture [6].

Event Number	Event mnemonic	Event name
0x00	SW_INCR	Instruction architecturally executed, software increment
0x01	L1I_CACHE_REFILL	Level 1 instruction cache refill
0x02	L1I_TLB_REFILL	Level 1 instruction TLB refill
0x03	L1D_CACHE_REFILL	Level 1 data cache refill
0x04	L1D_CACHE	Level 1 data cache access
0x05	L1D_TLB_REFILL	Level 1 data TLB refill
0x06	LD_RETIRED	Instruction architecturally executed, load
0x07	ST_RETIRED	Instruction architecturally executed, store
0x08	INST_RETIRED	Instruction architecturally executed
0x09	EXC_TAKEN	Exception taken
0x0A	EXC_RETURN	Instruction architecturally executed, exception return
0x0B	CID_WRITE_RETIRED	Instruction architecturally executed, write to CONTEXTIDR
0x0C	PC_WRITE_RETIRED	Instruction architecturally executed, software change of the PC
0x0D	BR_IMMED_RETIRED	Instruction architecturally executed, immediate branch
0x0E	BR_RETURN_RETIRED	Instruction architecturally executed, procedure return
0x0F	UNALIGNED_LDST_RETIRED	Instruction architecturally executed, unaligned load or store
0x10	BR_MIS_PRED	Mispredicted or not predicted branch speculatively executed
0x11	CPU_CYCLES	Cycle
0x12	BR_PRED	Predictable branch speculatively executed
0x13	MEM_ACCESS	Data memory access
0x14	L1I_CACHE	Level 1 instruction cache access
0x15	L1D_CACHE_WB	Level 1 data cache write-back
0x16	L2D_CACHE	Level 2 data cache access
0x17	L2D_CACHE_REFILL	Level 2 data cache refill
0x18	L2D_CACHE_WB	Level 2 data cache write-back
0x19	BUS_ACCESS	Bus access
0x1A	MEMORY_ERROR	Local memory error
0x1B	INST_SPEC Operation	speculatively executed
0x1C	TTBR_WRITE_RETIRED	Instruction architecturally executed, write to TTBR
0x1D	BUS_CYCLES	Bus cycle
0x1E	CHAIN	For odd-numbered counters, increments the count by one for each overflow of the preceding even-numbered counter. For even-numbered counters there is no increment.
0x1F	L1D_CACHE_ALLOCATE	Level 1 data cache allocation without refill
0x20	L2D_CACHE_ALLOCATE	Level 2 data cache allocation without refill

B

Cortex-A53 PMU events

Table B.1: Cortex-A53 PMU events [28].

Event Number	Event mnemonic	Event name
0x60	BUS_ACCESS_LD	Bus access - Read.
0x61	BUS_ACCESS_ST	Bus access - Write.
0x7A	BR.INDIRECT_SPEC	Branch speculatively executed - Indirect branch.
0x86	EXC_IRQ	Exception taken, IRQ.
0x87	EXC_FIQ	Exception taken, FIQ.
0xC0	-	External memory request.
0xC1	-	Non-cacheable external memory request.
0xC2	-	Linefill because of prefetch.
0xC3	-	Instruction Cache Throttle occurred.
0xC4	-	Entering read allocate mode.
0xC5	-	Read allocate mode.
0xC6	-	Pre-decode error.
0xC7	-	Data Write operation that stalls the pipeline because the store buffer is full.
0xC8	-	SCU Snooped data from another CPU for this CPU.
0xC9	-	Conditional branch executed.
0xCA	-	Indirect branch mispredicted.
0xCB	-	Indirect branch mispredicted because of address miscompare.
0xCC	-	Conditional branch mispredicted.
0xD0	-	L1 Instruction Cache (data or tag) memory error.
0xD1	-	L1 Data Cache (data, tag or dirty) memory error, correctable or non-correctable.
0xD2	-	TLB memory error.
0xE0	-	Attributable Performance Impact Event. Counts every cycle that the DPU IQ is empty and that is not because of a recent micro-TLB miss, instruction cache miss or pre-decode error.
0xE1	-	Attributable Performance Impact Event. Counts every cycle the DPU IQ is empty and there is an instruction cache miss being processed.
0xE2	-	Attributable Performance Impact Event. Counts every cycle the DPU IQ is empty and there is an instruction micro-TLB miss being processed.
0xE3	-	Attributable Performance Impact Event. Counts every cycle the DPU IQ is empty and there is a pre-decode error being processed.

