Energy- and time-aware scheduling for heterogeneous high-performance embedded systems

Röder, J.P.

Publication date
2023

Document Version
Final published version

Energy- and Time-Aware Scheduling for Heterogeneous High-Performance Embedded Systems

Julius Röder

This thesis discusses the topic of high-performance embedded systems, their widespread usage, and the need to optimize their hardware utilisation concerning energy consumption and time. Throughout the thesis, various techniques are suggested to execute problems on less powerful hardware, thus saving resources and expenses. Alternatively, these techniques can be utilised to execute more complex problems using the same hardware.

First, the measurement setup used throughout the dissertation is detailed, as well as a set of experiments that determine the importance of the sampling rate for accurate energy measurement of high-performance embedded systems. Second, a novel energy model, system model, and energy-aware scheduler are introduced. They outperform existing schedulers. Third, various ranking algorithms are explored and compared. Fourth, a new task model is presented, which divides tasks into multiple phases, allowing for a more fine-grained separation of workloads. Lastly, a ranking-independent Reinforcement-Learning(RL)-based scheduler is presented. While performing similarly to a greedy heuristic, experiments show that including Graph Convolutional Neural Networks in the RL-scheduler improves the final schedules significantly.

Parallel Computing Systems

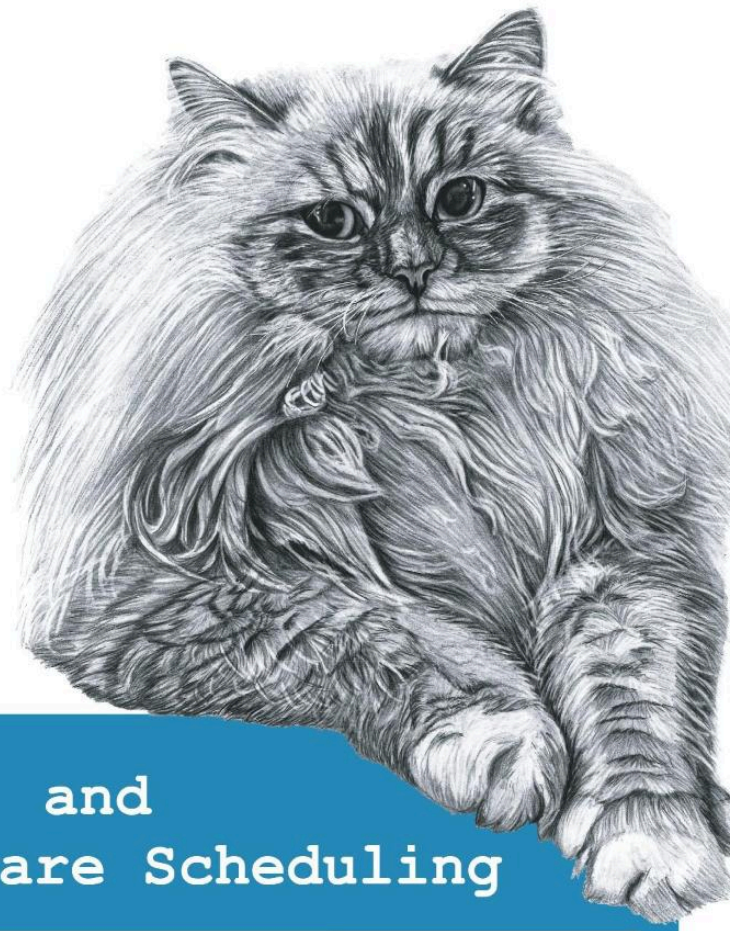
ISBN 978-94-6473-096-8

9 789464 730968

Energy- and Time-Aware Scheduling for Heterogeneous High-Performance Embedded Systems

Julius Röder

University of Amsterdam



Energy- and Time-Aware Scheduling

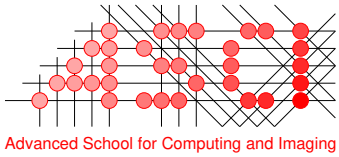
for Heterogeneous High-Performance Embedded Systems

Julius Röder

ENERGY- AND TIME-AWARE SCHEDULING FOR HETEROGENEOUS HIGH-PERFORMANCE EMBEDDED SYSTEMS

JULIUS PHILIPP RÖDER

This work is partially supported by the European Union Horizon-2020 research and innovation programmes TeamPlay (grant agreement No. 779882) and ADMORPH (grant agreement No. 871259). Additionally, this work is supported and partly funded by the HiPEAC project which has received funding from the European Union Horizon-2020 research and innovation programme under grant agreement No. 871174 (HiPEAC6 Network). Lastly, this work is partially supported by CERCIRAS COST Action CA19135 funded by COST Association.



This work was carried out in the ASCI graduate school.
ASCI dissertation series number 444.

Cover portrait by: Nicoletta Maraschin
Cover design by: Quentin Malgaud
Thesis template: classicthesis by André Miede and Ivo Pletikosić.
Printed and bound by Ipskamp printing

ISBN: 978-94-6473-096-8

ENERGY- AND TIME-AWARE SCHEDULING FOR HETEROGENEOUS HIGH-PERFORMANCE EMBEDDED SYSTEMS

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. ir. P.P.C.C. Verbeek
ten overstaan van een door het College voor Promoties
ingestelde commissie,
in het openbaar te verdedigen in de Agnietenkapel
op woensdag 24 mei 2023, te 13.00 uur

door

Julius Philipp Röder

geboren te Bad Kissingen

Promotiecommissie

Promotor:	prof. dr. A.D. Pimentel	Universiteit van Amsterdam
Copromotores:	dr. C.U. Grelck	Universiteit van Amsterdam
	prof. dr. S.J. Altmeyer	Universität Augsburg
Overige leden:	prof. dr. ir. C.T.A.M. de Laat	Universiteit van Amsterdam
	prof. dr. K.B. Akesson	Universiteit van Amsterdam
	dr. P. Grosso	Universiteit van Amsterdam
	dr. A.M. Oprescu	Universiteit van Amsterdam
	dr. M. Völp	University of Luxembourg
	prof. dr. U.P. Schultz	University of Southern Denmark
	prof. dr. K.I. Eder	University of Bristol

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

Dedicated to Maggie and Baloo.

ACKNOWLEDGMENTS

I am deeply grateful to all those who have supported me throughout my PhD journey. First, I would like to thank my partner Helena for her patience, support and love during the ups and downs of the PhD. Clemens, my supervisor, for taking the time for all the meetings we had, tirelessly reading my drafts, and for all the enjoyable conversations we had during our various travels. Ben, thank you for your help with the day-to-day challenges of the PhD and for being a entertaining friend during our travels and time off. Lukas, thank you for our lively and seemingly endless discussions that somehow always got a "little" sidetracked. Sebastian, thank you for continuing our collaboration after leaving the UvA, hosting me in Augsburg, and providing feedback for both my papers and my dissertation. Andy, thank you for taking the time to read my thesis and providing invaluable feedback. I am grateful to the committee members for reading my thesis and taking part in my PhD examination. At this point, I'd also like to express my gratitude to Lukas and Marco for translating my summary.

I'd like to thank all my friends and colleagues: Mary, Joe, Ralph, Reggie, Spiros, Giulio, Misha, Tim, Taha, Lukasz, Jelle, Marco, Marius, Hongyun, Dolly, Xin, Leonardo, Daphnee, Peter, Pooya, Jun, Yang, Huan. I'd like to thank my friends: Alex, "The J's", Nikita, and Ky. Quentin, Nancy, Steve, Fanny, Kate, Chris, Charlotte, thank you all for asking how the PhD is going, it meant a lot to me. Quentin thanks for the awesome cover!

I'd like to thank all the other people in the SNE cluster. Next, I'd like to thank all people involved in the TeamPlay project - you made the project enjoyable and informative. I'd also like to thank everyone who made me feel welcome during my 3 months in Augsburg: Tilmann, Florian, Christoph, Christian, Martin, Alexander. Also special thanks to my roommates Markus and Pia!

I would like to thank my family for their continuous support throughout the PhD. And last, I would like to thank our cats, Maggie (the star of this cover) for always being a little bit more grumpy than me and Baloo for always purring for absolutely no reason.

To those with whom I have crossed paths in the last 4.5 years but didn't mention by name, thank you from the bottom of my heart.

CONTENTS

1 INTRODUCTION

Single-board computers that use heterogeneous Multiprocessor System-on-Chip (MPSoC) have become widely available in the last decade. The hardware of such single-board computers is similar to modern smartphones and offers relatively high performance compared to the price. Relevant single-board computers are, therefore, also called high-performance embedded systems. As such, it is unsurprising that more and more applications target the heterogeneous MPSoC found on high-performance embedded systems. An MPSoC usually contains multiple different processing elements (e.g., big.LITTLE [1] and an accelerator). Executing an application on an MPSoC involves deciding how the application should use the CPU and other available compute units (e.g., GPU, FPGA).

Deciding how to use the available resources is known as *scheduling*. In general, scheduling is the process of assigning tasks (e.g., a task in a machine shop) to the resources required (and available) for the task (e.g., a specific machine). Scheduling can be done manually or automatically. In the following paragraphs, we explore the growing importance of scheduling on a historical example: the automation of elevators. This example shows how the applications and available features of elevators evolved over time and why scheduling, in general is becoming ever more critical.

Let us consider the evolution and automation of elevators. The first electric elevator was invented in the 1890s and was operated by a human [99]. The first step towards automating elevators was the introduction of the self-levelling elevator. The operator still controlled the elevator's speed, but releasing the controls would let the elevator stop at the next level. Before this invention, the operator had to stop at precisely the right spot [28, 42]. The advancement of elevators continued with the introduction of relay logic and micro-controller-based controls which could execute more sophisticated control logic/algorithms. Thus, elevator control systems started to consider different factors such as daily traffic patterns [66]. Despite this development, manual elevators were still widely used during the first half of the 20th century due to reluctant passengers. The elevator operator strikes contributed to the broader adoption of automatic elevators [62].

The controller of the first fully automatic elevators was based on relay logic to control the speed, position and doors. The first automatic elevators did not do any peak-time passenger prediction or round-time calculations, but they worked. One could press a button, and the elevator went to the correct floor, and the door opened at the right spot. The relay logic required was very reliable, but compared to micro-controllers, relay logic was large, needed a lot of energy and was maintenance heavy (a large number of moving parts). Additionally, the relay-controller size increased exponentially with the number of elevators and floors. Lastly, the control logic was static. Thus, one could not simply change the scheduling without rewiring the logic [128]. Therefore, elevator manufacturers had to either take the very costly step to produce unique logic for each circumstance or provide the same logic no matter the use case of each elevator.

Today's applications are vastly more complex from a computational point of view than making an elevator stop at the right floor. Let us continue with the elevator example. Modern elevator systems for tall buildings have several elevator cabins working together. Additionally, different cabins may target different floors, some even being significantly faster than others to reach the top of tall buildings quicker [4, 99]. The first versions of these multi-cabin systems functioned in a static manner. For example, in the morning in an office building, all elevators return to the ground floor immediately after completing their task instead of idling on the current floor.

As the available hardware (e.g., CPUs, GPUs) advances, so do multi-elevator systems. One example is that elevator management, and scheduling is moving from a statically offline-computed schedule (e.g., based on Monte Carlo simulation [5, 127]) to more dynamic approaches based on live passenger numbers (e.g., [18, 134]). The dynamic multi-elevator scheduler by Bapin et al. [18] uses the surveillance cameras in hallways in front of elevators and inside the cabins to count passengers and the number of waiting people based on Deep Learning (DL). This information is then used in the elevator scheduling algorithm to reduce waiting and travel time. The new scheduler was tested on a 10-storey office building with five elevators. The authors show that their new scheduler reduces travel time by up to 40%. Thus, demonstrating the importance of scheduling in the real world.

Using more complex data, such as images or voice information in connection with deep learning to automate tasks can be observed across a wide range of application areas, ranging from identifying poachers in wildlife preserves [74], over identifying health issues in remote areas without access to good health care [9, 35, 92] to tracking cars in urban areas [45] and fruit

harvesting [21, 23]. All these examples have one thing in common: the applications must be executed locally because the context of the applications often imposes some restriction on the computer that can be made available or on the data that can be sent.

Take, for example, a wildlife camera that identifies humans in areas where they don't belong (i.e., most likely poachers) [74]. On the one hand, the camera has limited bandwidth (communication via LoRaWAN at 27kbps [2]), and on the other hand, it has a limited energy and space budget. The energy budget for such a camera system is limited as it is battery-powered (and possibly recharged with solar panels). While the amount of space is not strictly fixed, a large computer would make deploying and hiding the camera trap significantly more challenging. Slowly uploading pictures to a remote server where they are analysed is not an option, as poachers must be identified quickly to notify authorities before more endangered animals are killed. However, we cannot deploy the camera with a workstation desktop either. We need powerful yet space-, thermal-, and energy-efficient computers. Health care [9, 35, 92] and public sector [15, 45] applications may be more concerned with privacy and security-related issues, thus, prefer to avoid sending sensitive data.

In conclusion, we need a lot of compute power locally. We need to use high-performance embedded systems efficiently. Hence, this thesis explores algorithms for scheduling heavy computational tasks onto different compute units (e.g., CPU, GPU etc.) available on heterogeneous MPSoC.

1.1 HIGH-PERFORMANCE EMBEDDED SYSTEMS

What exactly are high-performance embedded systems? Throughout this thesis, we use the Odroid-XU4 [59], depicted in Figure 1.1, as a good example of a high-performance embedded system. The Odroid-XU4 is a single-board Commercially-Off-The-Shelf (COTS) computer with a wide range of built-in features. It is powered by a Samsung Exynos 5 Octa 5422 [49] MPSoC with eight cores. The CPU cores are split between two core type clusters.

Figure 1.1: Odroid-XU4

The first cluster consists of power-efficient *LITTLE* cores. And the second cluster contains high-performance *big* cores [1]. Additionally, it has an onboard Mali GPU for graphics-heavy applications or acceleration via OpenCL. The Odroid-XU4 is a typical representative of a high-performance embedded system employing a heterogeneous MPSoC.

An absolute definition of high-performance embedded systems that will hold forever is impossible. As "high-performance" is a function of time and shifts as technology advances. In general, high-performance embedded systems typically can run a complete Linux operating system or a real-time patched Linux [114]. Strictly speaking *MPSoC* refers to only a single component of a high-performance embedded system. However, as the MPSoC is often the primary defining characteristic of a given high-performance embedded system, we will use the two terms *MPSoC* and *high-performance embedded system* interchangeably. Additionally, the MPSoC that we consider have multiple CPU cores and onboard accelerators such as GPUs and offer a lot of compute power at low energy consumption, small size and a relatively low cost. Thus, they are well suited to perform computationally heavy tasks in scenarios that are limited by energy, size and cost.

High-performance embedded systems are in stark contrast to more traditional embedded systems such as the micro-controllers from the ARM Cortex-M series [1]. Such microcontrollers are often programmed bare-metal (i.e., no operating system) or with a real-time operating system, but they cannot run a fully-fledged Linux. Additionally, they often even lack important instructions, such as half-, single-, and double-precision floating-point instructions (most of the ARM Cortex-M series), required for compute-heavy tasks. Thus, we define an MPSoC as *single board computers with multiple-CPU cores, possibly of different types, with at least one onboard accelerator such as a GPU.* Furthermore, we consider the area of *embedded systems* to encompass real-time systems (RTS), Internet of things (IoT), cyber-physical systems (CPS) and edge computing.

MPSoC systems can be found from various manufacturers; examples range from systems very similar to the Oroid-XU4, such as the Raspberry Pi [112], over the Nvidia Jetson lineup with powerful GPU's [101], all the way to X86 powered boards such as the Latte Panda 3 [85].

MPSoC offer an excellent energy-to-performance ratio but are more challenging to program than simple single-core CPU based embedded systems. One must engineer concurrent applications to take advantage of the theoretical performance improvements. Let us assume that applications can

1 https://developer.arm.com/ip-products/processors/cortex-m/

be broken down into different parts (i.e., different tasks) for concurrent execution. Engineers must then decide which task is executed where and when (i.e., on which CPU core). First, tasks behave differently depending on the core type (e.g., ARM big.LITTLE [1]) chosen for execution. Furthermore, engineers must consider that different tasks may impact/interfere each other if executed concurrently. Second, the design space is further increased by considering onboard GPUs or other accelerators (e.g., Field-Programmable Gate Array (FPGA)). Again, an engineer must decide which task should take advantage of the accelerator. This decision may also vary depending on whether energy consumption, time or some other extra-functional property is most important. The combination of different target CPU cores, interferences, available accelerators etc., creates an ample design space and makes it difficult for engineers to take full advantage of the available hardware.

Achieving high system utilisation when using MPSoC is essential for several reasons. First, better utilising our computer systems means that companies can choose to use systems with lower specifications. This, in turn, means saving resources (e.g., less silicon) and money in large volume products.

Second, the overall energy consumption of the ICT sector is predicted to increase by 60% between 2022 and 2030 [11]. Reducing the energy consumption of a single embedded device may not have the same impact as decreasing the energy consumption of a whole data centre. However, the number of embedded devices is growing rapidly. For example, the number of active IoT (a sub-category of embedded systems) endpoint connections is expected to grow by 18% to 14.4 billion devices in 2022. By 2025 the number of devices is expected to almost double to a total of 27 billion connected Industrial Internet of things (IIoT) devices [10]. If we do not manage to decrease the energy consumption of the sector, it might become too expensive to maintain the exponential growth of devices. In light of climate change and the overall CO_2 footprint of the IIoT sector, as well as the cost of electricity, reducing the energy consumption of all devices, is crucial.

Third, individual applications may enormously benefit from, for example, improved energy efficiency. If one decreases the energy consumption of computers used for human body detection in Unmanned Aerial Vehicles (UAV) during search and rescue missions [110], then the UAV could fly longer and cover larger patrol areas. In the case of the cameras identifying poachers [74], increasing battery life could, for example, lower the maintenance cost.

In summary, the difficulty of programming high-performance embedded systems, in combination with the need to improve their efficiency and utilisation, leads to our overarching research question:

How can we further automate and simplify the process of improving the efficiency and utilisation of heterogeneous high-performance embedded systems?

1.2 IMPROVING EFFICIENCY AND UTILISATION OF MPSOC

Let us start with the first primary goal of this thesis - reducing energy consumption. To improve energy efficiency, we must accurately measure energy consumption. While researching various energy-measurement methods and setups, we noticed that both researchers and reviewers of accepted papers do not seem to value the exact details of energy-measurement setups. Many authors (e.g., [16, 56, 75, 89, 108, 142]) do not give exact details on how they measure energy consumption. Thus, the absence of these details is seemingly not viewed as important enough to influence the review process. Authors who give information on the setup (e.g., [76]) do not seem to consider their setup's impact on their experiments' accuracy. One variable that might impact the accuracy of a measurement setup is the sampling rate at which we measure power in Watt. Power is a continuous signal and, thus, needs to be measured sufficiently frequently to obtain an accurate estimate of the actual power trace. Hence, the power trace is inaccurate if the sampling rate is too low. An inaccurate power trace also leads to an erroneous energy consumption estimate which is the area under the power trace (i.e., integration of the power trace). This leads us to our first research question:

- *RQ 1: What is the impact of the sampling frequency on energy measurement accuracy?*

In Chapter 3, we aim to answer this question by analysing a set of 42100 power traces. Each power trace is measured at a high sampling rate (4kHz). Then, we show the error resulting from lower sampling rates. Furthermore, we investigate the minimum required sampling rate that can be considered equivalent to the original power trace (equivalence testing).

After investigating the impact of the sampling rate, we continue with our original goal to improve the energy efficiency of MPSoC. We start with the premise that the applications of interest can be represented as Directed Acyclic Graphs (DAG). In a DAG, nodes represent tasks in an application, and edges represent data dependencies (for a more detailed discussion, see Section 2.2). These tasks in the application must be executed on an appropriate compute unit. Thus, we must decide when and where a task is executed or, in other words, how it is scheduled and mapped. Different tasks

have inherently different properties; e.g., some tasks might be more energy efficient on one type of compute unit (CU) (e.g., CPU, GPU) and others on another type. One way to improve (e.g., lower the energy consumption) the execution of a DAG application is, thus, the scheduling of the different tasks. This leads us to our second research question:

- *RQ 2: How can we improve the energy consumption of DAG applications for heterogeneous high-performance embedded systems through scheduling?*

In Chapter 4, we propose a new system model, a fine-grained energy model and a new energy-aware scheduler. Our new system model uses multi-version tasks, which have equivalent functional behaviour (i.e., identical input yields equivalent output), but different non-functional behaviour (e.g., run-time, energy consumption). Next, we propose a new energy model that supports Dynamic Voltage and Frequency Scaling (DVFS) [86, 121] as well as multiple voltage islands [69, 83]. DVFS is a technique used to adjust the energy consumption and compute power of compute units. Voltage islands cluster multiple compute units together, and all CU in an island operate with the same voltage settings. Last, we present a new scheduling algorithm called energy Forward List Scheduling (eFLS), which can fully use the new system and energy models. We show that our approach produces schedules that are more energy-efficient than schedules produced by the two existing state-of-the-art scheduling algorithms: Heterogeneous Earliest Finish Time (HEFT) [131] and ARSH-FATI [133]. Additionally, we compare the solutions derived by our heuristic against optimal solutions derived by an Integer Linear Programming (ILP) formulation to demonstrate that our heuristic does not produce solutions that are significantly worse than the optimal solutions. We make this comparison for sufficiently small problems as the ILP approach does not scale well to large problems.

Scheduling algorithms, such as eFLS, HEFT and ARSH-FATI, need a total order of tasks. A DAG provides a partial order. Thus, the first step in many scheduling algorithms is to rank the tasks before scheduling them. This leads us to our third research question:

- *RQ 3: What is the importance of the ranking algorithm used in our energy-aware scheduling approach?*

In Chapter 4, we show that the ranking used can significantly impact the final application's performance. Additionally, we introduce our new Heterogeneous Energy-aware Ranking (HER) algorithm and compare its performance against a set of base ranking algorithms.

To utilise not only part of an MPSoC but the entire chip, we do not only need to use the different CPU cores but also the onboard accelerators. GPU and other accelerator workloads often need to be launched from the CPU. Thus, in Chapter 4, we considered tasks that require the GPU to occupy both the CPU and the GPU for the entire execution time. This results in periods where a CPU core and the GPU are considered busy, but in fact, one idles. This leads to our fourth research question:

- *RQ 4: How can we increase the hardware utilisation of heterogeneous high-performance embedded systems?*

In Chapter 5, we extend our methodology to split GPU (or other accelerator tasks) into multiple phases. This allows a more fine-grained scheduling approach and reflects the use of resources in a more realistic manner. Therefore, we introduce our heterogeneous Forward List Scheduling (hFLS) algorithm and show how it improves hardware utilisation over state-of-the-art schedulers such as HEFT.