Table B.2: Cortex-A53 PMU events (continued) [28].

Event Number	Event mnemonic	Event name
0xE4	-	Attributable Performance Impact Event. Counts every cycle there is an interlock that is not because of an Advanced SIMD or Floating-point instruction, and not because of a load/store instruction waiting for data to calculate the address in the AGU. Stall cycles because of a stall in Wr, typically awaiting load data, are excluded.
0xE5	-	Attributable Performance Impact Event. Counts every cycle there is an interlock that is because of a load/store instruction waiting for data to calculate the address in the AGU. Stall cycles because of a stall in Wr, typically awaiting load data, are excluded.
0xE6	-	Attributable Performance Impact Event. Counts every cycle there is an interlock that is because of an Advanced SIMD or Floating-point instruction. Stall cycles because of a stall in the Wr stage, typically awaiting load data, are excluded.
0xE7	-	Attributable Performance Impact Event Counts every cycle there is a stall in the Wr stage because of a load miss.
0xE8	-	Attributable Performance Impact Event. Counts every cycle there is a stall in the Wr stage because of a store.
-	-	Two instructions architecturally executed. Counts every cycle in which two instructions are architecturally retired. Event 0x08, INST.RETIRED, always counts when this event counts.
-	-	L2 (data or tag) memory error, correctable or non-correctable.
-	-	SCU snoop filter memory error, correctable or non-correctable.
-	-	Advanced SIMD and Floating-point retention active.
-	-	CPU retention active.



Cortex-A72 PMU events

Table C.1: Cortex-A72 PMU events [29].

Event Number	Event mnemonic	Event name
0x40	L1D_CACHE_LD	Level 1 data cache access - Read
0x41	L1D_CACHE_ST	Level 1 data cache access - Write
0x42	L1D_CACHE_REFILL_LD	Level 1 data cache refill - Read
0x43	L1D_CACHE_REFILL_ST	Level 1 data cache refill - Write
0x46	L1D_CACHE_WB_VICTIM	Level 1 data cache Write-back - Victim
0x47	L1D_CACHE_WB_CLEAN	Level 1 data cache Write-back - Cleaning and coherency
0x48	L1D_CACHE_INVALID	Level 1 data cache invalidate
0x4C	L1D_TLB_REFILL_LD	Level 1 data TLB refill - Read
0x4D	L1D_TLB_REFILL_ST	Level 1 data TLB refill - Write
0x50	L2D_CACHE_LD	Level 2 data cache access - Read
0x51	L2D_CACHE_ST	Level 2 data cache access - Write
0x52	L2D_CACHE_REFILL_LD	Level 2 data cache refill - Read
0x53	L2D_CACHE_REFILL_ST	Level 2 data cache refill - Write
0x56	L2D_CACHE_WB_VICTIM	Level 2 data cache Write-back - Victim
0x57	L2D_CACHE_WB_CLEAN	Level 2 data cache Write-back - Cleaning and coherency
0x58	L2D_CACHE_INVALID	Level 2 data cache invalidate
0x60	BUS_ACCESS_LD	Bus access - Read
0x61	BUS_ACCESS_ST	Bus access - Write
0x62	BUS_ACCESS_SHARED	Bus access - Normal
0x63	BUS_ACCESS_NOT_SHARED	Bus access - Not normal
0x64	BUS_ACCESS_NORMAL	Bus access - Normal
0x65	BUS_ACCESS_PERIPH	Bus access - Peripheral
0x66	MEM_ACCESS_LD	Data memory access - Read
0x67	MEM_ACCESS_ST	Data memory access - Write
0x68	UNALIGNED_LD_SPEC	Unaligned access - Read
0x69	UNALIGNED_ST_SPEC	Unaligned access - Write
0x6A	UNALIGNED_LDST_SPEC	Unaligned access
0x6C	LDREX_SPEC	Exclusive operation speculatively executed - LD-REX
0x6D	STREX_PASS_SPEC	Exclusive instruction speculatively executed - STREX pass
0x6E	STREX_FAIL_SPEC	Exclusive operation speculatively executed - STREX fail
0x70	LD_SPEC	Operation speculatively executed - Load
0x71	ST_SPEC	Operation speculatively executed - Store
0x72	LDST_SPEC	Operation speculatively executed - Load or store
0x73	DP_SPEC	Operation speculatively executed - Integer data processing

Table C.2: Cortex-A72 PMU events (continued)[29].

Event Number	Event mnemonic	Event name
0x74	ASE_SPEC	Operation speculatively executed - Advanced SIMD
0x75	VFP_SPEC	Operation speculatively executed - VFP
0x76	PC.WRITE_SPEC	Operation speculatively executed - Software change of the PC
0x77	CRYPTO_SPEC	Operation speculatively executed, crypto data processing
0x78	BR.IMMED_SPEC	Branch speculatively executed - Immediate branch
0x79	BR.RETURN_SPEC	Branch speculatively executed - Procedure return
0x7A	BR.INDIRECT_SPEC	Branch speculatively executed - Indirect branch
0x7C	ISB_SPEC	Barrier speculatively executed - ISB
0x7D	DSB_SPEC	Barrier speculatively executed - DSB
0x7E	DMB_SPEC	Barrier speculatively executed - DMB
0x81	EXC.UNDEF	Exception taken, other synchronous
0x82	EXC.SVC	Exception taken, Supervisor Call
0x83	EXC.PABORT	Exception taken, Instruction Abort
0x84	EXC.DABORT	Exception taken, Data Abort or SError
0x86	EXC.IRQ	Exception taken, IRQ
0x87	EXC.FIQ	Exception taken, FIQ
0x88	EXC.SMC	Exception taken, Secure Monitor Call
0x8A	EXC.HVC	Exception taken, Hypervisor Call
0x8B	EXC.TRAP.PABORT	Exception taken, Instruction Abort not taken locally
0x8C	EXC.TRAP.DABORT	Exception taken, Data Abort, or SError not taken locally
0x8D	EXC.TRAP.OTHER	Exception taken – Other traps not taken locally
0x8E	EXC.TRAP.IRQ	Exception taken, IRQ not taken locally
0x8F	EXC.TRAP.FIQ	Exception taken, FIQ not taken locally
0x90	RC.LD_SPEC	Release consistency instruction speculatively executed – Load-Acquire
0x91	RC.ST_SPEC	Release consistency instruction speculatively executed – Store-Release



Available PAPI events on ARM Juno r2 platform

Table D.1: Available PAPI events on ARM Juno r2 platform.

Event name	Description
BUS_READ_ACCESS	Bus read access
BUS_CYCLES	Bus cycle
** LOCAL_MEMORY_ERROR	Local memory error
BUS_ACCESS	Bus access
L2D_CACHE_WB	Level 2 data cache WriteBack
L2D_CACHE_REFILL	Level 2 data cache refill
L2D_CACHE_ACCESS	Level 2 data cache access
L1D_CACHE_WB	Level 1 data cache WriteBack
L1I_CACHE_ACCESS	Level 1 instruction cache access
DATA_MEM_ACCESS	Data memory access
BRANCH_PRED	Predictable branch speculatively executed
CPU_CYCLES	Cycles
BRANCH_MISPRED	Mispredicted or not predicted branch speculatively executed
** UNALIGNED_LDST_RETIRED	Procedure return, instruction architecturally executed, condition check pass
* BR_IMMED_RETIRED	Software change of the PC, instruction architecturally executed, condition check pass
* PC_WRITE_RETIRED	Write to CONTEXTIDR, instruction architecturally executed, condition check pass
** CID_WRITE_RETIRED	Change to Context ID retired
* EXCEPTION_RETURN	Instruction architecturally executed (condition check pass) Exception return
EXCEPTION_TAKEN	Exception taken
INST_RETIRED	Instruction architecturally executed
* ST_RETIRED	Store Instruction architecturally executed, condition check
* LD_RETIRED	Load Instruction architecturally executed, condition check
L1D_TLB_REFILL	Level 1 data TLB refill
L1D_CACHE_ACCESS	Level 1 data cache access
L1D_CACHE_REFILL	Level 1 data cache refill
L1I_TLB_REFILL	Level 1 instruction TLB refill
L1I_CACHE_REFILL	Level 1 instruction cache refill
** SW_INCR	Instruction architecturally executed (condition check pass) Software increment