The work in Chapters 4 and 5 heavily relies on the initial ranking of tasks. Hence, we set out to explore ranking-independent scheduling strategies. Additionally, throughout the work for this thesis, we noticed that the state space for modern scheduling scenarios is becoming larger and larger (e.g., DVFS, accelerators, voltage islands, multiple-phases, thermal concerns, environmental factors that change application behaviour dynamically etc.). ILP, Genetic Algorithm (GA) and Evolutionary Algorithm (EA) schedulers are only of limited use as they do not scale well concerning either scheduling time or the performance of the solution. This leads us to investigate Reinforcement Learning (RL) for offline scheduling. RL agents receive the current state of the environment and a set of potential actions. From this, the agent calculates the value of each action and then the most valuable action is passed to the environment. The environment carries out the action and returns a reward based on which the agent can learn. Thus, RL agents make decisions sequentially based on the latest information. Additionally, a RL approach can evaluate all possible actions in one pass. Therefore, our final research question is:

- *RQ 5: To what extent can we use reinforcement learning to replace traditional scheduling methods?*

In Chapter 6, we show the advantages and disadvantages of a multi-core capable RL based scheduler that is ranking independent. We compare the RL scheduler for DAG applications against a traditional greedy heuristic.

We show that our RL approach can produce schedules that are up to 11% shorter than schedules generated by a greedy heuristic. Additionally, we show that across a large dataset the RL generated schedules and greedy heuristic generated schedules perform similarly. The RL generated schedules are on average 2.8% longer.

1.3 THESIS ORGANISATION

Figure 1.2 lays out the organisation of the thesis and how the individual chapters are connected. First, we cover background information in Chapter 2. Chapter 3 focuses explicitly on the impact of sampling frequency on measurement accuracy. Our findings for Chapter 3 led to the experimental setup used in Chapter 4. In Chapter 4, we introduce a new task model designed especially for heterogeneous DVFS-enabled MPSoC. Furthermore, we present a fine-grained energy model and a new scheduler called eFLS which can take advantage of DVFS and supports multiple voltage islands. Lastly, we investigate the impact of different ranking strategies on the performance of eFLS and introduce and compare HER to established ranking strategies. In Chapter 5, we introduce a new task model and scheduler (hFLS) that divides tasks into separate phases. Finally, in Chapter 6, we investigate an RL approach for scheduling dependent tasks onto a homogeneous multi-core system without an accelerator. Lastly, in Chapter 7, we draw our conclusions, answer the research questions and look at future work.

1.4 PUBLICATIONS AND AUTHOR CONTRIBUTIONS

A complete list of publications by the author can be found on Page 135. Additionally, on Page 137, the links to the published code and datasets can be found. Below is a list of the authors' contributions to each paper and how each paper relates to the research chapters.

Ch.3 **J. Roeder**, S. Altmeyer, and C. Grelck *"Can we trust our energy measurements? A study on the Odroid-XU4"* [144], in *15th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT at ECRTS 2022)*.
J.R. designed the experimental setup, collected the data, carried out the statistical analysis, wrote the paper and did the final editing.

Ch.4 **J. Roeder**, B. Rouxel, S. Altmeyer, and C. Grelck *"Interdependent Multi-version Scheduling in Heterogeneous Energy-aware Embedded Systems"* [147], in *13th Junior Researcher Workshop on Real-Time Computing (JRWRTC at RTNS 2019)*.
J.R. defined the system model, came up with the ILP formulation, wrote the paper and did the final editing.

J. Roeder, B. Rouxel, S. Altmeyer, and C. Grelck *"Energy-aware scheduling of multi-version tasks on heterogeneous real-time systems"* [148], in *Proceedings of the 36th Annual ACM Symposium on Applied Computing (SAC 2021)*.
J.R. defined the system model and energy model. Furthermore, J.R. came up with the scheduling heuristic and the ILP formulation. Lastly, J.R. designed the experimental setup, collected the data, carried out the statistical analysis, wrote the paper and did the final editing.

J. Roeder, B. Rouxel, S. Altmeyer, and C. Grelck *"Scheduling multi-version tasks on heterogeneous IoT systems using energy-aware ranking"* [149], in *Under review*.
This paper is a Journal paper extension of [148]. As such, J.R. invented the additional algorithms, carried out the required experiments and analysis, extended the paper and did the final editing.

Ch.5 **J. Roeder**, B. Rouxel, and C. Grelck *"Scheduling DAGs of Multi-version Multi-phase Tasks on Heterogeneous Real-time Systems"* [151], in *14th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC 2021), Singapore. IEEE*.
J.R. defined the system model and created the scheduling heuristic and the ILP formulations. J.R. designed the experimental setup, collected the data, carried out the analysis, wrote the paper and did the final editing.

Ch.6 **J. Roeder**, B. Rouxel, and C. Grelck *"Q-learning for Statically Scheduling DAGs"* [150], in *2020 IEEE International Conference on Big Data (Big Data)*.
J.R. defined the system model, created the Reinforcement Learning based scheduler, designed the experimental setup, collected the data, carried out the analysis, wrote the paper and did the final editing.

J. Roeder, A.D. Pimentel, and C. Grelck *"GCN-based reinforcement learning approach for scheduling DAG applications"* [146], in *19th Artificial Intelligence Applications and Innovations (AIAI 2023)*.
This paper is an extension of [150]. As such, J.R. invented the additional algorithms, carried out the required experiments and analysis, extended the paper and did the final editing.

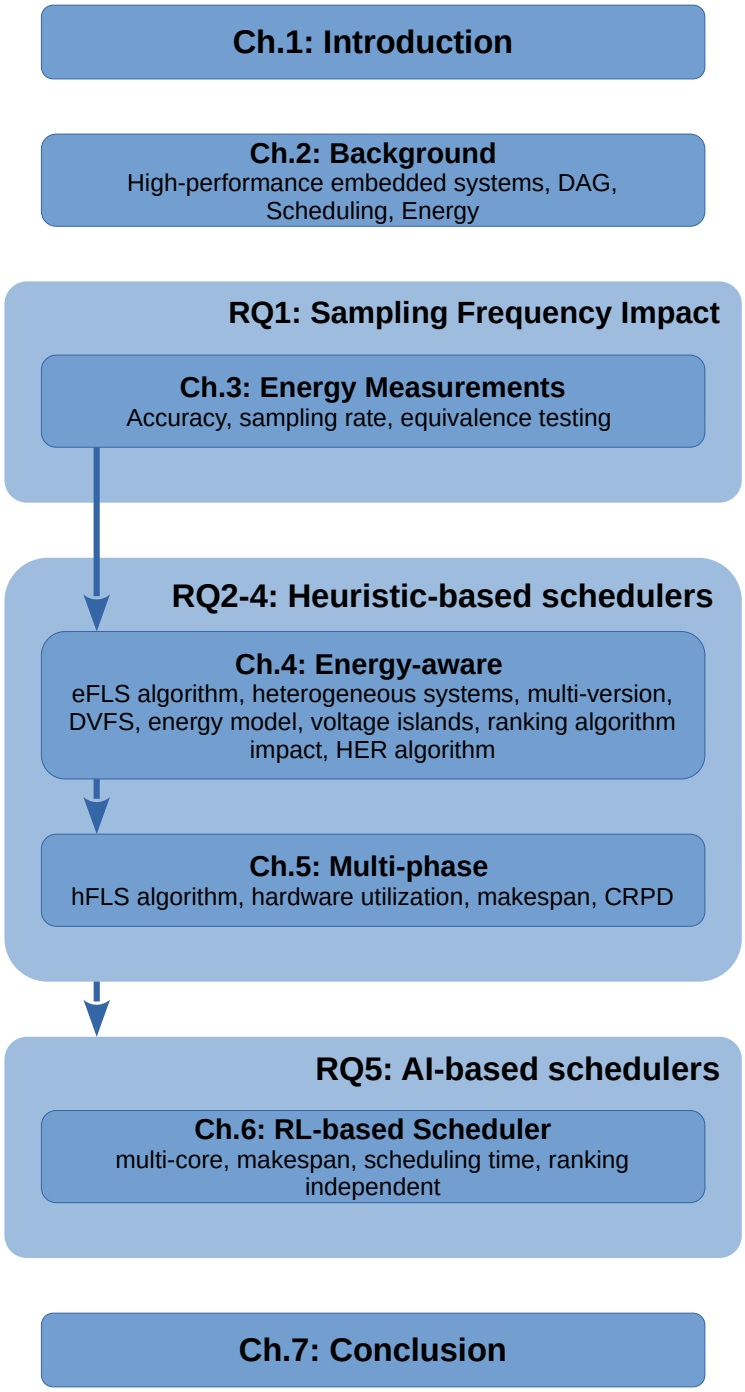


Figure 1.2: Thesis organisation, including chapters, keywords and research questions

2 BACKGROUND

In this Chapter, we discuss and explain core ideas relevant to the rest of the thesis. First, in Section 2.1, we explore this work's central example target system and its features, such as DVFS. Then we give an overview of the system model used (Section 2.2). Last, we provide an overview of scheduling principles and explain how a general Forward List Scheduling (FLS) scheduler works (Section 2.3).

2.1 HIGH-PERFORMANCE EMBEDDED SYSTEMS: ODROID-XU4

Throughout this work, we use the Odroid-XU4 [59] as an example of an MPSoC. The Odorid-XU4 is a single board computer by Hardkernel [1]. It is in many ways representative of other COTS single-board computers such as the Raspberry Pi platform. Moreover, the Odroid-XU4 is also similar to many modern smartphones, which come with similar System-on-Chip (SoC). The main SoC powering the Odroid-XU4 is the Samsung Exynos 5 Octa 5422 [49]. It is an octa-core CPU with four *big* out-of-order cores for performance and four in-order energy-efficient *LITTLE* cores. Additionally, it contains a Mali-based GPU. Figure 2.1 shows the main parts of the Odroid-XU4 in a block-diagram. Table 2.1 lists MPSoC boards similar to the Odroid-XU4.

2.1.1 *DVFS and Voltage Islands*

Changing voltage and frequency dynamically (i.e., DVFS) is a widely used technique to reduce compute units' power and energy consumption. It is used in laptops, desktops, servers, phones etc. For example, the CPU runs at the lowest clock frequency for most of the time and only increases the frequency when additional computational power is needed. The OS automatically does this type of DVFS. The DVFS is almost a continuous variable on some systems. On the Odroid-XU4, DVFS can be done in discrete steps. The DVFS settings for the *LITTLE* cores range from 200MHz to 1500MHz at 100MHz intervals, thus, 14 steps. The range for the *big* cores is slightly larger ranging from 200MHz to 2000MHz at 100MHz steps, resulting in 19

1 https://www.hardkernel.com/

Table 2.1: Examples of systems that we consider to be MPSoC.

Name	CPU	Accelerator	Memory
Latte Panda 3	4x Intel Celeron N5105	Intel UHD Graphics	8GB
Nvidia Jetson Nano	4x ARM Cortex-A57	128-core NVIDIA Maxwell	4GB
Nvidia Jetson TX2	2x NVIDIA Denver 2 cores, 4x ARM Cortex-A57	256-core NVIDIA Pascal	4GB
Odroid-N2+	4x ARM Cortex-A73, 2x ARM Cortex-A53	Mali-G52	2-4GB
Odroid-XU4	4x ARM Cortex-A15, 4x ARM Cortex-A7	Mali-T628	2GB
Raspberry Pi 4	4x ARM Cortex-A72	Broadcom VideoCore IV	1-8GB

possible frequencies. The on-chip Mali-GPU also allows for DVFS ranging from 177MHz to 600MHz in 7 steps (177MHz, 266MHz, 350MHz, 420MHz, 480MHz, 543MHz, 600MHz).

Voltage, frequency, power consumption, and compute power are all connected. A higher (clock) frequency means we can do more operations per second; thus, increasing the frequency increases the compute power. However, increasing the frequency comes at a cost. The gates must be able to keep up with the clock. Forcing gates to switch faster requires a higher voltage, increasing power consumption.

The highest DVFS setting results in the highest performance. However, this results in increased power and energy consumption and high temperatures. Many systems follow a *race-to-idle* philosophy where one uses the highest clock frequency during workload and then the lowest clock frequency when idle [6, 51, 81]. A more nuanced DVFS approach can help achieve the best energy efficiency. To achieve higher speeds, one has to increase the voltage level significantly at the upper end of the DVFS scale, which only results in diminishing marginal returns. The decrease in execution time is not as significant as the required voltage increase. This results in convex energy curves as shown in Figure 2.2. Furthermore, one cannot just use the best

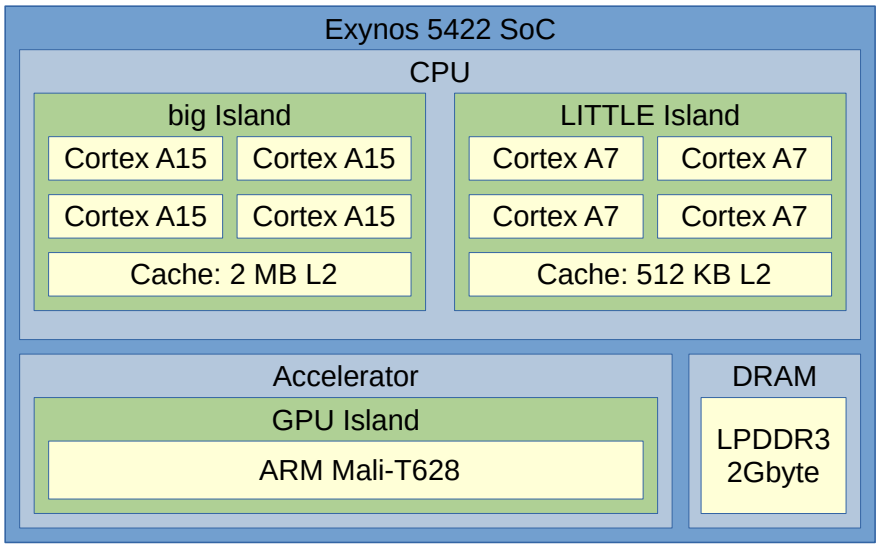


Figure 2.1: Block diagram of the Exynos 5 Oca 5422 showing the different hardware components, including the voltage islands.

DVFS settings per core type because different tasks have different sweet spots even on the same core type, clearly shown in Figure 2.2.

In modern SoC, cores of the same type are often clustered into *voltage islands*. This means we cannot determine the DVFS settings per core but only per island. The Odroid-XU4 has three voltage islands, shown in Figure 2.1 as the dark green boxes. The *LITTLE* cores form one island, the *big* cores form one island and the *Mali-GPU* the last island. Thus, all *LITTLE* cores have the same DVFS settings, and the same is true for the other islands. Clustering different sections of SoC into islands comes with the obvious disadvantage that if we have one busy core at a high DVFS setting, the other idle cores use more energy than under the lowest DVFS setting. However, clustering cores/sections of silicon into islands is advantageous as it uses less silicon than dedicating one island to every core [69].

2.2 TASK MODEL OVERVIEW

This section gives an overview of the base task model used throughout this work. We work with applications represented as Directed Acyclic Graphs(DAG), hereafter also called task graphs. An example DAG is shown in Figure 2.3. In a task graph $G = (\tau, E)$ the set of nodes/vertices τ represents the tasks, and the set of edges E represents data dependencies between tasks,

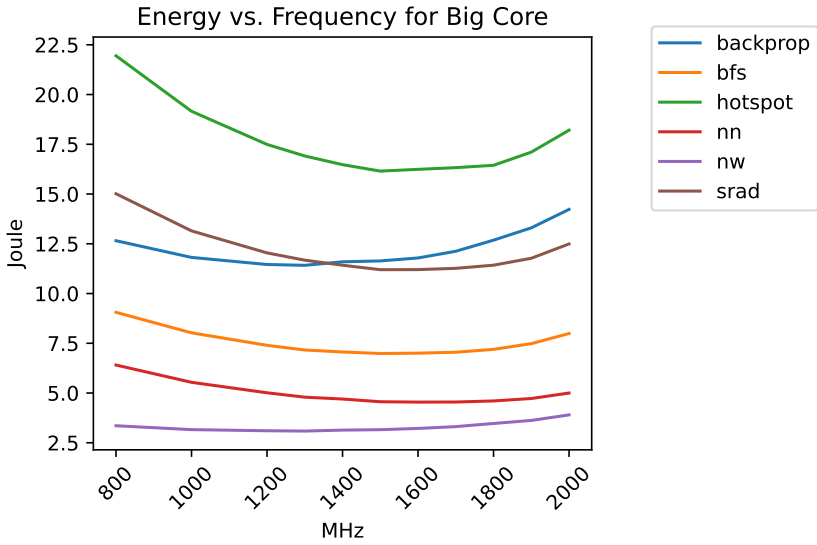


Figure 2.2: Energy consumption required to execute different Rodinia benchmarks [27] on the *big* core of the Odroid-XU4 at different clock frequencies.

i.e., a producer task needs to be completed before the corresponding consumer task may start executing. A *task* can be anything that the user wants to run on the target system, ranging from simple tasks such as outputting to a file to complex computations. A *task* can take any number of inputs and generate any number of outputs. Additionally, a DAG can have a deadline.

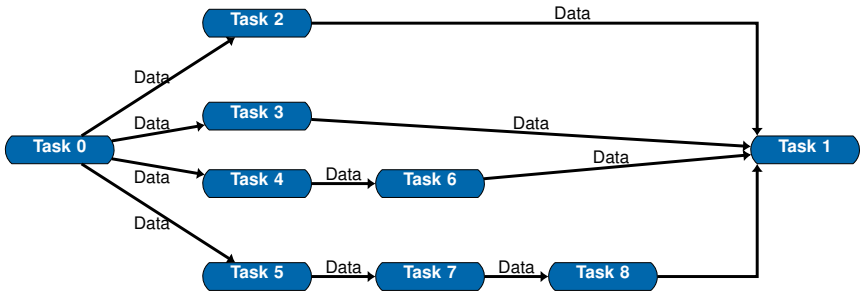


Figure 2.3: Example DAG with nine tasks.

2.3 SCHEDULING

In this section, we give an overview of *scheduling* for DAG and, based on a small example, demonstrate how a simple Forward List Scheduling(FLS) would work. Let us first start by defining mapping and scheduling. Mapping and scheduling go hand in hand. Mapping is the process of determining the core on which a task will be executed. Scheduling is the process of determining when a task will be executed. Mapping and scheduling can be done offline (i.e., statically before execution) or online (i.e., dynamically at runtime). In this thesis, mapping and scheduling are done statically. Additionally, both are done simultaneously and depend on each other, i.e., the time at which a task is executed heavily depends on the target core. From here onwards, *scheduling* will refer to scheduling both in time and space.

Furthermore, according to the taxonomy proposed by Davis and Burns [38], our approaches can also be classified as partitioned, time-triggered and non-preemptive. Partitioned means that a task may not migrate between processors. Time-triggered means that our tasks start executing after a specific amount of time (in contrast to event-triggered tasks). And non-preemptive means that our tasks may not be preempted while being executed.

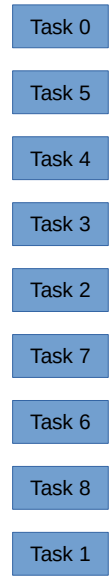
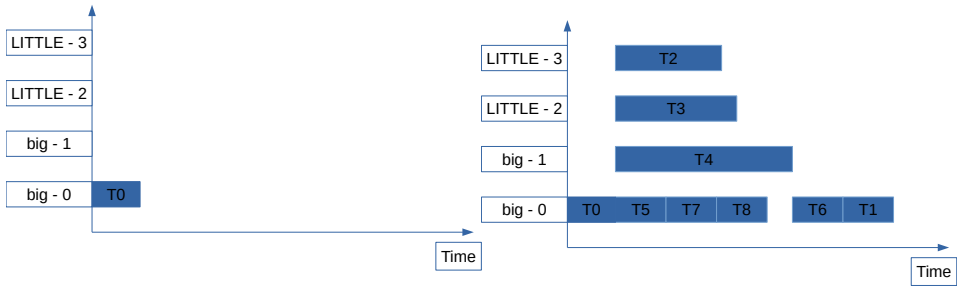


Figure 2.4: Example ranking of the DAG shown in Figure 2.3.

Let us continue with an example application. The application is represented by the DAG shown in Figure 2.3. The DAG provides a partial order of tasks. In this example, Tasks 2 to 5 will be executed after Task 0 as they must wait for the data provided by Task 0. However, we cannot definitively say that Task 2 must be executed before Task 3, as there is no dependency between the tasks. However, many scheduling algorithms (e.g., FLS, ARSH-FATI [133]) require a total order of tasks. Thus, before scheduling, we must rank the tasks. Figure 2.4 shows an example of a total order for the example DAG in Figure 2.3.

Different ranking algorithms can be used, such as Breadth-First or Depth-First. Furthermore, if two tasks have the same "rank", certain properties such as the Worst Case Execution Time (WCET) of the tasks can be used as a tiebreaker.

After ranking, we can use a scheduler such as Forward List Scheduling to schedule the application. For this example, let us assume that our target architecture has four cores, two *LITTLE* cores and two *big* cores. FLS will try the next task in the ranking on all available cores and pick the best option (e.g., lowest energy consumption, shortest makespan). Thus, the first task to be scheduled is *Task 0* (as seen in the ranking in Figure 2.4). FLS tries the task on each core and then finally schedules it on the core, which leads to the best local result. Therefore, before arriving at a final output, the scheduler would also try out cores *big - 0*, *big - 1*, *LITTLE - 2* and *LITTLE - 3*. In this case, core *big - 0* and *big - 1* are the best options. We pick the first best option as a tiebreaker, as shown in Figure 2.5a. The best local result depends on the preferred optimisation target. The scheduler then continues with the following task (*Task 5*) and tries it on all cores before deciding on a final schedule. The scheduler will continue to schedule one task after another in order of the ranking until all tasks are scheduled. The final schedule of the example is shown in Figure 2.5b.



(a) Initial schedule for Figure 2.3 after the first round of an FLS scheduling algorithm.

(b) Final schedule for Figure 2.3.

Figure 2.5: The two figures show the first and final example schedules for the DAG shown in Figure 2.3.

3

ENERGY MEASUREMENT

This chapter explores the importance of sampling frequency in energy measurements. We explain and detail important energy measurement concepts relevant to the other chapters of this thesis. This includes a description of the measurement system that we use. Furthermore, we introduce the statistical methods required for exploring the impact of sampling frequency on energy measurements. Overall we answer research question 2 in this chapter. This includes a visual representation of the error when using a too low sampling rate.