Legend:

- * - These events could not be read with UEFI 16.04 Flash (only in LTK kernel).
- ** - These events could not be read on both UEFI and LTK Flashes.

All shown PAPI events could not be available on the LSK kernel because the PAPI could not be installed.



**Description of each successfully
compiled benchmark**

Table E.1: Description of each successfully compiled benchmark.

OpenBLAS	
Saxpy	Vector multiplication. $\mathbf{y} \leftarrow \alpha \mathbf{x} + \mathbf{y}$
Sgemv	Matrix-Vector multiplication. $\mathbf{y} \leftarrow \alpha \mathbf{A}\mathbf{x} + \beta \mathbf{y}$
Sgemm	Matrix-Matrix multiplication. $\mathbf{C} \leftarrow \alpha \mathbf{A}\mathbf{B} + \beta \mathbf{C}$
Sdot	Dot product. $dot \leftarrow \mathbf{x}^T \mathbf{y}$
Sscal	Vector multiplication. $\mathbf{y} \leftarrow \alpha \mathbf{x}$

SPEC CPU2006	
Bzip2	Compression

PARSEC	
Blacksholes	Option pricing with Black-Scholes Partial Differential Equation (PDE)

SD-VBS	
Tracking	Motion, Tracking and Stereo Vision
Texture_synthesis	Image Processing and Formation
Stitch	Image Processing and Formation
Disparity	Motion, Tracking and Stereo Vision
Mser	Image Analysis

F

Benchmark's experimental values

Table F.1: Benchmark's solo values for each frequency.

OpenBLAS						
	Cortex-A72 (big)			Cortex-A53 (LITTLE)		
Saxpy	f_0	f_1	f_2	f_0	f_1	f_2
CPI	1,013	1,012	1,012	1,300	1,301	1,301
Power Cluster [mW]	232,53	443,20	664,03	69,64	130,39	191,47
Power System [mW]	745,24	750,93	768,04	746,04	745,25	742,40
#Total Instructions	1500247299					
	Cortex-A72 (big)			Cortex-A53 (LITTLE)		
Sgemv	f_0	f_1	f_2	f_0	f_1	f_2
CPI	1,011	1,009	1,012	1,301	1,301	1,301
Power Cluster [mW]	240,91	461,73	689,41	70,89	132,20	195,45
Power System [mW]	748,53	747,48	746,19	749,27	750,09	752,00
#Total Instructions	2593101984					
	Cortex-A72 (big)			Cortex-A53 (LITTLE)		
Sgemm	f_0	f_1	f_2	f_0	f_1	f_2
CPI	0,624	0,624	0,624	1,340	1,344	1,343
Power Cluster [mW]	418,16	815,99	1227,80	115,38	228,04	334,56
Power System [mW]	755,18	767,99	768,00	748,11	756,58	757,03
#Total Instructions	2291902269					
	Cortex-A72 (big)			Cortex-A53 (LITTLE)		
Sdot	f_0	f_1	f_2	f_0	f_1	f_2
CPI	1,010	1,011	1,011	1,305	1,305	1,305
Power Cluster [mW]	233,07	442,67	665,59	69,56	129,07	190,53
Power System [mW]	745,25	750,93	716,81	746,04	750,93	750,93
#Total Instructions	1506845772					
	Cortex-A72 (big)			Cortex-A53 (LITTLE)		
Sscal	f_0	f_1	f_2	f_0	f_1	f_2
CPI	1,015	1,015	1,015	1,281	1,281	1,281
Power Cluster [mW]	233,31	443,29	664,98	69,75	130,89	193,09
Power System [mW]	752,48	755,95	760,12	750,08	753,14	753,66
#Total Instructions	3729911229					
SPEC CPU2006						
	Cortex-A72 (big)			Cortex-A53 (LITTLE)		
Bzip2	f_0	f_1	f_2	f_0	f_1	f_2
CPI	0,538	0,539	0,538	1,109	1,109	1,109
Power Cluster [mW]	246,07	469,61	703,15	73,72	138,64	204,00
Power System [mW]	746,06	742,45	742,80	741,30	741,28	736,72
#Total Instructions	2852058201					

Table F.2: Benchmark's solo values for each frequency (continued).