This chapter is based on:

- **J. Roeder**, S. Altmeyer, and C. Grelck *"Can we trust our energy measurements? A study on the Odroid-XU4"* [144], in *15th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT at ECRTS 2022)*.
- **J. Roeder**, S. Altmeyer, and C. Grelck *"Energy Measurements of 9 Rodinia Benchmarks executed on the Odroid-XU4."* [145], in *Dataset published on UvA figshare*.

3.1 INTRODUCTION

Energy consumption is one of the most critical design criteria for battery-powered systems. Thus, it is not surprising that decreasing energy consumption from the software side is an essential topic in various research fields such as IoT, edge computing and cyber-physical systems (e.g., [16, 24, 30, 56, 76, 89, 102, 108, 142, 148]; for a survey see [123]).

A crucial part of energy-related research is measuring energy consumption to show tangible improvements on real hardware. To measure a device's energy consumption, we need to measure voltage and current, two continuous signals. Continuous signals are measured in discrete intervals at a given sampling rate. Theoretically, we need to measure at twice the highest frequency desired to be measured (Nyquist rate [103]); otherwise, the time

series might be distorted. However, from a practical point of view, it is unclear what the highest desired frequency in this case is.

In general, we find that authors and reviewers place little importance on the measurement setup, as papers do not report the setup or lack details on the devices and methods used, e.g., [16, 56, 75, 89, 108, 142]. Publications that report the measurement system used do not investigate or consider the impact of the measurement setup on the measurements' accuracy. For example, [76] naturally used the energy measurement system (SmartPower2[1]) provided by the manufacturer of their target board (Odroid-XU4). According to the publication, the SmartPower2 measures at 1Hz. Additionally, we could not find any studies on the measurement error of the SmartPower2. In this chapter, we raise substantial doubts about the reliability of low-frequency measurements. As a community that makes decisions based on energy consumption, we must know that our experimental setups are reliable.

In this chapter, we systematically investigate the impact of the sampling frequency on the energy measurement accuracy. More specifically, we measure the energy consumption of an Odroid-XU4 executing a variety of benchmarks. The measurement system used samples at a high rate; the original power traces can then be downsampled. We then compare the downsampled traces against the original traces. That way, we can alter the sampling frequency of the voltage and current measurements while keeping all other variables equal.

The chapter is organised as follows. We provide background information and detail our methodology in Section 3.2. Section 3.3 covers our results and discussion. Then in Section 3.4, we discuss related work. Finally, we present our conclusion in Section 3.5.

3.2 BACKGROUND & METHODOLOGY

In order to investigate the importance of sampling frequency, we need a measurement system, a target system, programs to measure and a way to compare different sampling frequencies. In this section, we start with a short discussion about power measurements in general. We then dive into the importance of sampling frequency. Next, we introduce our experimental setup and the benchmarks used. Lastly, we detail the statistical tests needed.

1 https://www.hardkernel.com/?s=smartpower2&post_type=product&lang=en

3.2.1 *Power Measurements*

Power measurements can be done at the AC or DC sources. Discussing and comparing the advantages of either method is beyond the scope of this work. However, in general, the AC-DC converter (i.e., power supply) will have some inefficiencies, and measuring after the converter (i.e., at DC) disregards that loss. Furthermore, the loss can fluctuate with the load, i.e., power supplies are most efficient at a given load and have lower efficiency at lower/higher loads. An additional reason to measure after the converter is that the energy consumption is most crucial for battery-powered systems, which use DC.

For an overview of different DC measurement methods, see [97] and [64]. In this chapter, we consider the shunt resistor method, which observes the voltage drop across a resistor as it is widely used. We place the resistor in series with the load. And as we know the resistor value, Ohm's law can be applied to calculate the current of the load. Furthermore, we can place the resistor before the load (high side) or after the load (low side). Low side sensing is cheaper as the amplifier is more straightforward, but it has some disadvantages compared to high side sensing. More specifically, high side sensing is not sensitive to ground disturbances and can detect fault conditions. Therefore, high side sensing is more often used when accurate measurements are needed. In contrast, low side sensing is mostly used in mass production systems where accuracy is not as crucial [13]. We will focus on high side sensing for increased accuracy.

The resistive current sensing method can be deployed directly on a target board, i.e., the board comes with an integrated power measurement function (e.g., Odroid-XU+E[2] used in [75]). Or the method can be deployed on a separate device such as the SmartPower2 or Qoitech Otii[3]. Onboard sensors are polled from the target system itself and can be polled at different frequencies. Additionally, onboard sensors are intrusive as the polling of the sensors impacts the energy consumption of the target.

The voltage drop across the resistor is amplified and converted using an Analogue-Digital-Converter (ADC). Current sense amplifiers such as the TI INA250[4] can be combined with an ADC. The ADC then digitises the information for further analysis.

2 https://www.hardkernel.com/shop/odroid-xue/
3 https://www.qoitech.com/otii/
4 https://www.ti.com/product/INA250

Once we obtain the voltage and current readings, we can calculate the power (Watt). Multiple power readings result in a power trace. As we know the time between different power readings, we can calculate the area under the trace, resulting in the energy consumption (Joule).

3.2.2 *Sampling Frequency*

Continuous signals cannot be converted to digital information continuously. Instead, we have to measure them at discrete intervals. The accuracy of the measurements heavily depends on the sampling frequency. In theory, to reproduce an (AC) power signal, one needs to measure voltage and current at four times the highest sinusoidal frequency [132]. However, the DC consumption is not sinusoidal and instead alternates with the requirements of the load. In the case of a microcontroller, the current requirements change with the Dynamic Voltage and Frequency Scaling (DVFS) settings, instructions per clock and the actual instructions being executed [135]. Thus, the required sampling frequency depends on the length of the program being executed and the instruction mix.

3.2.3 *Setup and Target system*

High-performance embedded systems like the Odroid-XU4 and the NVidia Jetson Nano are all relatively similar with respect to clock frequency and CPU architecture. It is an octa-core system with 4 big cores (Cortex-A15), 4 LITTLE cores (Cortex-A7) and a Mali-GPU (T628 MP6). The two separate core clusters and the GPU all form individual voltage islands (i.e., 3 voltage islands). The voltage and the frequency can be set separately for each voltage island. The Odroid-XU4 runs an RT-patched Linux.

The Odroid-XU4 is accompanied by an energy measurement system called the SmartPower2. However, due to the low sampling frequency (1Hz), we decided not to use the system. Instead, we measure the energy consumption of the Odroid-XU4 with the Qoitech Otii on the high side. The Otii has a maximum measurement error of 0.1% + 150µA (i.e., at higher currents, the error is approaching 0.1%) and has a sampling frequency up to 4kHz. The main criticism of the shunt resistor method is that a single shunt is only helpful in a limited current range [26, 64]. The Otii has multiple shunts to measure very low currents (10 µA with 0.6% error) up to 5A peaks. It measures across all shunts simultaneously; thus, switching the current range (i.e., between shunts) does not result in any loss of data points.

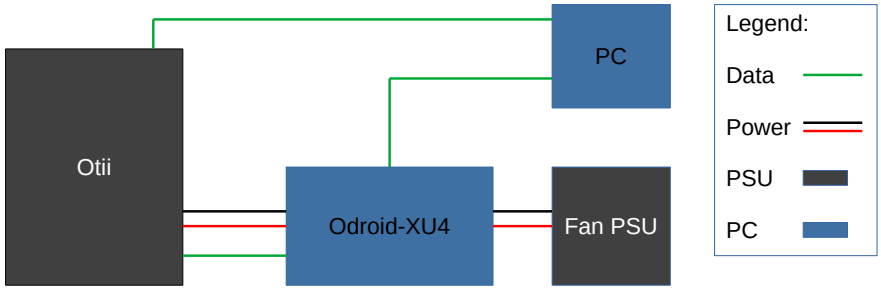


Figure 3.1: Measurement setup including: Qoitech Otii, Odroid-XU4, Fan power supply, and PC

Figure 3.1 shows our setup. The Odroid-XU4 receives its power from the Otii and is simultaneously connected via the UART pins to the Otii. This means that the power measurements can be directly linked to messages sent by the Odroid-XU4. The fan of the Odroid-XU4 is powered via a separate circuit and thus does not affect the power measurements of the Odroid-XU4. Before each set of measurements, we calibrate the Otii. Additionally, we warm up all connected components by executing the *heartwall* benchmark 50 times.

3.2.4 *Downsampling*

The Otii continuously samples at 4kHz. Thus, lowering the sampling rate is impossible; instead, we downsample the results. That means if we sample at 4kHz but want a sampling rate of 2kHz, we only consider every second measurement. Thus, unrelated sampling factors do not play a role (e.g., different measurement error on another measurement device). In this chapter, we investigate 22 sampling rates (in Hz: 1, 2, 3, 4, 5, 10, 20, 30, 40, 50, 100, 150, 200, 250, 300, 350, 400, 500, 600, 800, 1000, 2000, 4000).

3.2.5 *Benchmarks*

We use the Rodinia benchmark suite [27] as target programs/tasks. The suite offers a range of targets (C, OpenCL, Cuda), different algorithms & workloads and is widely used. Berkeley's dwarf taxonomy inspired the suite's benchmark selection [14]. Each benchmark can further be adjusted via its input parameters. This leads to an extensive range of run times and processor loads.

We use nine benchmarks (backpropagation, BFS, Heartwall, Hotspot, Kmeans, LU-Decomposition, Nearest Neighbour, NW, SRAD) out of the suite as they can be executed on the Odroid-XU4 with minimal adaptations. The other benchmarks would have required significant changes to the code. Besides the input parameters, we also vary the target DVFS settings and core. We measure all benchmarks on the LITTLE cores, the big cores, and the GPU (i.e., OpenCL version). However, two benchmarks (BFS and SRAD) were only measured on the big and on the LITTLE cores because the OpenCL versions did not work on the Odroid-XU4. This leads to 842 unique benchmark/target/DVFS combinations. For each combination, we collected 50 power traces; thus, we collected 42100 power traces. The resulting dataset is available for download [5][145]. Additionally, the repository containing the analysis scripts is also available [6].

We do not consider other benchmarks such as STR2RTS [115] or ULPMark [46] as they are developed for significantly different devices (e.g., microprocessors, ultra-low power IoT systems). However, future work could expand on the benchmarks and experiment types. Interesting additional benchmarks include the ADASMark[47] and MLMark[48] tests. Unfortunately, these were not yet available when the experiments were conducted. Furthermore, the experiments could also be expanded to include multi-core tests, for example, running benchmarks on the big and LITTLE cores in parallel.

3.2.6 *Statistical equivalence testing*

We measure a non-deterministic system (out-of-order pipeline etc.). Additionally, the measurement system is not perfect and contains some noise. Thus, we repeat measurements for each combination, as there is not a single "correct" value. That also means that downsampling a single time series and then calculating the error will indicate how much worse a lower frequency is. However, this approach does not offer a statistical indication. Therefore, we must analyse all sets and their downsampled counterparts with statistical tests.

In a regular two-sided t-test, we test if two samples are different. The null hypothesis is that there is no difference (μ_D) between the two samples (Equation (1)).

5 https://doi.org/10.21942/uva.19665564.v1
6 https://bitbucket.org/uva-sne/energymeasurementanalysis/

$$H0 : \mu_D = 0 \tag{1}$$

$$H1 : \mu_D ! = 0 \tag{2}$$

Suppose the t-test indicates a significant difference (e.g., p-value smaller than 0.05). In that case, we can reject the null hypothesis and accept the alternative hypothesis that the two samples are different (Equation (2)). Thus, a t-test offers evidence in favour of the alternative hypothesis at a given confidence level (e.g., 99%). If the t-test is not significant, this is often counted as support for the null hypothesis, i.e., that there is no difference between the samples or no effect. However, a non-significant test result often results from limited statistical power. Thus, it is impossible to know whether a non-significant result indicates equivalence (absence of an effect) or only false equivalence and lacks statistical power [111].

Instead of proving the absence of an effect, we can show that the likelihood of an effect being smaller than a given (low) value to be significant. This is called equivalence testing. To test for equivalence between two samples, we use a method called *Two One-Sided T-tests* (TOST) [111]. As a TOST consists of two tests, it has two null hypotheses (Equation (3)) and (Equation (4)). The first test determines if the difference between the two samples (μ_D) is smaller than the accepted lower bound ($-M$). The second one tests if the difference is larger than the upper bound M.

$$H0_1 : \mu_D < -M \tag{3}$$

$$H0_2 : \mu_D > M \tag{4}$$

Combining both test results in the alternative hypothesis (Equation (5)) that μ_D falls between $-M$ and M. Thus, if both t-tests are rejected, we have support for the alternative hypothesis that the difference between the two samples is smaller than a chosen M [96]. Figure 3.2 visualises the difference between a normal t-test and a TOST.

$$H1 : -M < \mu_D < M \tag{5}$$

The majority of our 842 measurement sets are not normally distributed (76.0%) according to both the Shapiro-Wilk test [122] and D'Agostino-Pearson's test [34]. One randomly picked set (out of 842 sets) of the energy consumption distributions can be seen in Figure 3.3. It clearly shows that the values are clustered heavily in the centre with only a few samples in the tails, hence not representing a normal distribution. Therefore, we use a non-parametric TOST based on Wilcoxon's Signed Rank test [137]. We do all tests at a 99.9% confidence ($\alpha = 0.1\%$).

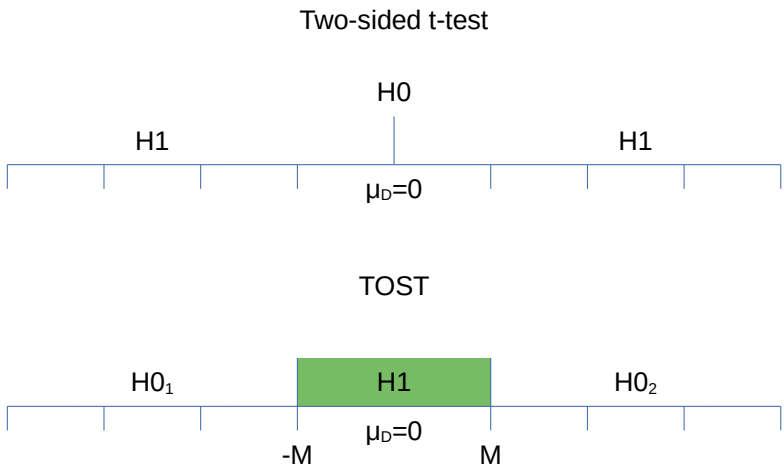


Figure 3.2: Comparison of a two-sided t-test and a TOST.

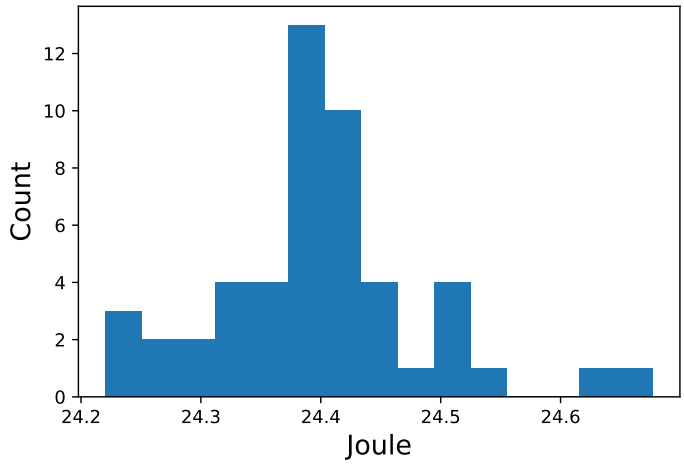


Figure 3.3: An example energy consumption distribution randomly picked from the 842 different sets. The kurtosis is 1.53, and the skewness is 0.66.

One significant difference between a standard t-test and an equivalence test is that one needs to determine what (low) difference (M) is acceptable (i.e., considered to be less than a substantial effect). We analyse the impact of 8 "acceptable error" levels (20%, 10%, 8%, 6%, 4%, 2%, 1%, 0.5%) and the sampling level required to achieve equivalence at that level across all 842 experiment combinations.

3.3 RESULTS & DISCUSSION

The 42100 power trace time series can be analysed in multiple different ways. Table 3.1 summarises basic statistics of all power traces and shows that our benchmarks/target/DVFS combinations cover a wide range of run times and power. Overall we observe that the downsampled traces mainly resulted in a power consumption underestimation (98.9% of the cases) and in very few cases of overestimation (1.1%).

Table 3.1: Summary statistics for all benchmark executions.

	Runtime (s)	Power (W)
Mean	9.87	2.99
Min	0.90	1.82
Max	48.15	8.44

Figures 3.4 and 3.5 show one of the power traces. Figure 3.4 shows the original power trace at the full sampling frequency of 4kHz and Figure 3.5 shows two downsampled versions. The solid blue line in Figure 3.5 shows how the power trace looks like if we had sampled at 1Hz. Compared to the original trace, we can see that it misses most of the data. Furthermore, it also completely misses data at the last second, as the execution time was 3.98 seconds. It is possible to make up for the last missed measurement by either measuring at second 4 or using the last known measurement. Either method will still lead to a significant error. The dashed red line shows the same power trace but downsampled to 10Hz. It already has much more detail than the 1HZ line but still misses a significant part of the signal.

Figure 3.6 shows the maximum percentage error between the original energy measurement and the downsampled measurement for each frequency. Thus, the maximum error observed across all 842 combinations at 1Hz is 80%. The maximum error only drops below 0.5% at a sampling frequency of 500Hz.

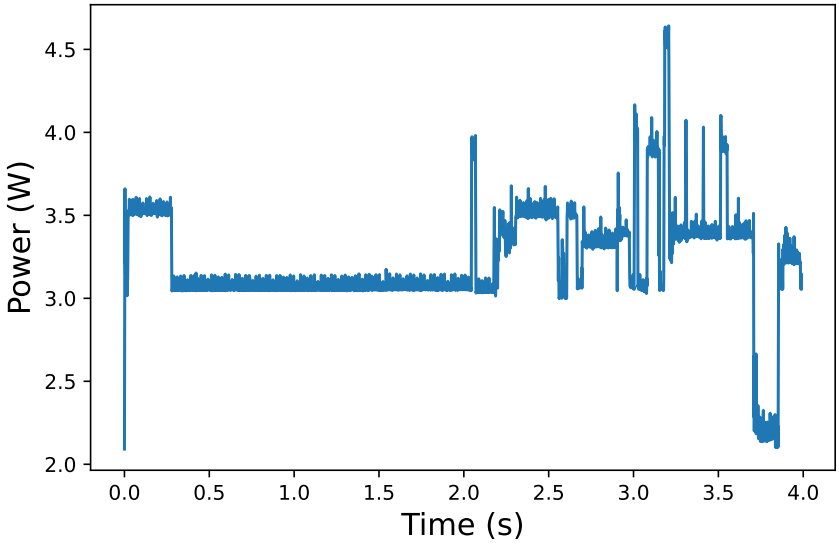


Figure 3.4: Original power trace sampled at 4000Hz.

The maximum error only represents a single measurement and does not carry any statistical meaning, which is the reason for employing equivalence testing. Figure 3.7 shows the minimum frequency required to achieve equivalent results for all 842 combinations compared to the full sampling frequency. Thus, if a measurement error of up to 20% is acceptable, then a 30Hz sampling rate would lead to an equivalent result for all experimental combinations. At an acceptable error of 0.5%, 600Hz results in an equivalent result, which indicates a similar level as Figure 3.6.

Lastly, in Figures 3.8 and 3.9, we investigate the relation between the error, benchmark run-time and sampling frequency. Interpreting the 3D graph showing the relationship between all three is not straightforward as the resulting graph contains a lot of non-continuous data points (Figure 3.8). To ease the interpretation, we smooth the data and the relation between the three variables using a polynomial, multi-variable regression based on a Multi-Layer-Perceptron (Scikit-learn: default parameters, hidden layer size = (64, 128, 256, 512)). This allows us to interpolate the error to other sampling frequencies and run times. We use 80% of the data for training. The mean absolute error on the test set is 0.0065.

We use the regression to predict a measurement error given a sampling frequency and run-time. Plotting the regression for the sampling frequency range 1Hz to 140Hz and run-times between 0.5 and 40 seconds results in

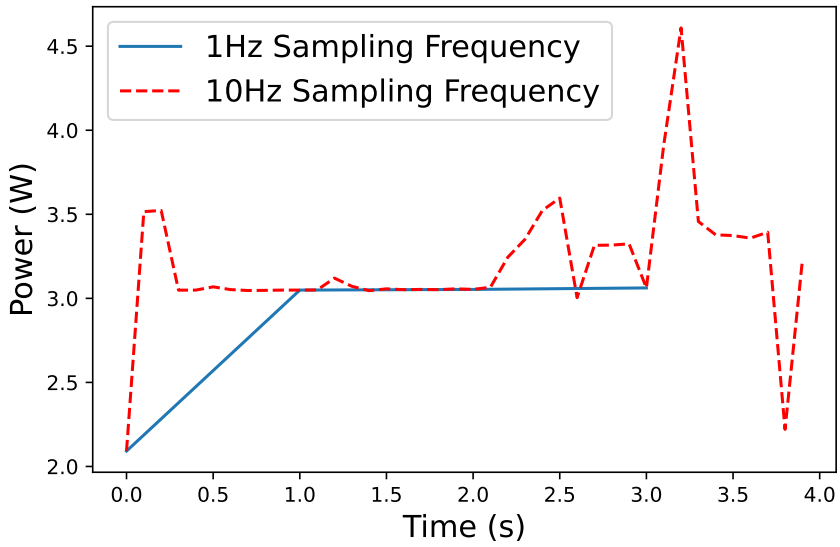


Figure 3.5: Downsampled power traces.

Figure 3.9. The figure shows that low sampling frequencies lead to poor results for the selected benchmarks, even for longer run times. That means that the selected long-running benchmarks contained a significant amount of faster peaks that were missed at a low sampling rate. The error for short tasks remains higher even with higher sampling frequencies. As such, the results obtained with a SmartPower2 are of limited use in an academic setting.

For this set of benchmarks, input parameters, target platform, and DVFS settings, a sampling frequency between 350Hz and 600Hz is sufficient (given an error of 1% and below). However, much shorter programs might need significantly higher sampling rates, or one will have to measure the target task differently. For example, measuring a very short task (a few CPU cycles) will be missed even at a sampling frequency of 4kHz; thus, artificially inflating the task could work (e.g., a loop).

In the remainder of this work, we use the same energy measurement setup described in this chapter. Additionally, we take care to execute sufficiently long-running tasks so that the energy consumption can be measured accurately.

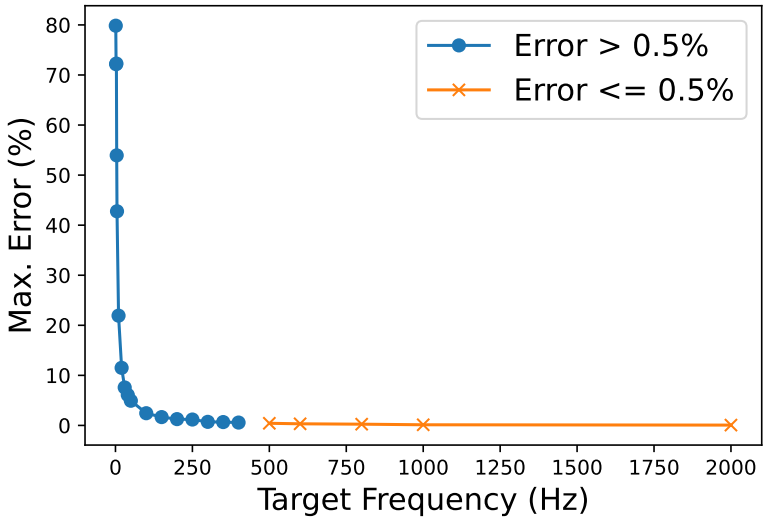


Figure 3.6: Maximum error rate at each artificial frequency across all 842 experiment sets.

3.4 RELATED WORK

Cloutier et al. demonstrate that decreasing the sampling frequency from 100Hz to 1Hz results in a significant loss of the power trace detail [30]. However, they do not further investigate the impact of this decrease on energy measurement accuracy. Additionally, we can show that the accuracy of measurements at 100Hz is significantly lower than at 4kHz.

Diouri et al. investigate different energy measurement systems for servers [43]. They conclude that higher sampling rates are not necessarily good as they can introduce noise that could mask other trends. However, only because a signal is noisier doesn't mean that the noise is erroneous and can thus be disregarded for energy measurements. One can always downsample a trace or smooth it to investigate possible hidden trends. Furthermore, server measurements could already be noisier than high-performance embedded systems due to architectural reasons, different target applications and short background tasks. Looking at Figure 3.4, we cannot confirm that a high sampling rate masks the trends of an application. Lastly, Diouri et al. do not investigate if the downsampled traces lead to equivalent energy measurements.

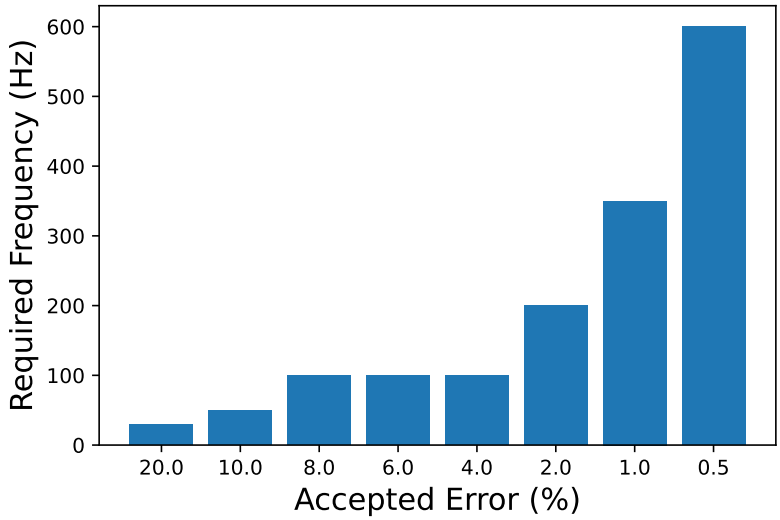


Figure 3.7: Frequency required to reach equivalence given an acceptable error.

Djupdal et al. [44] develop a high-performance embedded system oriented energy measurement system. And in [73], the authors describe two high-sampling frequency power measurement methods (up to 500kHz) for servers and server components. However, they do not analyse the importance of the sampling frequency and if lower sampling frequencies can achieve similar results.

Buschoff et al. [26], and Jiang et al. [77] develop measurement techniques for low-powered embedded systems. They target devices with long sleep times that only consume energy in a few fast bursts. In contrast, we focus on high-performance embedded systems that carry out computationally demanding tasks.

Nakutis et al. [97] and Hergenröder et al. [64] summarise the different power measurement methods and highlight the importance of the sampling frequency. However, neither paper empirically shows the resulting error.

3.5 CONCLUSION

Research into reducing the energy consumption of embedded systems is popular. Hence, we need to measure the energy consumption of embedded systems. However, researchers and reviewers alike often pay little attention and consideration to how to measure energy consumption. One crucial

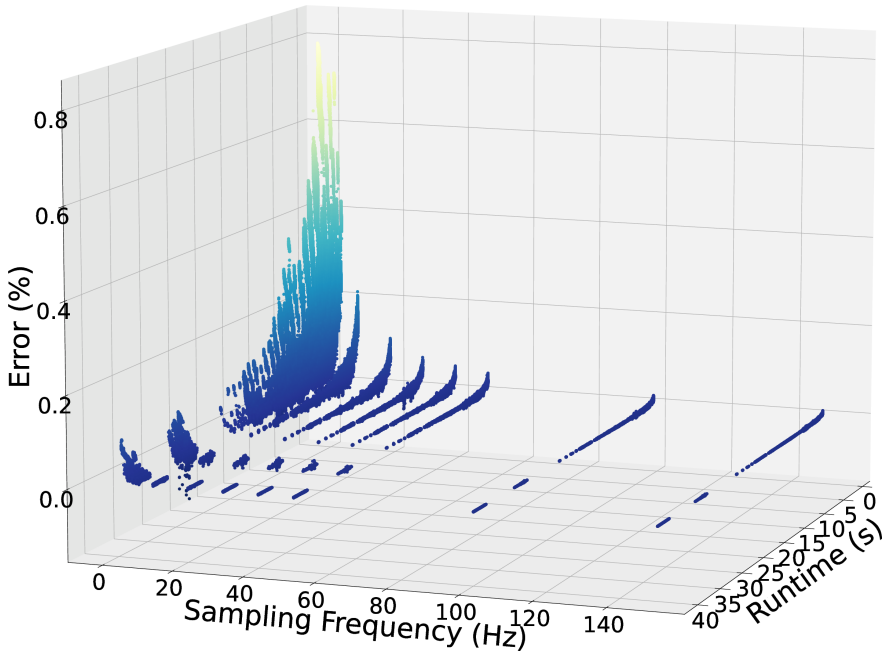


Figure 3.8: Relation between the error, benchmark run-time and the sampling frequency for all power traces downsampled to between 1Hz and 140Hz.

aspect of energy measurements for high-performance embedded systems is the sampling frequency of the analogue signal.

We show that for a wide range of Rodinia benchmarks executed on the Odroid-XU4, the minimum sampling rate is 350Hz if a 1% measurement error is acceptable. Measuring at 1Hz results in errors as high as 80%. This shows that systems such as the Hardkernel SmartPower2 (measurement system accompanying the Odroid-XU4) cannot be used to draw conclusions and that measurement methods with low sampling rates are only of limited use in an academic setting. Some papers on reducing the energy consumption of high-performance embedded systems should be re-evaluated.

To reliably research and investigate methods for reducing energy consumption, we must measure energy consumption accurately. That means we need to pay more attention to our experimental setup and report our setup accurately. Careless experimental designs lead to two problems: First, we potentially focus too much on the wrong methods (false positive conclusion). Second, we discard methods that do not look promising but are, in reality, good options (false negative conclusion).

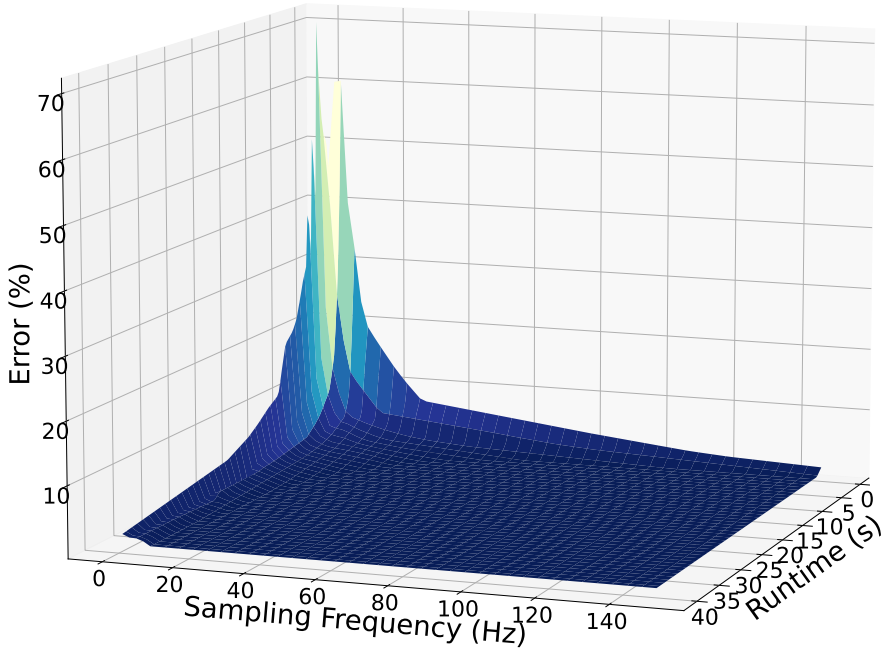


Figure 3.9: Regression analysis of the error with respect to the benchmark run-time and the sampling frequency.

4

ENERGY-AWARE SCHEDULING

This chapter explores our energy-aware scheduling technique. We define a task model which integrates multiple versions for dependent tasks. This is followed by introducing a new fine-grained energy model which can easily be integrated into scheduling strategies. Next, we propose new energy-aware ranking algorithms (i.e., establishing a total order) to initialise the Forward List Scheduling heuristics. The new ranking algorithms are compared against state-of-the-art ranking methods (BFS with WCET, DFS with WCET, BFS with laxity, and BFS with energy laxity), and overall our new approach decreases energy consumption by an average of 26.3% and 27.8% in comparison to two state-of-the-art schedulers (HEFT and ARSH-FATI). Additionally, our approach generates schedules that are close to optimal (determined using an ILP) concerning energy consumption. Lastly, we empirically show that our energy-aware scheduler predictions are close to the actual energy consumption measured (largest error 15.8%) on an Odroid-XU4 board.

This chapter is based on:

- **J. Roeder**, B. Rouxel, S. Altmeyer, and C. Grelck *"Interdependent Multi-version Scheduling in Heterogeneous Energy-aware Embedded Systems"* [147], in *13th Junior Researcher Workshop on Real-Time Computing (JRWRTC at RTNS 2019)*.
- **J. Roeder**, B. Rouxel, S. Altmeyer, and C. Grelck *"Energy-aware scheduling of multi-version tasks on heterogeneous real-time systems"* [148], in *Proceedings of the 36th Annual ACM Symposium on Applied Computing (SAC 2021)*.
- **J. Roeder**, B. Rouxel, S. Altmeyer, and C. Grelck *"Scheduling multi-version tasks on heterogeneous IoT systems using energy-aware ranking"* [149], in *Under review*.

4.1 INTRODUCTION

In recent years, we have seen an ever-increasing number of deployed IoT and IIoT devices. In 2022 the number of active IoT endpoint connections

is expected to grow by 18% to 14.4 billion devices. By 2025 the number of devices is expected to almost double to 27 billion connected (I)IoT devices [10]. Not only do we see a significant increase in the number of devices but also a broader range of possible deployment areas. Especially in the IIoT domain, such as the medical and public sector, many growth opportunities exist as more image and big data edge analysis applications are developed [45, 84, 88, 109]. One recent example of a public sector application is the use of deep learning for vehicle tracking on the edge [45]. This analysis must be done on the edge as privacy, security, bandwidth, and storage are all limiting factors [36, 45]. These types of applications benefit from the growing number of high-performance heterogeneous single-board computers, such as the Odroid-XU4 [59] or Nvidia Jetson boards [101], as they offer remarkable compute power at low energy consumption. Energy consumption is not only crucial in light of climate change and the overall CO_2 footprint of the (I)IoT sector due to the large number of devices (27 billion by 2025) but also for individual battery-powered devices where recharging is prohibitively expensive or even impossible. Examples of such applications are wildlife/poacher camera traps located in remote areas and communicate via LoRaWAN/Radio etc. (e.g., [74]). Replacing or recharging batteries comes at a high cost. Thus, the cameras are movement sensor triggered to preserve energy. However, a movement does not necessarily mean an exciting event occurred. Additionally, wildlife research requires identified animals and not full images. Storing or sending all frames from the camera is wasteful. Hence, many papers focus on deploying edge-based deep learning to these traps (e.g., [117, 120, 138]). In conclusion, we not only need to reduce the energy consumption of the overall IoT sector, but we also need to reduce the energy consumption for the growing number of use cases that require heavy computations on edge-based battery-powered devices. Hence, the challenge is to reduce the overall energy consumption on complex, heterogeneous architectures.

To tackle this challenge, we need to utilise the heterogeneous capacity of the hardware entirely. This, among others, includes heterogeneous CPUs (e.g., big.LITTLE [1]) and accelerators (e.g., GPUs). Different compute units may require architecture-dependent binaries (e.g., CPU vs GPU) due to different instruction set architectures (ISA). The absence of binary compatibility makes multi-version tasks a natural, if not necessary, starting point for our work on scheduling for modern heterogeneous IoT platforms. Multi-version tasks have equivalent functional behaviour (i.e., identical input yields equivalent output) but different non-functional behaviour, namely time and energy

consumption. As multi-version tasks are required to tackle binary incompatibility, we can further exploit this feature and support versions resulting from, e.g., different compiler flags or different functionally-equivalent algorithms. The necessity to include multiple task versions further increases the complexity of reducing the overall energy consumption (e.g., balancing which task and version make use of an accelerator).

Most modern systems allow Dynamic Voltage and Frequency Scaling (DVFS). For each task, CPU, or GPU type, there is a clock frequency that minimises energy consumption. A lower frequency leads to longer runtime and thus increases static energy consumption. A higher frequency leads to shorter runtime, but the necessary higher voltage increases dynamic energy consumption. In modern CPUs, the clock frequency cannot be altered per core but only per core cluster (i.e., voltage island). Hence, we need to pick the best frequency for each voltage island with respect to the different tasks executed on that island. We take advantage of DVFS for different voltage islands to reduce the energy consumption of a whole application consisting of multiple tasks.

Heterogeneous platforms, multiple task versions, voltage islands and DVFS extend common scheduling challenges: schedulers now must decide on which computing unit and at what frequency a task (version) should be executed to reduce the overall energy consumption. We propose an offline Forward List Scheduling (FLS) based approach for multi-version task scheduling, which: 1) fully utilises the heterogeneous CPU and accelerators; 2) takes advantage of per voltage island DVFS; 3) dynamically adjusts the frequency throughout the application run time; 4) selects the optimal version of each task with respect to the energy consumption of the whole application. The motivation to use an offline scheduling approach over a dynamic one is that the dynamic approach would need to include all versions and DVFS data in the binary. This introduces prohibitive overhead as all the data has to be kept in memory. Furthermore, we propose a greedy heuristic instead of evolutionary algorithms (EA) or genetic algorithms (GA), as EAs and GAs often require unacceptably long evaluation times [125].

Many relevant applications can be modelled as Directed Acyclic Graphs (DAG), where the nodes represent different tasks of an application and the edges represent the dataflow between tasks. From a DAG, we can extract the partial order of tasks. However, most greedy-, evolutionary- and genetic-algorithms (e.g.[78, 94, 116, 133]) require a total order of tasks instead. Thus, we propose a new energy-aware ranking algorithm and explore the effect of different ranking strategies on our offline heuristic.

The remainder of the chapter is organised as follows: In Section 4.2, we describe the system model. In Section 4.3, we describe the heuristic algorithm and our new energy-aware ranking algorithm. Section 4.4 details the experimental setup. In Section 4.5, we investigate the impact of different ranking algorithms and select a sub-set of ranking algorithms for the remaining sections. Section 4.6 demonstrates the viability of our heuristic scheduler in comparison to a single-version approach, a makespan heuristic, HEFT [131] and the energy-aware genetic algorithm-based scheduler ARSH-FATI [133]. Section 4.7 shows that our heuristic scheduler produces results close to the optimum and that the predicted energy consumption is close to the actual energy consumption. In Section 4.8, we discuss related approaches before concluding in Section 4.9.

4.2 SYSTEM MODEL

In this section, we first detail our platform model (Section 4.2.1), followed by the task model (Section 4.2.2) and lastly, we explain our energy model (Section 4.2.3).

4.2.1 *Platform Model*

Our approach is fully platform-independent and can be applied to a wide range of heterogeneous (embedded) system architectures. Our model supports: multiple voltage islands, DVFS, heterogeneous CPUs, GPU-style accelerators and in-application frequency switching. Additionally, we support GPU tasks that require not only the GPU but also a CPU core for controlling the GPU. The only restrictions our approach has is that a voltage island needs to be homogeneous and that tasks can be bound to a specific core. Our approach does not limit the number of voltage islands. Each voltage island can have a differing number of cores and DVFS parameters. We consider co-processors to be separate voltage islands. Our approach does not limit the number of different co-processors either.

4.2.2 *Task Model*

We consider applications represented as Directed Acyclic Graphs (DAG), hereafter called taskgraphs. In a graph, $G = (\tau, E)$ the set of nodes/vertices τ represents the tasks, and the set of edges E represents data dependencies between tasks, i.e., a producer task needs to be completed before the cor-

responding consumer task may start executing. Our task model supports multiple sources and sinks.

Each task τ_i consists of a (non-empty) set of task versions V. The different versions of a task τ_i are functionally equivalent (i.e., they implement an equivalent input/output relation) but differ in their non-functional properties. Different versions can be the result of: 1. targeting different functional units (e.g., big core, LITTLE core or GPU); 2. using varying compilation flags to, for example, optimise code for energy consumption, binary size, speed or architecture features [105]; 3. different algorithms or implementation variants. This results in an ample state space, where picking the best version for a given task can be challenging. All tasks in the taskgraph must be executed. However, only one version of each task is executed. Hence, the scheduler chooses the version to achieve the best trade-off between energy and performance.

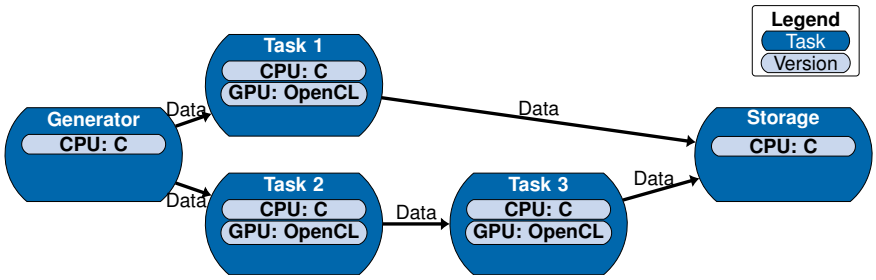


Figure 4.1: Illustration DAG with multi-version tasks.

Figure 4.1 presents a synthetic taskgraph which we use to illustrate our approach. It consists of five different tasks: one generator task, one storage task and three computational tasks in between. The tasks have one or two different versions each. They target different compute unit types: CPU and GPU. Edges are labelled and represent data transfer (or data dependencies) between tasks.

4.2.3 *Energy Model*

Aligned with the state-of-the-art [19, 55, 72] our energy model Equation (6) consists of a static part (E_s) and a dynamic part (E_d). Both static and dynamic energy consumption are computed during scheduling as they depend on scheduling decisions (e.g., selected versions, selected frequencies). In contrast to previous research[19, 55, 72], we consider the impact of frequency on

static energy consumption. Frequencies are not continuous and the dynamic energy consumption is measured per task.

$$E = E_s + E_d \tag{6}$$

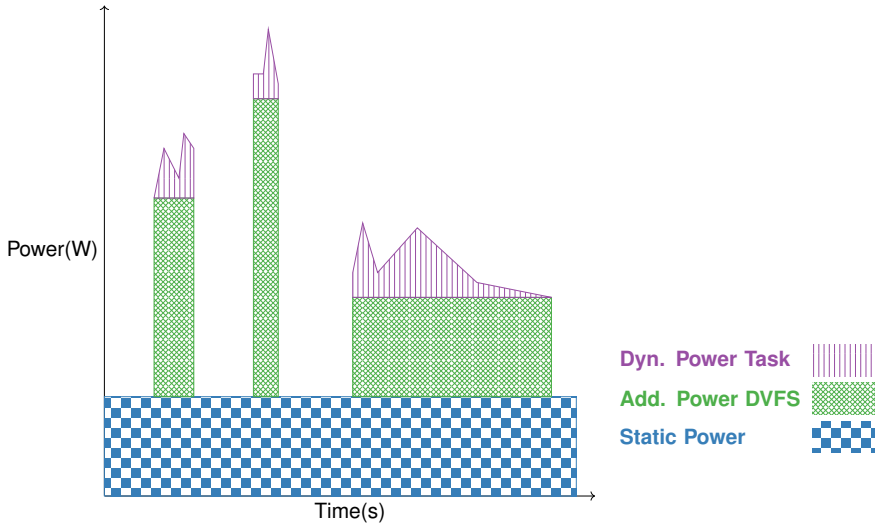


Figure 4.2: Energy Model

Figure 4.2 illustrates our energy model based on an example, the x-axis represents the time, and the y-axis represents the power. The checkerboard blue box (spanning the complete width of the figure) represents the energy consumption due to the device being powered up. The crosshatch green boxes represent the energy consumption required to run at higher frequencies. The third part of our energy model is the dynamic power required to run a task, represented by the irregular shapes on top of the static power components.

The static energy consumption (E_s), Equation (7), can be split into two elements E_{sm} and E_{sf}. Firstly, E_{sm} corresponds to the energy consumed by the board because it is powered on and depends only on the worst-case duration (i.e., application makespan) it remains powered on. Thus, E_{sm} is equal to the average measured power, in Watt, required at the lowest frequency ($W_{average}$) multiplied by the overall schedule makespan (C) of the application (Equation (8)).

$$E_s = E_{sm} + E_{sf} \tag{7}$$

$$E_{sm} = C * W_{average} \tag{8}$$

Secondly, E_{sf} corresponds to the additional energy consumption required to operate the board at a given frequency. As the board consists of multiple voltage islands, E_{sf} (Equation (9)) corresponds to the consumed energy by all voltage islands ($i \in I$) at each specific frequency ($f \in F_i$).

$$E_{sf} = \sum_{i \in I} \sum_{f \in F_i} C_{i,f} * W_{i,f} \tag{9}$$

Hence, the time spent on each voltage island at each frequency ($C_{i,f}$) is multiplied by the average additional energy consumption at that frequency ($W_{i,f}$). This time depends on the scheduler's decisions and, more precisely, on the Worst-Case Execution Time (WCET) of the selected version for each task. If a voltage island contains more than one core, it is crucial not to over-accumulate the time spent in each frequency. The WCET of two tasks executing concurrently on the same voltage island (i.e., on two cores) should not be summed up to compute the time spent in a given frequency. Instead, the longest time of the two tasks must be accounted for. Let us consider the two task schedule shown in Figure 4.3. Both tasks are scheduled on the same voltage island (*V1*) but on different cores. That means that the total time $C_{1,f}$ at frequency f is 10 and not 13. The same holds for the schedule in Figure 4.4 where $C_{1,f}$ is equal to 10 and not 12.