PARSEC						
	Cortex-A72 (big)			Cortex-A53 (LITTLE)		
Blacksholes	f_0	f_1	f_2	f_0	f_1	f_2
CPI	1,206	1,208	1,207	1,515	1,523	1,524
Power Cluster [mW]	220,7	416,2	624,5	83,8	157,1	232,3
Power System [mW]	771,1	771,0	775,3	760,7	762,7	761,3
#Total Instructions	2428090722					
SD-VBS						
	Cortex-A72 (big)			Cortex-A53 (LITTLE)		
Tracking	f_0	f_1	f_2	f_0	f_1	f_2
CPI	0,664	0,667	0,667	1,440	1,439	1,439
Power Cluster [mW]	243,20	462,41	693,87	65,97	121,43	179,58
Power System [mW]	746,64	744,47	750,83	740,76	745,74	743,76
#Total Instructions	2612055004					
	Cortex-A72 (big)			Cortex-A53 (LITTLE)		
Texture_synthesis	f_0	f_1	f_2	f_0	f_1	f_2
CPI	0,690	0,692	0,689	1,413	1,413	1,412
Power Cluster [mW]	244,99	468,32	699,88	67,20	124,20	183,00
Power System [mW]	742,40	742,59	716,20	742,39	745,60	742,39
#Total Instructions	1985513902					
	Cortex-A72 (big)			Cortex-A53 (LITTLE)		
Stitch	f_0	f_1	f_2	f_0	f_1	f_2
CPI	0,644	0,644	0,643	1,355	1,355	1,355
Power Cluster [mW]	241,48	461,33	691,09	67,22	124,66	183,74
Power System [mW]	742,43	746,91	750,81	739,55	743,50	736,72
#Total Instructions	2617650281					
	Cortex-A72 (big)			Cortex-A53 (LITTLE)		
Disparity	f_0	f_1	f_2	f_0	f_1	f_2
CPI	0,898	0,897	0,897	1,337	1,336	1,337
Power Cluster [mW]	226,77	431,96	644,36	73,92	138,69	204,22
Power System [mW]	744,59	747,43	750,80	737,26	738,74	735,43
#Total Instructions	1585389190					
	Cortex-A72 (big)			Cortex-A53 (LITTLE)		
Mser	f_0	f_1	f_2	f_0	f_1	f_2
CPI	0,779	0,780	0,780	1,392	1,392	1,392
Power Cluster [mW]	237,81	452,57	677,91	69,51	129,31	190,72
Power System [mW]	742,43	746,06	747,37	739,61	740,08	745,26
#Total Instructions	2506552802					

Table F.3: Benchmark's solo values for each frequency (continued).

Custom Benchmarks						
	Cortex-A72 (big)			Cortex-A53 (LITTLE)		
CPU_bound	f_0	f_1	f_2	f_0	f_1	f_2
CPI	1,169	1,169	1,169	1,124	1,124	1,124
Power Cluster [mW]	210,68	397,87	593,11	72,17	135,33	200,63
Power System [mW]	748,53	752,54	746,06	742,40	744,37	742,38
#Total Instructions	2225002565					
	Cortex-A72 (big)			Cortex-A53 (LITTLE)		
MEM_bound	f_0	f_1	f_2	f_0	f_1	f_2
CPI	2,686	3,552	4,009	3,793	5,191	5,792
Power Cluster [mW]	228,95	404,40	589,26	78,46	139,37	202,71
Power System [mW]	833,16	844,80	848,46	805,93	819,20	816,36
#Total Instructions	635032500					