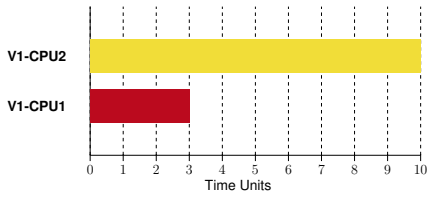


Figure 4.3: Illustration of the total time ($C_{1,f} = 10$) computation for two tasks scheduled at the same time on a single voltage island.

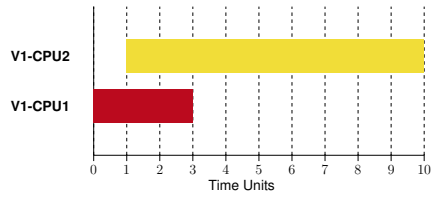


Figure 4.4: Illustration of the total time ($C_{1,f} = 10$) computation for two tasks scheduled on the same voltage island with an overlap.

If two tasks are executed on different voltage islands, the WCET of each task must be accounted for, as the time spent in each frequency is voltage island specific. Thus, in Figure 4.5 the $C_{1,f}$ of the first voltage island (*V1*) is equal to 3, and the $C_{2,f}$ of the second voltage island (*V2*) is equal to 10.

The dynamic energy consumption depends on the workload executed by a compute unit at a given frequency. Unlike most existing power models, we measure the energy consumed by each task version for each frequency on each corresponding compute unit. Measures are performed *a-priori* to

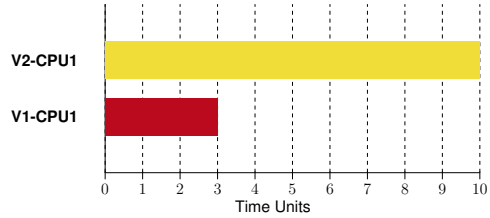


Figure 4.5: Illustration of the total time ($C_{1,f} = 3$ and $C_{2,f} = 10$) computation for two tasks scheduled on two different voltage islands.

scheduling the application and in isolation (all other compute units are idle, and no other task is executing). To compute the dynamic energy consumption, we measure the total energy consumption and then subtract the two static power components (i.e., the energy consumption due to running at the base DVFS and the additional energy consumption due to the increased frequency). We work with average power ($W_{average}$) for the static part.

The total dynamic energy, Equation (10), consumed by an application E_d is the sum of all selected task versions dynamic energy at a given frequency targeting a specific compute unit ($E_{p,u,f}$). This approach allows us to better account for different energy requirements of tasks since Balsini et al. [16] and Vasilakis et al. [136] showed that one-size-fits-all dynamic energy consumption for the whole application is unrealistic.

$$E_d = \sum_{p \in \tau} E_{p,u,f} \tag{10}$$

Not all task versions are present in the final energy consumption estimation. The version selection depends on the scheduling decisions. Even though we skipped this constraint in the above equations for clarity, it is present in scheduling Algorithm 4.3.1. Splitting static and dynamic energy consumption allows us to model DVFS for the three voltage islands on the Odroid-XU4 platform. This concept can be extended to account for all additional DVFS capable compute units or voltage islands.

Like Guo et al. [55], we neither consider Dynamic Power Management (DPM) nor the switching cost of changing the frequency of the voltage islands. DPM refers to a set of techniques that place a system or parts of a system in a low-power sleep mode [37]. These techniques are only beneficial when the idle slot is longer than a certain threshold. The idle time might not be long enough as all CPU cores in a cluster must be idle. Additionally, we already decrease the frequency to the least possible if all cores in a cluster

are idle. We do not consider the frequency switching cost as it is comparable to the cost of context switches in a multitasking environment (i.e., marginal) [106, 123].

4.3 ENERGY-AWARE FORWARD LIST SCHEDULING

Algorithm 4.3.1 Scheduling algorithm

Input: An application DAG composed of a finite set of multi-version tasks (τ) and a non-empty finite list of cores (Cores).
Output: A schedule.

```
1:  function ListSchedule(τ = τ_1, ..., τ_n | τ_x = (v), Cores)
2:      Qready ← ranking(τ)
3:      Qdone ← []
4:      schedule ← new Schedule()
5:      while t ← Qready.pop_front() do
6:          //tmpSched: best schedule for the current task
7:          tmpSched ← new Schedule()
8:          tmpSched.energy ← ∞
9:          foreach v ∈ t.versions do
10:             foreach u ∈ Cores do
11:                 if v runs on u then
12:                     copy ← schedule
13:                     copy.Schedule_task(Qdone, t, v, u)
14:                     copy.Update_energy()
15:                     if copy.energy < tmpSched.energy then
16:                         tmpSched ← copy
17:                     end if
18:                 end if
19:             end for
20:         end for
21:     end while
22:     schedule ← tmpSched
23:     Qdone.push_back(t)
24:     return schedule
25: end function
```

Our proposed heuristic is based on Forward List Scheduling (FLS). FLS first orders the tasks and then adds them one by one to the schedule without backtracking. From here on-wards, we will refer to our energy-aware, multi-

version task scheduling heuristic as *eFLS*. In Section 4.3.1, we sketch out the overall eFLS algorithm. Section 4.3.2 describes how we determine the start time of a given task. In Section 4.3.3, we detail the specifics of our new energy-aware ranking strategy.

4.3.1 *Scheduling Algorithm*

Our proposed scheduling algorithm is sketched out in Algorithm 4.3.1. It uses the taskgraph and a non-empty finite list of cores (Cores) as inputs. The taskgraph contains task objects that contain all necessary versions and DVFS information. First, it ranks the tasks in the DAG into a list (Line 2). Then, a loop iterates over all tasks while tasks that need to be scheduled exist (Lines 5–23). Each task has a non-empty finite set of versions that are all tested on all possible cores (Lines 9–10).

The different task versions also account for different frequencies, i.e., the frequency is a characteristic of v (Line 9). In Line 11 we check if a given version can be executed on the given core (u). After scheduling the specific task version to an appropriate core (Line 13, referring to Algorithm 4.3.2), we compute the energy consumption of the new schedule (Line 14).

The version and mapping resulting in the lowest energy estimation is selected (Lines 15–16). Thus, the selection of the best core and version is greedy. And finally, we add the scheduled task to the list of scheduled tasks Qdone (Line 23). The final schedule is returned when all tasks are scheduled (Line 24).

Proof of termination. We only loop over a finite set of tasks τ, and each task has a finite set of versions. Additionally, the number of Cores is finite. Lastly, all function calls are guaranteed to terminate as they iterate over finite data structures (e.g., Update_energy() iterates over all scheduled tasks.).

Complexity. The complexity of Algorithm 4.3.1 is $\mathcal{O}((n^2 + n \times v) + (n^3) \times v \times u)$, where n is the number of tasks, v is the number of versions and u is the number of cores. The first part of the expression $(n^2 + n \times v)$ comes from the ranking function in Line 2. Algorithm 4.3.1 itself loops over all tasks, versions and cores (i.e., $n \times v \times u$). And lastly the function Sched_task() in Line 13 has a complexity of $\mathcal{O}(n^2)$.

4.3.2 *Scheduling a task.*

Algorithm 4.3.2 sketches out the method to determine the start time of the current task (cur_task). For readability, we describe an algorithm that

assumes one accelerator per accelerator type. However, the algorithm is easily extendable to multiple accelerators per type. Our approach uses an *As Soon As Possible* (ASAP) strategy. Each task must start after its causal predecessors are finished (Line 2), where ρ is the start time of a task, and C is the runtime of a task. Then, while there are changes (Line 4) in the last iteration, we enforce: 1) that there is no overlap between two tasks that require the same core (Lines 6 – 12); 2) that all tasks running at the same time on the same CPU voltage island run at the same frequency (Lines 14 – 23) 3) that there is no overlap between versions that require the same accelerator (Lines 25 – 35).

If any of these three cases happen, the start time (ρ) of the current task (cur_task) is postponed (Lines 9, 18 and 29). Postponing the start of the current task might not be optimal in the case of non-matching frequencies. However, we test more than one frequency. If the current task can be executed at the frequency of the other tasks, this scenario is also explored. The algorithm compares the different alternatives and selects the best. Lastly, the task member attributes are updated with the version attributes (Line 37), and we add the task to the schedule (Line 38).

Proof of termination. Algorithm 4.3.2 is guaranteed to terminate as we only postpone the current task, which can be moved as far as the end of the current schedule. In this case, this task would be executed without any other concurrent task. This ensures that all if-conditions are satisfied and no more changes would be required.

Complexity. The worst case complexity of Algorithm 4.3.2 is $\mathcal{O}(n^2)$. Let us consider an application with n tasks, and all tasks but one have been scheduled onto a single core system. In the worst-case scenario the last task, t_n, has to be moved past all previously scheduled tasks. If t_n is only postponed once per iteration of the while loop, we need a total of n^2 iterations.

4.3.3 *Heterogeneous Energy Ranking*

Our new energy-aware ranking strategy, *Heterogeneous Energy Ranking* (HER), builds on the well-known HEFT [131] ranking strategy and has been developed to generate rankings that work well for energy-aware scheduling. HER is one ranking strategy that can be used in Algorithm 4.3.1 Line 2.

Algorithm 4.3.3 sketches out the HER algorithm. It uses the DAG (τ) and a non-empty finite set of cores ($Cores$) as inputs, builds the ready queue ($Qready$) and returns it. First, it traverses the graph in a reverse fashion

Algorithm 4.3.2 Scheduling of a task

Input: Finite list of scheduled tasks (Qdone), Current task to schedule (cur_task) and its version (version), Current core (cur_core).
Output: Add a new task to the schedule.

```
1:  function ScheduleTask(Qdone, cur_task, version, cur_core)
2:      cur_task.ρ ← max_{x∈predecessors(cur_task)}(x.ρ + x.C)
3:      Change ← True
4:      while Change do
5:          Change ← False
6:          foreach t ∈ Qdone do
7:              if t is mapped on cur_core then
8:                  if t overlaps in time with cur_task then
9:                      cur_task.ρ ← t.ρ + t.C
10:                     Change ← True
11:                 end if
12:             end if
13:         end for
14:         foreach t ∈ Qdone do
15:             if t is mapped on cur_core.volt_island then
16:                 if t overlaps in time with cur_task then
17:                     if t.freq ≠ version.freq then
18:                         cur_task.ρ ← t.ρ + t.C
19:                         Change ← True
20:                     end if
21:                 end if
22:             end if
23:         end for
24:         if version.requires_coprocessor() then
25:             foreach t ∈ Qdone do
26:                 if t.requires_coprocessor() then
27:                     if t.coprocessor() == version.coprocessor() then
28:                         if t overlaps in time with cur_task then
29:                             cur_task.ρ ← t.ρ + t.C
30:                             Change ← True
31:                         end if
32:                     end if
33:                 end if
34:             end for
35:         end if
36:     end while
37:     cur_task.update(version)
38:     this.add_task(cur_task)
39: end function
```

(Line 2), i.e., starting with the last task(s). In this case, the traversal algorithm does not matter, as long as the partial order of the DAG is maintained. For each task, we calculate the tasks energy consumption (Line 3) using the version aggregator ($version\_agg$). Additionally, we calculate the energy consumption of the successors ($succ\_agg$, Line 4).

The tasks total energy consumption for ranking purposes is then equal to the sum of the version energy consumption ($version\_energy$) and the successor energy consumption ($succ\_energy$, Line 5). Lastly, the $Qready$ list is built (Line 7) and returned (Line 8). The ready queue is ranked based on the energy consumption of each task $\tau_i.energy$, starting with the maximum energy consumption, i.e., the first task has the highest energy consumption and thus will be scheduled first.

Proof of termination. We only loop over a finite set of tasks, τ. All tasks have a finite number of versions and a finite number of successors.

Complexity. The worst case complexity of Algorithm 4.3.3 is $\mathcal{O}(n^2 + n \times v)$, where n is the number of tasks in the DAG. We loop over all tasks and all successors of a task (n^2). Additionally, we loop over all versions for all tasks, $n \times v$.

Algorithm 4.3.3 Heterogeneous Energy Ranking

Input: An application DAG composed of a finite set of multi-version tasks (τ) and a non-empty finite set of cores ($Cores$).
Output: A sorted ready queue ($Qready$).

```
1: function HER_SORT(τ = τ1, ..., τn|τx = (v), Cores)
2:     for τi ∈ linearise(reverse(τ)) do
3:         version_energy ← version_agg(τi, Cores)
4:         succ_energy ← succ_agg(successors(τi))
5:         τi.energy ← version_energy + succ_energy
6:     end for
7:     Qready ← Sort(τ,τi.energy)
8:     return Qready
9: end function
```

Version Aggregator. Determining a task's energy consumption is not straightforward, as a task can have many different versions with different target compute units, functionally equivalent implementations and different DVFS settings. Thus, we must aggregate the energy information from all different task versions.

A version's energy consumption could either be only the dynamic energy consumption (i.e., E_d in Equation (6)) or it could be the total energy con-

sumption (i.e., E in Equation (6)). An argument for using only the dynamic energy is that the static energy (E_s) is heavily dependent on the rest of the system (e.g., other tasks being executed). On the other hand, using only the dynamic energy component ignores that a long-running version could have significantly higher energy consumption than a shorter version, despite having low dynamic energy consumption. Thus, we investigate both static and dynamic energy consumption during the version aggregation phase.

Next, we need to tackle how the energy consumption of different versions is combined/aggregated. The following six different options are explored in Section 4.5: *minimum*, *average*, *sum*, *variance*, *minimum + variance* and *minimum + standard deviation*. *Minimum* is the natural choice for an aggregator because we want to minimise energy consumption. *Average* and *sum* contain more information about the other versions; this could increase a task's priority if the minimum version cannot be picked due to interference with previously scheduled tasks. The reason to consider *variance* is that it increases the priority of tasks with significant variations in energy consumption between versions. Thus, this prioritises tasks that have low minimum versions but where the alternative versions have a much higher energy consumption. Combining minimum and variance/standard deviation allows both energy-intense tasks and tasks with high variance/standard deviation to be scheduled first. Using standard deviation over variance might be helpful as the variance could be larger than the minimum energy consumption and thus overshadow the minimum. Hence, when using the dynamic minimum aggregator (*Dyn. min.*), we use the lowest dynamic energy consumption across all versions of a specific task. In total, this leads to twelve different version aggregators.

Let us look at an example to clarify the meaning of the different aggregators and their advantages. Table 4.1 lists the energy consumption of five different tasks. Each task has at least two versions and, at most, six versions. Let us assume that all tasks can be scheduled and that all we need to do is to determine the scheduling order. Table 4.2 shows the different orders depending on the used version aggregator. Thus, in the case of using the *minimum* aggregator, task four (*T-4*) would be scheduled first as it has the highest minimum energy consumption (19). Task three has a low priority when employing *minimum*, *average*, and *sum* aggregators but has a higher priority when using one of the *variance* aggregators due to the significant difference between versions ($variance = 5.2$). Each aggregator leads to a slightly different scheduling order.

Table 4.1: Hypothetical energy consumption of different task-versions.

	T-1	T-2	T-3	T-4	T-5
Version 1	15	11	15	20	12
Version 2	12	15	10	19	30
Version 3	16	16	16	n.a.	16
Version 4	13	17	18	n.a.	18
Version 5	14	15	n.a.	n.a.	2
Version 6	14	n.a.	n.a.	n.a.	11

Table 4.2: Hypothetical scheduling order of different tasks depending on the used version aggregator.

	T-1	T-2	T-3	T-4	T-5
Min	2	3	4	1	5
Average	5	3	4	1	2
Sum	2	3	4	5	1
Variance	4	3	2	5	1
Min+Var	5	4	2	3	1
Min+Std	2	4	3	1	5

Successor Aggregator. We consider two options for the successor aggregator: *maximum* or *sum* of all successors. Using the *maximum* ranks tasks by the largest energy path. Using the *sum* gives higher importance (i.e., earlier scheduling slot) to tasks with many successors.

Combining the two successor aggregators with the twelve version aggregator options results in a total of 24 different HER ranking methods. Together with four base ranking methods, we are considering a total of 28 ranking methods. The base ranking methods are: DFS with WCET as a tie-breaker, BFS with WCET as a tie-breaker, BFS with laxity as a tie-breaker and BFS with energy laxity as a tie-breaker. The initial ranking algorithm has a massive impact on the schedule. Hence, we explore all 28 possible ranking methods.

4.4 EXPERIMENTAL SETUP

We describe the target hardware in Section 4.4.1, followed by Section 4.4.2, which details our energy measurement setup. Then we introduce the task-graph generation (Section 4.4.3) that we use throughout Sections 4.5 to 4.7. At last, we explain our DVFS approach for the target platform (Section 4.4.4). This experimental setup is used throughout the rest of the chapter.

4.4.1 *Target Platform*

Our approach can be used on a wide variety of target platforms, as described in Section 4.2. However, for the sake of illustration and concrete experimental validation, we focus on the Odroid-XU4 board from here onward.

The Odroid-XU4 [59] platform is based on the ARM big.LITTLE CPU architecture [1] complemented by a Mali GPU accelerator [49]. The CPU is an Exynos 5 Octa 5422 chip [49], which embeds two clusters of four cores each. One cluster includes energy-efficient in-order Cortex-A7 cores (LITTLE), while the other provides high-performance, out-of-order, deep-pipeline Cortex A-15 cores (big). Each cluster forms a separate voltage island, i.e., the voltage and frequency can only be changed for all cores in a cluster at the same time. The two core types are ISA-compatible. Each core has its own L1 cache, and each cluster has a shared L2 cache (512KB for LITTLE cores, 2MB for big cores). The Mali GPU features six shader cores and shares the off-chip physical memory with the CPU, thereby avoiding common data transfer overhead between the CPU and GPU. The GPU is a third voltage island independent from the CPU.

4.4.2 *Energy Measurements*

Energy consumption is measured with the Otii system by Qoitech[1]. It is a non-intrusive high-side[2] power monitor with a sampling rate of 4kHz. The Otii has a maximum measurement error of $0.1\% + 150\mu A$. Thus, at the lowest measured current of 0.36A, we can expect a maximum error of 0.104%. Therefore, we use the measurement setup introduced in Chapter 3.

1 https://www.qoitech.com/products/standard

2 "High-side" refers to the placement of the shunt resistor used for current sensing. "High-side" means that the shunt resistor is placed between the positive supply and the load. This is in contrast to "low-side", where the current-sense resistor is placed between the load and ground.

Our target Odroid-XU4 platform communicates the start of software tasks via UART to the Otii at 115200bps. To ensure fair measurements, we take the following steps:

- We do not consider the fan's energy consumption, which is powered via a separate device.
- We calibrate the Otii device before the experiments.
- We warm up the involved devices by executing tasks before the actual experiments.
- We measure execution time, power and energy to solution (EtoS) at the same time.
- We run each experiment 50 times to obtain enough data for statistical testing and extracting worst-case measurements.

4.4.3 *Application code*

We are not aware of any high-performance IoT focused benchmarks. However, we would still like to execute real compute-heavy workloads on actual hardware. Therefore, we are building our own set of synthetic DAG applications based on Taskgraphs For Free (*TGFF*) [41], and the Rodinia benchmark suite [27]. To build the structure of the graph, we rely on TGFF. Additionally, TGFF allows us to assign a random number to each node in the graph. We use this feature to assign a random task type to each node. Then, we *a-posteriori* perform a mapping between *TGFF* task types to Rodinia benchmark. Let us assume that *TGFF* generates the graph in Figure 4.6. The edges represent data relationships between nodes (see Section 4.2.2), but the exact type of data is irrelevant for this clarification. Task 2 and Task 3 are pseudo-randomly assigned types 3 and 6, respectively. Types 3 and 6 are mapped to the *nn* and *nw* Rodinia benchmarks, respectively. Tasks 1 and 4 are two special task types, namely the source and sink nodes, representing some housekeeping tasks. There is only one source and sink node in each graph. Thus, the final application is represented by Figure 4.7. This results in an extensive collection of randomly generated taskgraphs with executable code. In order to compare different scheduling methods, we generate random DAG based applications.

Each task includes all IO, overhead and computations required for the Rodinia benchmark. We work with the following 8 benchmark tasks: *heartwall*,

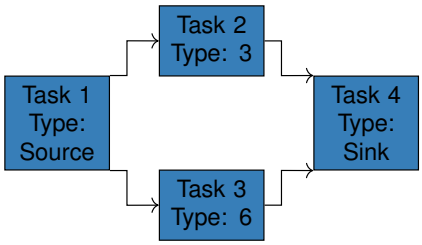


Figure 4.6: A *TGFF* generated graph with randomly assigned task types. The types are used to map random nodes to executable code in the form of Rodinia benchmarks.

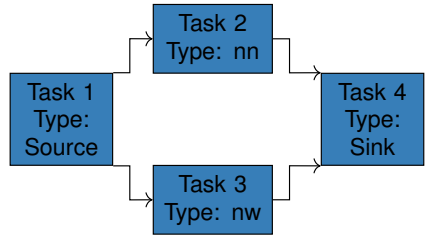


Figure 4.7: The randomly assigned task types of each node are mapped to Rodinia benchmarks. This allows us to execute randomly generated DAG applications with different computational characteristics.

hotspot, *k-means*, *lud*, *nn*, *nw*, *bfs* and *srad_v1*. We use these benchmarks because only these could be executed on the Odroid-XU4 with minor changes. These changes, among others, ensure thread safety when multiple instances of the same benchmark are run concurrently. The Rodinia benchmark suit does not provide a sequential implementation of the benchmarks. Therefore, we execute the OpenMP implementations on a single thread (i.e., set environmental variable *OMP_NUM_THREADS* to 1). Additionally, we use the OpenCL implementations of six of these benchmarks to demonstrate multi-version task scheduling. A similar approach was taken by De Bock et al. [39], who used TACLe benchmarks [50] for independent tasks. Links to the dataset and code used in this chapter can be found on page 137.

The energy and timing information required for our approach can be obtained with different methods, such as measurements or static analysis. In this chapter, we obtain the information needed through measurements. The dynamic energy consumption and timing measurements were collected simultaneously and in isolation for each task. The measurements cover all possible execution paths of the tasks, and we selected the worst observed values as the WCET estimates. To further increase the safety of our estimations and to account for contention, we increased the observed WCET by an arbitrarily chosen safety margin of 30%. The Rodinia tasks are straightforward computational tasks. Each execution of one task was done with the same input data, which determines the computational intensity and execution path.

4.4.4 *DVFS*

Previous papers [19, 31, 40] have shown that the energy consumption of an application with respect to the frequency follows a convex curve, i.e., the lowest energy consumption of an application is achieved at a mid-level frequency. A lower frequency results in long run times and, therefore, higher static energy consumption. A higher frequency results in a shorter run time, but the voltage increase required by the frequency increase offsets the shorter run time [19, 31, 40].

We find similar convex behaviour on the big cores for all selected Rodinia benchmarks. The minimum energy consumption is always achieved between 1.3GHz and 1.6GHz. Therefore, we consider all frequencies between 1.3GHz and 2GHz (i.e., the maximum clock frequency possible), allowing for a good trade-off between energy consumption and run time. On the LITTLE cores, the convex behaviour is not as pronounced as on the big cores (i.e., increasing the frequency from 1.3GHz to 1.4GHz does not increase the energy consumption much). The minimum energy consumption is achieved between 1.3GHz and 1.5GHz (i.e., the maximum clock frequency possible). Hence, we consider all frequencies between 1.3GHz and 1.5GHz. For the GPU, we consider all supported frequencies (177MHz to 600MHz).

4.5 RANKING STRATEGIES FOR EFLS

Greedy-, evolutionary- and genetic-algorithms for scheduling of dependent tasks rely on establishing a total order of tasks instead of a partial order as is obtained from a Directed Acyclic Graph (DAG) (e.g.[78, 94, 116, 133]). The exact ranking algorithm used can have a significant impact on the final schedule. Previous research [116] showed that no ranking algorithm consistently outperforms the others when minimising a schedules makespan. Hence, previous research schedules all DAGs using multiple ranking methods and then picks the best resulting schedule [116]. The ranking strategy which will perform best is unknown before scheduling.

We investigate the impact of different ranking algorithms in an energy-aware scheduling setting. For this purpose, we generate 500 taskgraphs with TGFF [41], with an average of 124.8 tasks ($minimum = 34$ and $maximum = 298$). All 500 taskgraphs are scheduled using all 28 different ranking algorithms (see Section 4.3.3). In Section 4.5.1, we investigate which standalone ranking leads to the best schedules with respect to energy consumption. Then in Section 4.5.2, we explore the scheduling time and

energy-reduction trade-off when selecting a sub-set of the different ranking algorithms. The sub-set selection is important as computing all 28 schedules is highly time-consuming.

4.5.1 *Best standalone ranking*

Figure 4.8 compares the relative energy efficiency of the schedules that result from the different ranking algorithms. *BFS with WCET* ranking was chosen as the baseline and is at 0%. A negative value means that a given ranking strategy resulted in a better schedule (lower predicted energy consumption) than *BFS with WCET*. The HER based ranking that combines average version dynamic energy consumption with a maximum successor aggregator (*Dyn. avr. max*) results in the best schedules on average. It improves the average predicted energy consumption by almost 5%. Additionally, it also outperforms the base ranking methods slightly (0.2%). The base ranking strategy combines *(BFS) Laxity*, *(BFS) Energy Laxity*, *BFS (with WCET)* and *DFS (with WCET)*. The classic *DFS* ranking results in schedules that are, on average, the worst with more than 15% higher predicted energy consumption.

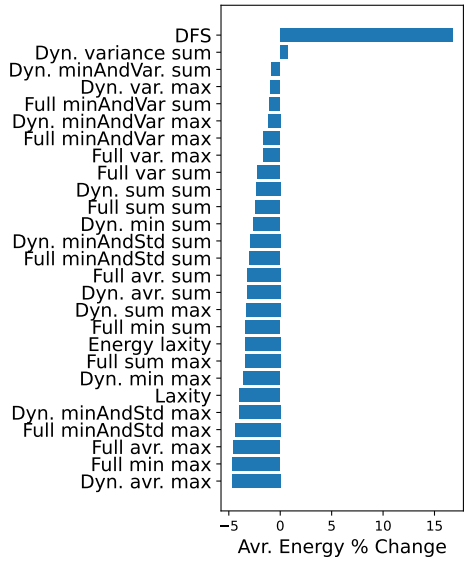


Figure 4.8: Average energy estimation increase (%) across different ranking algorithms compared to the energy consumption when ranking DAGs with *BFS*. Thus, using Dyn. avr. max results in a 5% decrease on average in comparison to BFS.

The five best ranking methods are all HER based and use the *max* successor aggregator. They feature the *average*, *minimum* and *minimum + standard deviation* version aggregators in conjunction with both the dynamic and the full energy consumption. The two best ranking methods, with respect to energy consumption, are not significantly different from each other (paired sample t-test, p-value= 0.73). All other ranking methods differ considerably from the best ranking method ($\alpha = 0.01$).

On average, the methods based on summing successors perform worse than the ones based on the maximum successor aggregator. All variance-based methods perform worse than non-variance-based HER ranking methods. The variance seems to play a too significant role in the ranking, i.e., overshadowing the minimum. However, most variance-based ranking algorithms still outperform BFS and DFS ranking. The laxity-based method is ranked the sixth best, and the energy laxity method takes the ninth spot.

The average across all 500 DAGs does not tell the full story. Looking at the percentage of DAGs where a ranking method results in the best schedule (Figure 4.9), with respect to predicted energy consumption (PEC), tells a slightly different story than Figure 4.8. The first four ranking methods that result in most schedules with the lowest PEC are *Dyn. avr. max*, *Full min max*, *Full MinAndStd max* and *Full avr. max*. Therefore, quite similar to the results of Figure 4.8 with only minor shifts in ranking. The bigger surprise is the fifth-best ranking method with respect to the percentage of DAG schedules with the lowest PEC. *Dyn. var. max* is ranked 24th in Figure 4.8 with respect to average predicted energy consumption (3.9% worse on average than *Dyn. avr. max*). However, in over 8% of the cases, it results in the best schedule. In total, 24 out of the 28 rankings result in at least one schedule that is the best compared to all other rankings. From here onwards, we will only consider these 24 rankings. Hence, similar to [116], it shows that no ranking algorithm always yields the best schedule.

Lastly, Figure 4.10 shows the average energy consumption improvement of a given ranking over the *Dyn. avr. max* ranking if the given ranking resulted in a better schedule. Additionally, it shows the number of DAGs for which a given ranking achieved a better result. So *Dyn. avr. sum* results in a better schedule in 114 cases, and these schedules have 1.1 % lower predicted energy consumption compared to the schedules resulting from the *Dyn. avr. max* method. The energy consumption improves by, on average, 1.3%. The following four ranking methods improved more than 160 graphs: *Dyn MinAndStd max*, *Full avr. max*, *Full min max* and *Full MinAndStd max*. Again

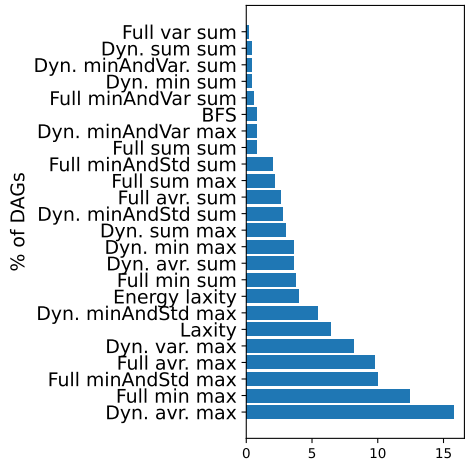


Figure 4.9: The chart shows the percentage in which case a given ranking is the best across all 28 ranking methods across all DAGs. Thus, Dyn. avr. max results in the best ranking in over 15% of all DAGs.

the one outlier is *Dyn. var. max*, which improved its 68 DAGs by an average of 4.4%.

4.5.2 *Selecting a ranking method sub-set*

Trying all 24 ranking methods and selecting the best schedule takes significant time. We aim to select a sub-set of ranking methods to reduce the scheduling time while producing low-PEC schedules. We start with the best standalone ranking method (*Dyn. avr. max*). Then, we add the next best ranking method one after another to the sub-set of ranking methods until all 24 methods are used. This results in Figure 4.11, which shows how the predicted energy consumption of all 500 DAGs improves when adding additional ranking algorithms to the sub-set. Thus, when scheduling all 500 DAGs with two HER ranking methods (*Dyn. avr. max* and *Full min. max.*), we can decrease the average predicted energy consumption by 0.53% in comparison to just using *Dyn. avr. max*. The second-ranking method added to the sub-set is the *Dyn. var. max.* method. It only improves the schedules of 68 DAGs, but these 68 schedules are improved significantly (see Section 4.5.1).

Adding ranking methods to the sub-set significantly changes (at alpha = 1%) the scheduling results until the 17th addition. When adding the 17th ranking method (i.e., *BFS*), the average predicted energy consumption of a schedule is decreased by 1.53%, and the p-value is 0.16 (i.e., not significant).

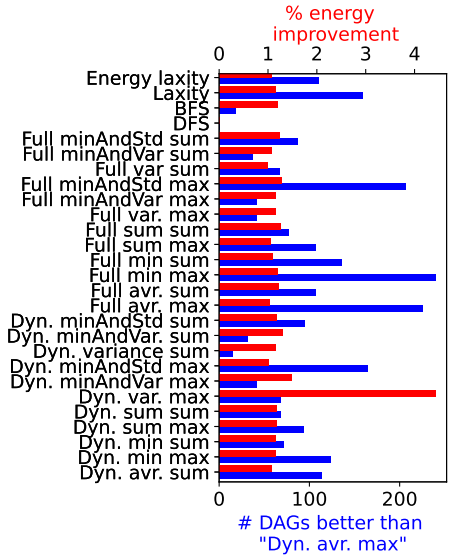


Figure 4.10: The bottom y-axis shows the number of DAGs that were improved by a given ranking algorithm over *Dyn. avr. max.*. The top y-axis shows the energy consumption improvement in percentage for the DAGs that improved. Thus, Dyn. var. max improved 68 DAGs by an average of 4%.

Using all 24 ranking methods results in an average 1.54% predicted energy consumption decrease compared to the schedules resulting from just using the *Dyn. avr. max* ranking. Additionally, it decreases the energy consumption by 2.02% in comparison to the base ranking methods (*DFS*, *BFS*, *BFS with laxity*, *BFS energy laxity*).

By using only six ranking methods (i.e., using 25% of ranking methods), we achieve over 80% of the possible improvements (i.e., 1.26%) and a 1.7% improvement over the base set. In the rest of the paper, we will make use of the first six ranking methods added to the sub-set (*Dyn. avr. max*, *Full min. max.*, *Dyn. var. max.*, *Full avr. max.*, *Full minAndStd max.*, *Laxity*). We use only six ranking methods as this represents a good trade-off between scheduling time and PEC. Using a larger sub-set of scheduling algorithms improves the result further but also significantly increases the time required to schedule a DAG. This is especially prohibitive considering that scheduling techniques are becoming more complex to take full advantage of heterogeneous architectures (e.g., [67, 151]).

Figure 4.12 compares the base set of ranking methods (*DFS*, *BFS*, *BFS with laxity*, *BFS energy laxity*) against the aforementioned set. It clearly shows that

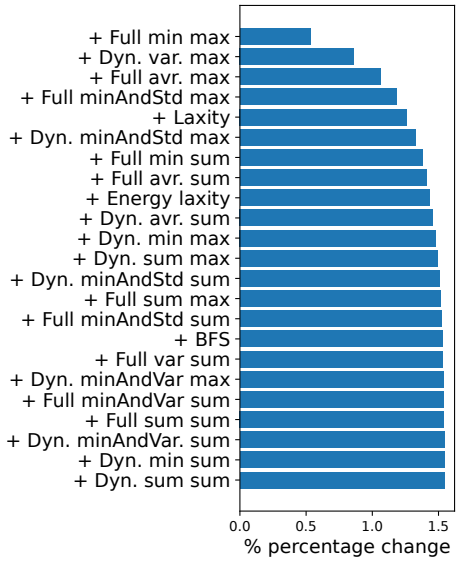


Figure 4.11: The chart shows the average percentage increase when adding one ranking after another to the ranking sub-set. Adding ranking algorithms leads to statistically significant energy consumption improvements, including the 16th ranking method. Adding the 17th method does not lead to significant improvements.

the new sub-set results in up to 23% lower energy consumption, and at most, the new sub-set results in 2.3% higher PEC, which we deem an acceptable trade-off. Additionally, 80.6% of new schedules are better, 10.2% of schedules stayed the same and in 9.2% of the DAGs, the resulting schedule use more energy.

In conclusion, we can demonstrate that HER-based schedules perform well. The best standalone HER ranking algorithm produces schedules that perform on average as well as the schedules of the base set of rankings but reduces the scheduling time by 75%. Alternatively, when selecting a ranking sub-set of the six best ranking algorithms, we can decrease the energy consumption of the generated schedules by up to 23%. Furthermore, we provide more evidence that not a single ranking method consistently outperforms all others (e.g., Figure 4.12). Of course, we could remedy this by using all known rankings; however, there are always more rankings that might perform better in a given scenario. Thus, while we show that improvements with respect to the resulting schedule are possible (and can be pretty significant), this also indicates that future work should focus on ranking agnostic scheduling methods.

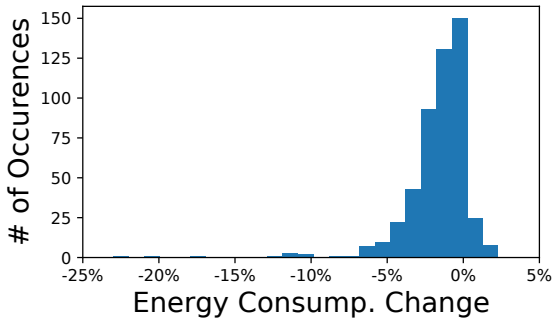


Figure 4.12: Histogram of the energy consumption reduction when using the new sub-set of ranking methods compared to the base set of ranking algorithms.

4.6 COMPARING EFLS WITH OTHER NON-OPTIMAL SOLUTIONS

To validate our proposed approach, we conduct a series of experiments. For this purpose, we use the same 500 taskgraphs as in Section 4.5. First, Section 4.6.1 compares a multi-version task eFLS approach to a single-version task eFLS approach. Next, we contrast our eFLS heuristic to a multi-version task strategy based on classic FLS (Section 4.6.2). In Section 4.6.3, we compare our eFLS approach against HEFT [131] and a version of HEFT that we modified to be energy-aware. Last, we scrutinise if the eFLS heuristic performs as well as an evolutionary meta-heuristic-based scheduler (Section 4.6.4).

4.6.1 *Single-version vs Multi-version tasks*

We compare our multi-version task eFLS scheduler against a single-version task eFLS scheduler. The single-version task scheduler is the same as the multi-version task scheduler (i.e., the single-version scheduler also uses all six ranking strategies); however, it has access to only the CPU version or the GPU version of a task. The version that is available to the single-version scheduler is the version that has the lowest energy consumption in isolation (i.e., we pick the most energy-optimal version before scheduling).

If the single-version eFLS scheduler only has access to the CPU versions. In that case, the multi-version task eFLS solutions are, on average, 16.0% more energy efficient and have a lower makespan than the single-version task eFLS solutions. The multi-version task solutions are up to 31.4% more

energy efficient and are at least 8.3% more energy efficient. The histogram of energy consumption reductions is shown in Figure 4.13.

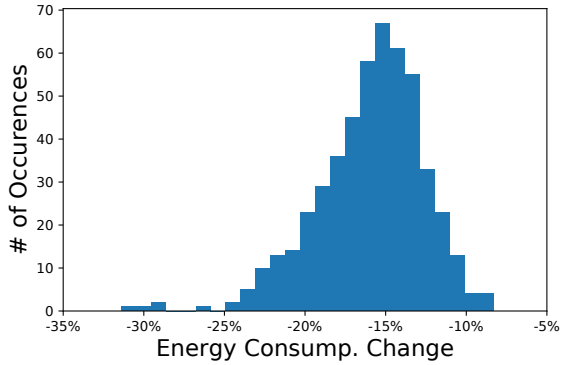


Figure 4.13: Histogram of the energy consumption reductions when providing multiple versions to the eFLS scheduler.

These results demonstrate that including the GPU versions is beneficial with respect to energy consumption. One might argue that instead of using the CPU versions for all tasks, using only the GPU version for some tasks will be more beneficial. However, it is unclear before scheduling which task should be executed on the GPU as this differs per task graph. For example, the *k-means* tasks are scheduled on the GPU in 24.2% of the cases, and the *LU decomposition* tasks are scheduled on the GPU in 77.0% of the cases. Combining this knowledge with the fact that there is only one GPU (i.e., GPU tasks have to run sequentially after each other), it is not surprising that providing the *k-means* and *LU decomposition* GPU versions only to the single-version eFLS scheduler does not improve the situation. Our results show that the multi-version schedules are, on average, 12.2% more energy efficient than the single version approach with the *k-means* and *LU decomposition* GPU versions only. Thus, including multiple versions and letting the scheduler pick the best one has clear advantages. Decreasing energy consumption by more than 12% could be a game changer for battery-operated devices.

4.6.2 *Energy optimising vs Makespan*

Next, we compare our multi-version task eFLS scheduler against a multi-version task energy-unaware scheduler. We use the same FLS approach as introduced in Section 4.3; however, instead of selecting the best version based on energy consumption, we select the best version based on run

time. This way, we mimic existing methods, e.g., [116]. The energy-unaware scheduler is only based on DFS with WCET, BFS with WCET and BFS with laxity ranking, as HER is an energy-specific ranking strategy. Additionally, in Section 4.6.3 we compare against HEFT [131].

As the FLS schedule focuses purely on minimising makespan, it takes full advantage of the high clock speeds available on the Odroid-XU4, thereby increasing the voltage to the maximum. Thus, it is not surprising that the eFLS generated schedules consume, on average, 26.4% less energy than the FLS generated schedules, with a standard deviation of 3.2%. The eFLS solutions are always more energy efficient. They are at least 14.0% more energy efficient and at most 34.4%. The makespan focus (and thus high frequency) means that the FLS solutions result in a makespan that is, on average, 16.8% lower than the makespan of the eFLS solutions. Therefore, we can show that our approach to minimising energy consumption is valid.

4.6.3 *eFLS vs HEFT and eHEFT*

We compare our eFLS approach with the well-known HEFT scheduler [131]. For a fair comparison, we added multi-version, DVFS and GPU-scheduling capabilities to HEFT. Similar to the makespan FLS method in Section 4.6.2, HEFT focuses purely on minimising the makespan. Therefore, it is not surprising that the estimated energy consumption of the generated schedules is, on average, 26.3% higher than the estimated energy of the eFLS schedules. Additionally, it is estimated that the HEFT schedules use at least 19.6% and, at most, 38.9% more energy. The histogram of energy consumption change when moving from HEFT to eFLS is shown in Figure 4.14.

To level the playing field between HEFT and eFLS, we changed the decision-making in HEFT to use energy consumption instead of the earliest finish time. The energy-aware HEFT (*eHEFT*) strategy is only different in its ranking strategy compared to eFLS. eHEFT offers a significant improvement over standard HEFT with respect to the energy consumption of the generated schedules.

Overall the eHEFT schedules result in 1.74% higher estimated energy consumption than eFLS which is significantly different (p-value$= 5.35^{-66}$). Figure 4.15 compares the energy consumption of schedules generated by eFLS to the energy consumption of schedules generated by eHEFT. While eFLS outperforms eHEFT on average, there are a few DAGs where eHEFT performs better. The best eHEFT generated schedule results in 2.2% lower estimated energy consumption than their eFLS counterpart. On the flip side,

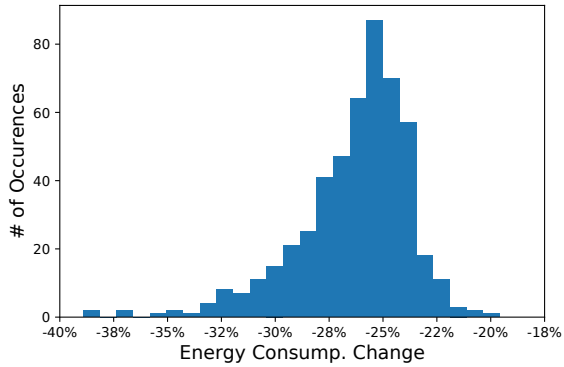


Figure 4.14: Histogram of the energy consumption reductions of our eFLS approach over HEFT.

the best eFLS generated schedules result in 23.0% lower estimated energy consumption than the eHEFT counterpart. If we repeat the sub-selection experiment from Section 4.5.2, eHEFT would only be the ninth ranking strategy that is added. Thus, the imbalance seen in Figure 4.15 and its low importance in the sub-selection experiment means that we do not add eHEFT to our ranking strategy selection.

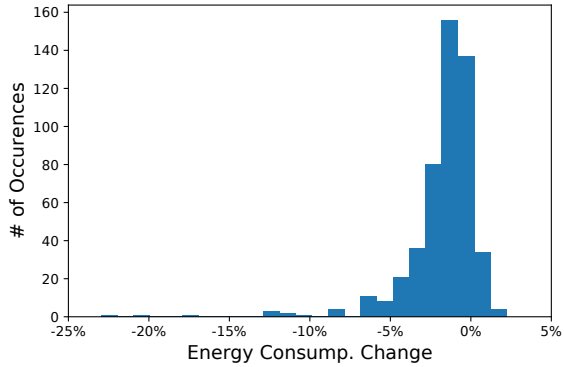


Figure 4.15: Histogram of the energy consumption reductions of our eFLS approach over eHEFT.

4.6.4 *eFLS vs ARSH-FATI*

As shown by Sheikh and Pasha [123], heterogeneous energy-efficient scheduling methods have primarily focused on optimising the energy consumption

of multiple independent tasks. The only exception we know of is the ARSH-FATI algorithm proposed by Ullah Tariq et al. [133]. The authors researched energy-efficient static scheduling for DAG taskgraphs. The paper focuses on Smart Networked Systems, and experimental results are simulation-based. The paper explores scheduling, mapping and DVFS based on population heuristics. The solutions generated by their approach are 24% more energy efficient than CA-TMES-Search and 30% more energy efficient than CA-TMES-Quick [58].

Their approach only selects the per voltage island DVFS once, whereas our approach can switch between frequency levels at runtime. Additionally, the approach by Ullah Tariq et al. neither differentiates between task versions nor considers the GPU. We implemented the proposed meta-heuristic from the description in their paper and used it to schedule the same taskgraphs as in the previous parts of Section 4.6. Besides the population size, we used the same hyperparameters as determined in [133]. The population size in the original paper was limited to 5; increasing the population size to 500 improved the performance of the meta-heuristic significantly. We restrict the solving time of ARSH-FATI to 6 hours. If the scheduling takes longer than 6 hours, we still obtain a schedule; however, it might not be the best possible schedule.

We compare eFLS against ARSH-FATI across three dimensions: scheduling time, makespan, and energy consumption. Figure 4.16 compares the solving time required for the ARSH-FATI algorithm and eFLS. The eFLS solving time equals the sum of all six ranking methods. The largest DAG that ARSH-FATI solved within the 6-hour time limit comprised 174 tasks. eFLS schedules the largest DAG with 299 tasks in under 85 minutes. Extrapolating the ARSH-FATI solving time indicates that ARSH-FATI would require more than 17 hours to finalise scheduling the DAG of 299 tasks.

Next, Figure 4.17 compares the length of the resulting schedules. We can see that ARSH-FATI results in longer schedules than eFLS. The DAGs for which ARSH-FATI exceeded its solving time continue the trend of the DAGs that were scheduled within the time limit. Thus, it is unlikely that ARSH-FATI would perform better with a higher time limit. Overall the schedules generated with eFLS are 46.7% shorter than those generated with ARSH-FATI.

Figure 4.18 compares the predicted energy consumption of eFLS and ARSH-FATI schedules. It shows a similar trend as Figure 4.17. The gap between the two sets of schedules is smaller for DAGs of smaller size.

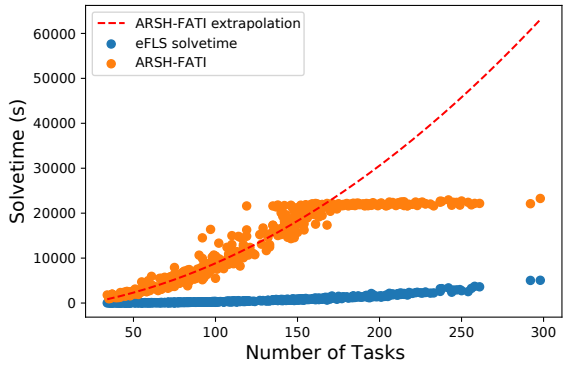


Figure 4.16: Graph comparing the solving time required for eFLS and ARSH-FATI. The graph additionally shows an extrapolation of the ARSH-FATI solving time, which was capped at 6 hours.

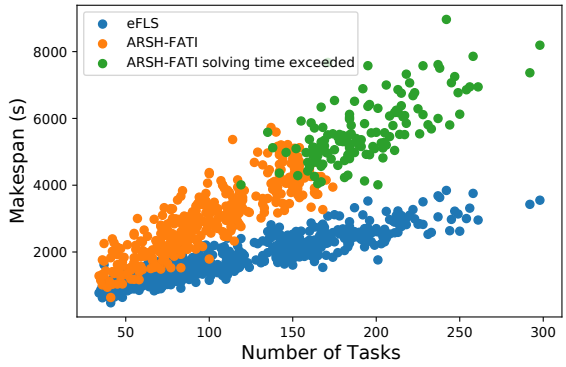


Figure 4.17: Comparing the makespan resulting from the eFLS and ARSH-FATI schedulers.

Figure 4.19 shows the improvement of our approach over the ARSH-FATI meta-heuristic. Our approach produces schedules that are, on average, 27.8% more energy efficient, at most 43.7% more efficient and at least 15.5% more energy efficient. Considering only the DAGs that ARSH-FATI completed within the time limit does not change the results significantly. The eFLS generated schedules still reduce the predicted average energy consumption by 27.5%.

Figure 4.18 indicates that ARSH-FATI performs slightly better on smaller taskgraphs. An additional experiment with smaller taskgraphs (average of 9.78 tasks) suggests that the gap between ARSH-FATI and eFLS is smaller. However, the eFLS scheduler still produces solutions that are, on average,

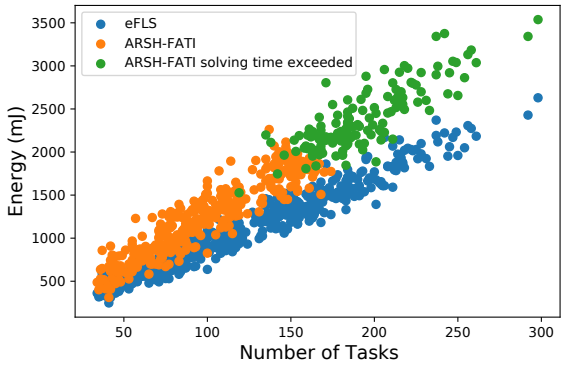


Figure 4.18: Energy consumption reduction of eFLS vs ARSH-FATI with respect to the number of tasks in a taskgraph.

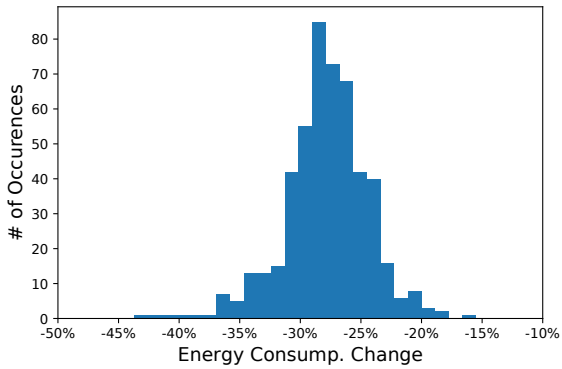


Figure 4.19: Histogram of the energy consumption reductions of our eFLS approach over the ARSH-FATI meta-heuristic.

17.1% more energy efficient. The significant difference between eFLS and ARSH-FATI shows that our approach can explore the optimisation state-space better than the current state-of-the-art approach, both with respect to makespan and energy consumption, at significantly lower scheduling times.

In conclusion, our experiments show that: multi-version outperforms single-version, our approach to energy scheduling is valid, and our approach is better at handling the optimisation space than two current state-of-the-art approaches.

4.7 COMPARING EFLS WITH AN OPTIMAL SOLUTION

Heuristic algorithms return approximate solutions by nature. Determining the over-approximation ratio requires comparing the results of the heuristic with those of an exact method. Fortunately, for scheduling problems, it is possible to generate optimal solutions using an Integer Linear Programming (ILP) formulation. However, solving ILP scheduling problems is an NP-hard problem and thus does not scale well with an increasing number of tasks. To estimate the over-approximation of our eFLS heuristic, we also introduce an ILP-based scheduler summarised in Section 4.7.1. Then we compare the ILP generated solutions to the eFLS generated solutions in Section 4.7.2. At last, we show that the predicted energy consumption matches the measured energy consumption (Section 4.7.3).

4.7.1 *ILP formulation*

An ILP formulation consists of a set of constraints that must be satisfied and an objective function that will be optimised. In some of the following constraints, logical operators $\vee$ and $\wedge$ are used for clarity; these operators can be linearised using [25].

Objective function. Our goal is to minimise the energy consumption over all tasks of an application as formalised by Equation (11). The energy consumption E of a schedule is equal to the sum of the static energy consumption and the dynamic energy consumption, as stated by Equation (6) in our energy model. Note that Equation (9) is present twice, once for the energy consumption of the CPU voltage islands ($time_{f,i}$) and once for the accelerator voltage islands ($acctime_{f,x}$).

Map a task to a core. Equation (12) ensures that task p is mapped on one and only one core u ($m_{p,u} = 1$). Equation (13) indicates if two tasks, p and q, are assigned to the same core $s_{p,q} = 1$.

$$\begin{aligned} &minimise\ E = makespan \times W_{average} + \sum_{i \in I} \sum_{f \in F_i} time_{f,i} \times W_{i,f} \\ &+ \sum_{x \in Acc} \sum_{f \in F_x} acctime_{f,x} \times W_{x,f} + \sum_{p \in \tau} E_p \end{aligned} \tag{11}$$

$$\sum_{u \in Cores} m_{p,u} = 1, \forall p \in \tau \tag{12}$$

$$s_{p,q} = \sum_{u \in Cores} m_{p,u} \wedge m_{q,u}, \forall (p,q) \in (\tau \times \tau), p < q \tag{13}$$

Prevent overlap on the same core. If two tasks are mapped to the same core, variable o determines the order of tasks p and q, $o_{p,q} = 1$ means p is scheduled before q. Thus, Equation (14) enforces that two tasks are executed in a given order, and only one of the two orders is possible at once.

Equation (15) prevents time-wise overlap of two tasks on the same core, i.e., q must start after the completion of p, if p is scheduled before q. It uses a big-M nullification [54] to *deactivate* the constraint if tasks are scheduled in the opposite order. M must always be greater than the left-hand side of the equality; we, therefore, use the sequential makespan of the application $M = \sum_{p \in \tau} max(C_p)$, as many other papers, e.g., [116].

$$s_{p,q} = o_{p,q} + o_{q,p}, \forall (p,q) \in (\tau \times \tau), p < q \tag{14}$$

$$\rho_p + C_p \leqslant \rho_q + (1 - o_{p,q}) \times M, \forall (p,q) \in (\tau \times \tau), p \neq q \tag{15}$$

Data dependencies in taskgraphs. Equation (16) ensures that if one task p depends on the data of another task q, the start time of p (ρ_p) is greater than the end time of q ($\rho_q + C_q$).

$$\rho_p \geqslant \rho_q + C_q, \forall p \in \tau, \forall q \in \textit{predecessors(p)} \tag{16}$$

Task version selection. Equation (17) enforces that exactly one version of each task is selected ($a_{p,i} = 1$). Each version is mapped to one and only one core ($x_{p,i,m} = 1$), Equation (18). And Equation (19) links version and accepted architecture ($x_{p,i,m}$). Then, Equation (20) sets the selected mapping at the task level ($w_{p,m}$).

$$\sum_{v \in v_p} a_{p,v} = 1, \forall p \in \tau \tag{17}$$

$$a_{p,v} = \sum_{u \in Cores} x_{p,v,u}, \forall p \in \tau, \forall v \in v_p \tag{18}$$

$$x_{p,v,u} = 0, \forall p \in \tau, \forall v \in v_p, \forall u \in \text{forbidden}(v) \tag{19}$$

$$w_{p,u} = \sum_{\forall v \in v_p} x_{p,v,u}, \forall p \in \tau, \forall u \in Cores \tag{20}$$

Energy & Timing & Frequency. Equations (21) to (23) set the energy, time, and frequency for each task (E_p, C_p, F_p) to the energy, time and frequency ($E_{p,v,u}$,$C_{p,v,u}$,$F_{p,v,u}$) of selected version $x_{p,v,u} = 1$.

$$E_p = \sum_{v \in v_p} (x_{p,v,u} \times E_{p,v,u}), \forall p \in \tau, \forall u \in Cores \tag{21}$$

$$C_p = \sum_{v \in v_p} (x_{p,v,u} \times C_{p,v,u}), \forall p \in \tau, \forall u \in Cores \tag{22}$$

$$F_p = \sum_{v \in v_p} (x_{p,v,u} \times F_{p,v,u}), \forall p \in \tau, \forall u \in Cores \tag{23}$$

Consistent CPU voltage island. When multiple tasks are mapped on the same CPU voltage island simultaneously, their frequencies must match as required by a voltage island. Equation (24) sets on which island i the task p is mapped ($is_{p,i} = 1$), while Equation (25) checks if two tasks p, q are on the same island ($sis_{p,q} = 1$). Then, Equation (26) checks if two tasks p, q overlap in time ($to_{p,q} = 1$) (obviously on different cores as enforced by previous constraints Equation (14)). And Equation (27) forces the frequency of two tasks p, q to be equal, $F_p = F_q$, if they are on the same voltage island at the same time.

$$is_{p,i} = \sum_{u \in i} (w_{p,u}), \forall p \in \tau, \forall i \in I \tag{24}$$

$$sis_{p,q} = \sum_{i \in I} (is_{p,i} \wedge is_{q,i}), \forall (p,q) \in (\tau \times \tau), p < q \tag{25}$$

$$to_{p,q} = (\rho_q + C_q) \geqslant \rho_p \wedge (\rho_p + C_p) \geqslant \rho_q, \forall (p,q) \in (\tau \times \tau), p < q \tag{26}$$

$$(sis_{p,q} \wedge to_{p,q}) \times (F_p - F_q) = 0, \forall (p,q) \in (\tau \times \tau), p < q \tag{27}$$

Time spent in each CPU voltage island frequency. To compute the time spent by each CPU voltage island at each frequency, we must look at each time quantum if there is a task active. Equation (28) scans all time steps between 0 and M, which is the longest possible (sequential) schedule, then set $ta_{t,p} = 1$ if the task p is active at that time. Note that it would be better to use the makespan of the schedule rather than M, but the makespan results from the scheduler's decisions and is therefore unknown when modelling the problem. Equation (29) then sets the binary variable $fa_{i,f,t} = 1$ if at least one task is active at time t with frequency f on island i. Finally, Equation (30) accumulates the time at which the island i runs at frequency f.

$$ta_{t,p} = (t \geqslant \rho_p) \wedge (t \leqslant (\rho_p + C_p)), \forall t \in [0; M], \forall p \in \tau \tag{28}$$

$$fa_{i,f,t} = ta_{p,t} \wedge (F_p == f) \wedge is_{p,i}, \forall t \in [0; M], \forall i \in I, \forall f \in F_i, \forall p \in \tau \tag{29}$$

$$time_{i,f} = \sum_{t \in [0;M]} fa_{i,f,t}, \forall i \in I, \forall f \in F_i \tag{30}$$

Prevent overlap of tasks on accelerators. On top of preventing tasks from overlapping on CPU cores (Equation (15)), we also need to prevent tasks from overlapping on accelerators. Equation (31) determines if task p requires accelerator x ($acc_{p,x} = 1$). Next Equation (32) determines if two tasks, p and q, are assigned to the same accelerator ($sa_{p,q} = 1$). If the two tasks are assigned to the same accelerator, Equation (33) determines the order, which

is either p and then q ($oa_{p,q} = 1$) or vice versa. Lastly, Equation (34) enforces the order using the same big-M nullification [54] as in Equation (15).

$$acc_{p,x} = \sum_{i \in v_p} (acc_{i,p,x} \times a_{p,i}), \forall p \in \tau, \forall x \in Acc \tag{31}$$

$$sa_{p,q} = \sum_{x \in Acc} acc_{p,x} \wedge acc_{p,x}, \forall (p,q) \in (\tau \times \tau), p < q, \forall x \in Acc \tag{32}$$

$$sa_{p,q} = oa_{p,q} + oa_{p,q}, \forall (p,q) \in (\tau \times \tau), p < q \tag{33}$$

$$\rho_p + C_p \leqslant \rho_q + (1 - oa_{p,q}) \times M, \forall (p,q) \in (\tau \times \tau), p \neq q \tag{34}$$

Compute time required at each accelerator frequency. A tasks accelerator frequency (afa_p) is based on the chosen version ($a_{p,i}$) and the accelerator frequency of the version $vfa_{i,p}$ (Equation (35)). The time required ($acctime_{f,x}$) in each frequency for all accelerators is computed in Equation (36). It sums up the run time for the different tasks τ for all frequencies F_x if the task is run on a given accelerator x and if the accelerator frequency matches.

$$afa_p = \sum_{i \in v_p} (a_{p,i} \times vfa_{i,p}), \forall p \in \tau \tag{35}$$

$$acctime_{f,x} = \sum_{p \in \tau} ((acc_{p,x} \wedge (afa_p == f)) \times C_p), \forall x \in Acc, \forall f \in F_x \tag{36}$$

4.7.2 *ILP vs eFLS*

To estimate the over-approximation of our eFLS heuristic (Section 4.3) over the optimal solution, we generate 500 taskgraphs with TGFF [41] and schedule them with both techniques. On average, the generated taskgraphs have 9.78 tasks with a standard deviation of 4.53. We use a different set of taskgraphs than in Sections 4.5 and 4.6 as the larger taskgraphs used in those sections are not schedulable by the ILP in a reasonable time frame. We then compare the predicted energy consumption of the generated schedules, i.e., we calculate the expected energy consumption of the taskgraph with respect to the two schedules and compare the energy consumption.

For each technique, the solving time vs the number of tasks can be found in Figure 4.20. The figure clearly shows that the ILP solving time increases exponentially with the number of tasks. Hence, demonstrating that the ILP does not scale well. For many DAGs, the ILP found non-optimal solutions as the solver ran out of time; these are solutions with gaps more significant than 0.1%. The ILP approach found optimal solutions, with a gap smaller than 0.1%, for 45% of the taskgraphs within 24 hours on a 16-core Intel Xeon Gold 6130 using Cplex. For the remaining 55%, the solver found solutions

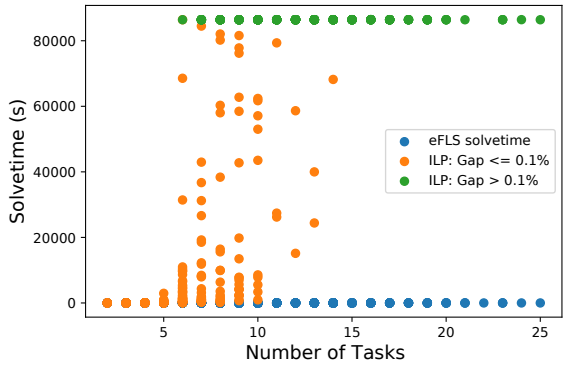


Figure 4.20: Solving time (s) of the ILP solver vs eFLS heuristic

with a gap larger than 0.1% in 38.2% of the cases and no solution in the last 16.8% of the taskgraphs. Next, we fit an exponential function ($y = ae^{t*x} + b$) to the number of tasks and solving time relationship. Using this exponential fit, we can extrapolate the solving time required by the ILP. For a taskgraph with 100 tasks, the ILP would need over 12800 days to be solved (approx. 35 years).

The average degradation of eFLS solutions compared to the optimal solution is 1.6%. The average energy degradation of the eFLS solutions, with respect to all ILPs with a solution, is 1.64%, which we deem an acceptable trade-off for shorter scheduling times and better scalability. The degradation distribution is shown in Figure 4.21 consisting of all ILP solutions, including the ones without an optimal solution. At best, both methods result in the same schedule (in 24.6% of the cases). At worst, the eFLS method results in a schedule that consumes 18.0% more energy.

4.7.3 *Energy consumption: predicted vs measured*

In this section, we compare the predicted energy consumption from the scheduler to the actual energy consumption when executed on the Odroid-XU4 board. There is no significant difference between the predicted makespan and the actual makespan as we employ a time-triggered approach.

We choose five arbitrary taskgraphs (Table 4.3) from Section 4.7.2 and execute the applications 50 times on the Odroid-XU4, for both the ILP and the eFLS generated schedules. An example taskgraph is shown in Figure 4.22. It consists of 9 tasks, where 5 of the tasks have GPU versions.

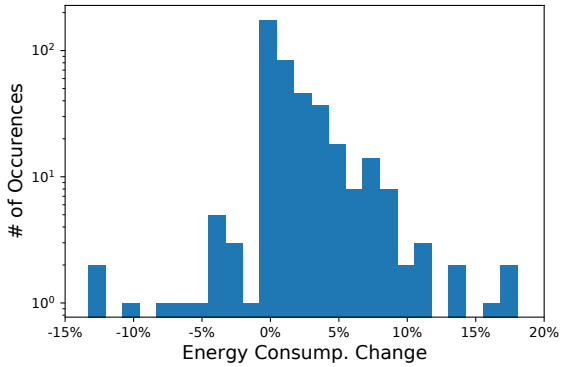


Figure 4.21: Distribution of energy degradation of the eFLS scheduler vs the ILP scheduler for all DAGs where the ILP solver arrived at a solution. (logarithmic scale)

For the taskgraph in Figure 4.22, both the ILP and the eFLS scheduler use the GPU for the *LU decomposition* task and execute the other tasks on the CPU. The most significant difference between the two schedules is that the ILP generated schedule mainly executes tasks at 1.3GHz and two tasks at 1.5GHz, whereas the eFLS schedule executes most tasks at 1.5GHz. Therefore, the eFLS generated schedule is a little faster and slightly worse at balancing runtime and energy consumption.

Table 4.3: The maximum and average error of the measured energy consumption vs the predicted energy consumption for each selected taskgraph for both the ILP and the eFLS solutions.

Taskgr.	#Tasks	ILP		eFLS		Degradation	
		max. er.	avg. er.	max. er.	avg. er.	Pred.	Meas.
97	6	-15.6%	-15.2%	-15.2%	-14.6%	7.9%	8.7%
151	7	-12.7%	-12.2%	-14.6%	-14.0%	3.0%	1.0%
159 [3]	9	-14.8%	-14.3%	-14.7%	-14.2%	2.7%	2.8%
225	10	-15.6%	-15.2%	-15.8%	-15.4%	3.0%	2.7%
320	6	-9.0%	-9.0%	-9.0%	-9.0%	0.0%	0.03%

Table 4.3 shows the error between the measured and the predicted energy consumption. All predictions overestimate energy consumption, likely due to the safety margin. The predicted energy consumption is at most off by 15.8%

3 Shown in Figure 4.22

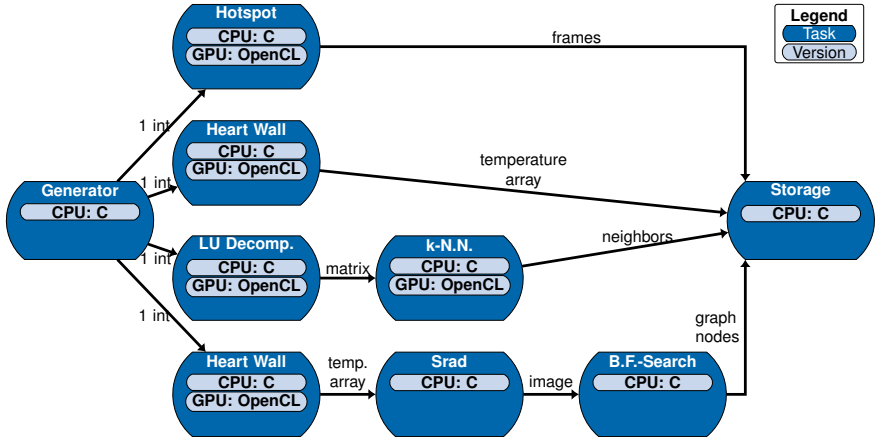


Figure 4.22: Illustration DAG composed of Rodinia benchmark tasks from Section 4.7.3 (Taskgraph number 159).

and at least by 9.0%. The measured energy consumption degradation of the eFLS solution is almost the same as the degradation of the predicted energy consumption. Thus, our predictions can be used to compare scheduling methods.

4.8 RELATED WORK

Most research until 2016 focused on energy-efficient scheduling for homogeneous multi-core platforms; for surveys, see [17, 52]. The two most explored techniques were Dynamic Voltage and Frequency Scaling (DVFS) and Dynamic Power Management (DPM). DVFS techniques balance clock frequency and voltage with required system performance, aiming for the best trade-off between energy consumption and execution time. DPM techniques switch CPU parts into a low-power state, thereby reducing energy consumption [17, 52].

In recent years attention has shifted from homogeneous to heterogeneous systems. Scheduling techniques now account for core mapping, and energy efficiency of tasks [123]. Most research focuses on two-type cores [79], and on scheduling independent tasks [123, 129]. In contrast, we address dependent tasks for multi-type CPUs with on-board GPU. We are also unaware of any energy-minimising approaches that work with multi-version tasks.

Most previous publications use energy models that rely on a standard power model that estimates the power consumption based on a static and a dynamic part [19, 55, 72, 129, 133]. However, this ignores that different tasks

may consume very different amounts of dynamic energy even if the tasks take the same amount of time (e.g., integer division vs double multiplication on big cores). Our energy model is based on per-task measurements. Thus, switching to a different architecture does not require a new power model but merely re-measuring tasks. Additionally, we do not consider core frequencies to be continuous, as this does not reflect the actual hardware. Instead, we consider the true set of available frequencies. Lastly, previous research does not consider DVFS for static energy consumption.

Unlike Guo et al. [55], we do not only focus on CPU power consumption but also on GPU power. Additionally, we work with time-triggered dependent tasks instead of independent parallel sporadic tasks. Zahaf et. al. [141] proposed scheduling approaches for soft real-time tasks running on heterogeneous multi-core platforms. They introduce integer nonlinear programming (INLP) and heuristics to determine the best parallelism for each independent task. In comparison, we minimise the energy consumption of dependent tasks executed concurrently. Thammawichai and Kerrigan [129] work with two-type heterogeneous multiprocessors, focusing on independent, preemptive tasks. The three approaches introduced use mixed-integer nonlinear programming (MINLP), nonlinear programming and a task ordering algorithm. Our work focuses on dependent non-preemptive dependent tasks.

Minakova et al. [94] focus on the heterogeneous scheduling of CNNs using a genetic algorithm. CNNs usually display a limited number of inter-task level parallelism (i.e., concurrency); furthermore, the tasks in a CNN that can be executed simultaneously are often the same (i.e., execution paths are symmetric). In contrast, our scheduler tackles a broader range of problems and considers DVFS, voltage-islands and multiple versions. Lastly, the genetic algorithm proposed in [94] also depends on a form of task ranking that was not further detailed. Kang et al. [78] also focus solely on scheduling CNNs across different compute units. In contrast to [94] Kang et al. take DVFS into account as it impacts execution times. However, compared to our work, they do not focus on reducing energy consumption nor actively change the DVFS settings at run-time. Lastly, their proposed scheduler also relies on first ranking the tasks and then mapping them in order to a given Compute Unit. The impact of the ranking is not considered. Lastly, Hu et al. [68] propose a dependent-task DVFS-aware greedy scheduler, but they do not support voltage islands, accelerators and multi-version tasks.

To summarise, in comparison to previous research, our work focuses on reducing energy consumption for dependent, time-triggered, multi-version

tasks on single heterogeneous devices. Furthermore, we extend the aspect of heterogeneity to take a heterogeneous CPU into account and the GPU. We introduce a more realistic energy model. Additionally, to the best of our knowledge, we are the first to explore the impact of different ranking algorithms on the quality of the schedules produced. Lastly, our experimentation is based on the Odroid-XU4, a more modern board than used in previous research.

4.9 CONCLUSION

The energy consumption and CO_2 footprint of the (I)IoT sector is increasing as the number of deployed devices keeps growing. However, not only does the number of deployed systems increase, but also the required compute power. (I)IoT systems are collecting more data that must be processed locally as devices are limited by bandwidth, privacy and security concerns. Only storing the data locally and physically collecting it (i.e., swapping the SD card) regularly is not a solution as the analysis results are often required fast (i.e., poachers being identified by a camera), and devices may be in remote locations. Hence, high-performance systems such as the Odroid-XU4 or the Nvidia Jetson boards are becoming widely used due to their high performance-to-power consumption ratio. On top of this, many application areas requiring such platforms are battery-powered. All of this means that we need to reduce the energy consumption of (I)IoT applications. To achieve this, we need to consider all features of modern hardware. To the best of our knowledge, we are the first to propose a scheduling approach that combines heterogeneous CPU and GPU, multi-version dependent tasks, a fine-grained energy model, an energy-focused ranking algorithm for FLS and in application frequency and voltage switching.

Our eFLS scheduling heuristic embraces heterogeneity and incorporates different CPU types, multiple-voltage islands and GPU-style accelerators. The scheduling heuristic can not only change the frequency at various intervals in the application but also employs an energy model to determine its direction that is more fine-grained than previous energy models used in research. Furthermore, our new ranking algorithm (Heterogeneous Energy Ranking) and the new ranking sub-set selection makes energy consumption a first-class citizen with respect to scheduling. Lastly, eFLS scales well with the size of taskgraphs.

We provide empirical evidence that our new ranking sub-set decreases the predicted energy consumption by up to 23% in comparison to the base

ranking methods (*DFS*, *BFS*, *BFS with laxity*, *BFS energy laxity*). Additionally, we demonstrate that our novel multi-version scheduling approach can take full advantage of a heterogeneous system, reducing energy consumption by, on average, 16% over a single version approach. We also show that there is no single best version for all tasks across all taskgraphs. This indicates that the multi-version approach produces more energy-efficient schedules than a single-version approach. Furthermore, we show that our eFLS scheduling approach outperforms a standard FLS scheduler and the state-of-the-art ARSH-FATI [133] scheduler by, on average, 27.8% with respect to energy consumption. Lastly, eFLS outperforms HEFT by 26.3% with respect to predicted energy consumption.

Our energy-aware Forward List Scheduling (eFLS) solutions experience a mean degradation of only 1.6%, with respect to energy consumption compared to optimal solutions derived by Integer Linear Programming (ILP). Lastly, we show that the energy predictions of our scheduling approach are similar to the measured energy consumption with a maximum error of 15.8%.

In summary, we introduce a new scheduling method that, in combination with a new ranking method, outperforms previous scheduling methods and can significantly improve the energy consumption of IoT applications. Additionally, we are the first to investigate a large variety of potential ranking algorithms.

5

MULTI-PHASE SCHEDULING

This chapter explores our multi-phase scheduling approach, which extends our previously introduced multi-version approach. The new task model splits a version of a task into multiple distinct execution blocks or phases. We show that splitting a version of a task into multiple phases offers substantial advantages with respect to hardware utilisation. This is followed by a new offline scheduling heuristic that maps and schedules tasks represented by versions and phases. Then, we demonstrate that our method significantly improves the schedulability (i.e. number of DAGs with a final makespan less than the deadline) by up to 11% and 24% in comparison to two multi-version phase-unaware schedulers (HEFT [131], and eFLS [148] respectively).

This chapter is based on:

- **J. Roeder**, B. Rouxel, and C. Grelck *"Scheduling DAGs of Multi-version Multi-phase Tasks on Heterogeneous Real-time Systems"* [151], in *14th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC 2021), Singapore. IEEE.*

The survey by Akesson et al. [3] shows that industry increasingly uses heterogeneous high-performance embedded systems. Companies not only use platforms with heterogeneous CPUs (e.g., ARM big.LITTLE architecture) but also employ systems with accelerator-style components such as GPU or FPGA (e.g., Nvidia Jetson, Odroid-XU4, Odroid-N2+). To increase the overall performance, taking full advantage of the available heterogeneity is crucial. Different binary incompatible compute units can be targeted by multiple alternatives/versions/implementations of a task [7, 67, 148]. Selecting the best task version depends on the schedule history and the available compute units. Therefore, we propose a multi-version-aware scheduling approach that benefits from more scheduling opportunities and, thus, creates more performant schedules.

Unlike CPUs, accelerators like GPUs require an external action to branch the application control flow to them, i.e., a CPU core is needed to prepare data and launch the GPU kernel. Hence, a task targeting a GPU has three natural phases: 1. a CPU phase that hands control to the GPU, 2. a GPU phase that does some computations and, 3. a CPU phase that handles

post-processing and resuming of the application. To tackle this problem we propose a new task model that splits tasks into a finer hierarchy of elements by adding phases, i.e., a task can be divided into phases. Phases enable more mapping and scheduling opportunities and increase hardware utilisation. Why can't we simply model multiple phases of one task as multiple independent tasks? The different phases of a task are more tightly coupled than independent tasks with respect to cache lines/memory etc.

Let us compare our proposed multi-phase approach with an approach that instead reserves both required hardware units for the entirety of the execution time of the task. Consider two independent tasks executed on an architecture with one CPU core and one GPU. *Task 1* starts with a CPU phase, followed by a GPU phase and finalises with a CPU phase. The Worst-Case Execution Time (WCET) of *Task 1* is 6 time units, including all phases in sequence. *Task 2* executes on the CPU only and has a WCET of 4 time units. A synchronous scheduling approach does not consider the different phases. Instead, it schedules the entire *Task 1* in one contiguous sequential block. A synchronous scheduling approach results in Figure 5.1 with a total makespan of 10 time units. When scheduling the task in one block,

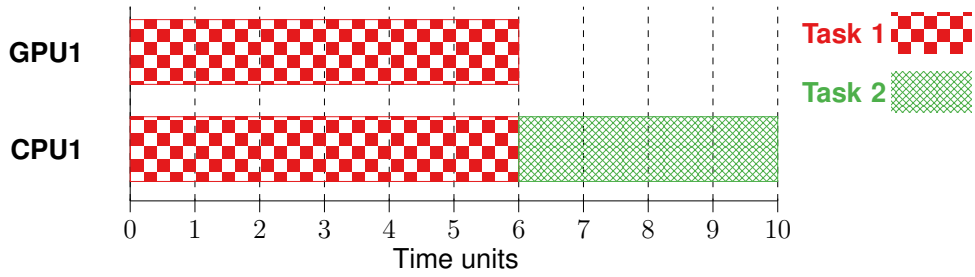


Figure 5.1: Motivational example without phases.

the CPU is stalled while executing the GPU phase. By splitting a task into different phases, we enable the asynchronous use of computing resources and schedule phases independently. Our approach is, thus, aware of this phase multiplexing and can rely on it to improve the schedulability rate for Directed Acyclic Graphs (DAG) with high utilisation, as shown in Figure 5.2 with a makespan of 6.

On top of our new multi-phase task model, we introduce an offline scheduling heuristic to map and schedule a set of tasks onto heterogeneous CU. Each task has at least one version, and each version has at least one phase. The static scheduler chooses which implementation variant of a task is used. The proposed strategy is formulated as a heuristic based on Forward List

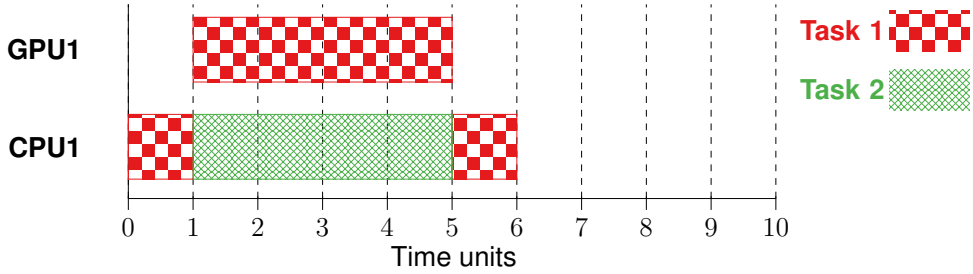


Figure 5.2: Motivational example with phases.

Scheduling (FLS). According to Davis and Burns [38], our heuristic can be classified as static, partitioned and time-triggered. Additionally, we support co-operative scheduling within the application; i.e. fixed preemption points in each task that can be used to execute another phase or task within the application.

This chapter builds upon our previous scheduling approach introduced in Chapter 4, where we tackled energy-aware scheduling for heterogeneous systems, mainly focusing on Dynamic Voltage and Frequency Scaling (DVFS) and energy modelling. Additionally, Chapter 4 followed a synchronous approach where any task requiring the GPU reserved both a CPU core and the GPU for the entire task length, leading to a CPU core being blocked while using the GPU and vice versa.

5.1 SYSTEM MODEL

We consider an application represented by a DAG, hereafter called task graph. A graph is a tuple $G = (D, \tau, E)$, where D is the deadline, τ represents the set of nodes/vertices, hereafter called tasks, and the set of edges E represents data dependencies between tasks. Hence, a producer task needs to be completed before the corresponding consumer task may start executing. Each task consists of a (non-empty) set of versions V.

The different versions of a task are functionally equivalent (i.e. they implement the same input/output relation) but differ in their structure and compute unit usage; hence, tasks differ in their non-functional properties (energy, time, etc.). Different versions can be the result of the following: 1. targeting different functional units; 2. different algorithms or implementation variants [148]; 3. using varying compilation flags to, for example, optimise code for energy consumption, binary size, speed, or architecture features [105].

Furthermore, each version is split into multiple phases. A phase is a sequential block of code executed on one compute unit. Each phase is characterised by a tuple $p = (C, U)$, where C is the Worst-Case Execution Time (WCET) and U is the set of compute units on which it can be executed and on which the WCET holds.

Figure 5.3 shows a task graph and illustrates different possibilities that our task model enables. The task graph consists of 9 tasks. Tasks 1, 4, 6 and 7 have one version. Each version consists of one phase that can be executed on any CPU type, indicated by *C*. Tasks 0, 2 and 8 have two available versions, where the first version is again a CPU version (*C*). The second version is a multi-phase version, indicated by *C-G-C*. The *C-G-C* version indicates that it is a three phase version, where the first and third phase can be executed on a CPU core (*C*) and the second phase (*G*) targets the GPU. Task 3 has two versions. The first version is a single phase version that targets *big* CPU cores only. The second version is a two phase version (*C-A*), where the first phase targets a CPU core (*C*) and the second phase targets an accelerator *A*. Task 5 also has two versions. The first single phase version targets the *LITTLE* CPU cores. The second version is a three phase version (*C-A-C*) that targets the CPU cores (*C*) and an accelerator (*A*).

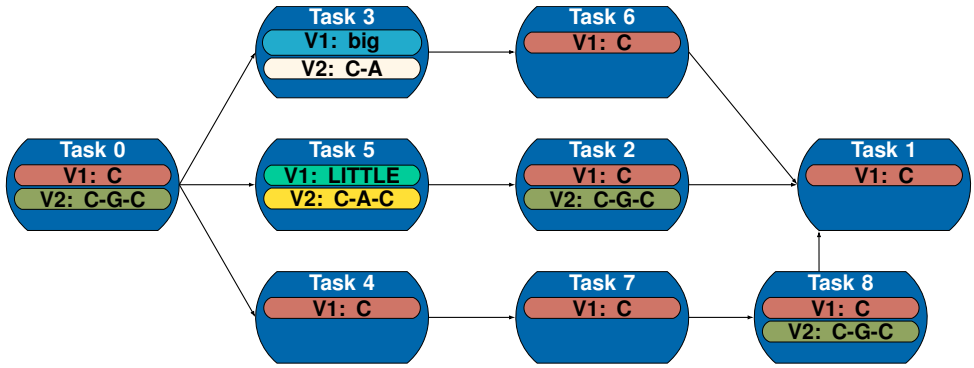


Figure 5.3: Example of a task graph where each task has one or more versions. A version contains one or more phases. *C* indicates that the phase can be executed on any CPU core, whereas *big* and *LITTLE* indicate a specific CPU architecture. *G* indicates that the phase has to be executed on the GPU. *A* stands for accelerator (e.g. FPGA).

We target heterogeneous platforms that embed any type of compute unit, including different types of CPU (e.g., big.LITTLE architectures), GPU, DSP, FPGA, and other accelerators. Our approach is fully platform-independent

and can be applied to a wide range of heterogeneous (embedded) system architectures.

5.2 INTERFERENCE

5.2.1 *Cache related delays*

Our approach allows for migration. For example, two different CPU phases of the same version can be executed on different CPU cores. We do not allow migration during execution, i.e., when a phase starts on a compute unit, it cannot migrate. Allowing *migration* of phases from the same task induces a non-negligible cost due to cache reloads. If two CPU phases of the same task version share cache lines spawning them on different cores affects the execution time of the second phase. Figure 5.4 shows an example of migration where the red part of the last phase of task *T0-V0-P3* needs to reload cache lines (instruction or data) that would have been present if executed on the same CPU core as the first phase.

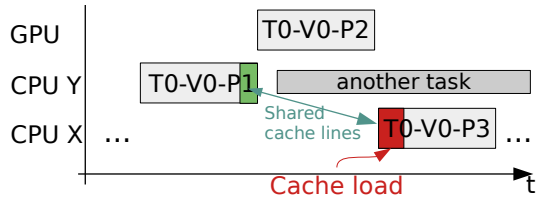


Figure 5.4: Example of migration. *T0* indicates that it is Task 0. *V0* indicates that it is version 0 of Task 0. And *P1* through *P3* indicates the phase.

Similarly, allowing asynchronous execution between phases has the same impact on the cache as migration, illustrated in Figure 5.5. Inserting a phase from another task between two phases of the same task (*entanglement*) can evict shared cache lines (both instruction or data). There are no shared cache lines between different compute unit types (e.g., CPU and GPU); thus, we assume an empty cache.

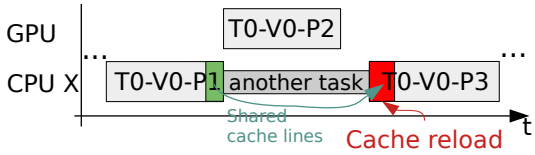


Figure 5.5: Example of entanglement. *T0* indicates that it is Task 0. *V0* indicates that it is version 0 of Task 0. And *P1* through *P3* indicates the phase.

This interference cost is well-known for preemption and is called *Cache Related Preemption Delay* (CRPD), e.g., [8]. Allowing phases from other tasks to execute in-between two phases of a task is similar to a preemption mechanism. Our scheduling strategy accounts for the CRPD by adding an extra cost to the WCET of a second phase that suffers from it. Computing the CRPD itself can be performed using many techniques, based on both static analysis [8] and dynamic analysis [98]. Our scheduling method is not linked to any particular CRPD analysis. We only require that an extra cost is added to the WCET in case of migration or entanglement.

5.2.2 *Shared resources interference*

Scheduling tasks for multi-core platforms requires accounting for hardware interference due to shared resources, e.g., bus, memory, etc. One possibility is to isolate the parts of the code that generate interference as phases and ensure that only one of these phases across all tasks can execute simultaneously. This approach would be similar to AER [90], or PREM [107]. However, these methods are unsuitable for our targeted applications and platforms (Odroid-XU4, Jetson TX2 etc.). The applications we consider (e.g., object detection) often have no predetermined load-compute-write structure, as the data might not fit into the cache. Therefore, we follow a more portable path and add the interference cost into the WCET to form the Worst Case Response Time (WCRT) of each phase and then schedule phases according to the WCRT [113].

5.2.3 *Data in memory interference*

All data transfer between two compute units (e.g., CPU to GPU) is accounted for in the CPU phases. Thus, we can target both shared memory and dedicated accelerator platforms. We consider memory management to be beyond the scope of this chapter. In other words, we assume sufficient memory to be available.

5.3 HETEROGENEOUS FLS

Our proposed heuristic is based on Forward List Scheduling (FLS). FLS first orders the tasks and then adds them one by one to the schedule without backtracking. We use three sorting algorithms: Depth First Search (DFS), Breadth First Search (BFS), and HEFT ranking [131]. For DFS and BFS, we

use the task WCET as a tie-breaking rule (larger WCET to be scheduled first). Furthermore, we introduce one additional tie-breaking rule for BFS based on *laxity*. Since it is shown in [116] that no sorting algorithm consistently outperforms the others, we generate four schedules, each resulting from one sorting strategy / tie-breaking rule combination, and select the one resulting in the lowest makespan as our heuristic solution. From here on-wards, we will refer to our heterogeneous, multi-version task scheduling heuristic as *hFLS*.

Our approach: We schedule all phases of a (task) version one after another. This way, we allow for migration between CPU cores or the CPU and an accelerator or even between multiple accelerators. When scheduling a phase, the heuristic checks if there is either migration or entanglement between the current phase and the previous phases of the same task and increases the WCET with the CRPD of the phase. Scheduling decisions are not changed retrospectively.

5.3.1 *Scheduling Algorithm*

Our proposed heuristic is sketched out in Algorithm 5.3.1. It takes as an input the DAG of the application represented by tasks τ, a set of compute units $\mathtt{CU}$ and a deadline $\mathtt{D}$. It sorts the tasks and creates a list (Line 2), then it loops over all the tasks in the list (Lines 5 – 45). We try all versions of a task (Lines 8 – 41), where each version starts with a clean schedule (Line 9). All phases of a version are scheduled (Lines 10 – 37) and all compute units $\mathtt{CU}$ are tried (Lines 14 – 34).

The scheduler ensures that a phase $\mathtt{ph}$ can execute on a given $\mathtt{CU}$ (Lines 15 – 16). Then we compute the Release Time (RT) of $\mathtt{ph}$ (Line 19). We evaluate if the RT and mapping suffer from either migration or entanglement interference and update the WCET of $\mathtt{ph}$ (Line 20). If the specific RT or mapping results in interference, the RT is recalculated (Line 22) with the new WCET. Section 5.3.2 explores the RT calculations. Section 5.3.3 explores the WCET update. After the RT and WCET updating steps, $\mathtt{ph}$ is, added to the schedule (Line 24).

Adding phase $\mathtt{ph}$ to the schedule could mean that a previously scheduled phase of a different task now suffers from interference. Thus, Line 26 finds and propagates such interference through the schedule. The interference propagation algorithm is explored in Section 5.3.4.

The best core mapping for a phase depends not only on the $\mathtt{makespan}$ but also on the release time. An earlier release time of the first phase could

lead to a better $makespan$ for the latter phases (Lines 29). The best core for a phase is saved in $bestCoreSchedule$ (Line 30). Once the best core for a phase has been found, the $versionSchedule$ is updated (Line 35), which is the base for scheduling the next phase.

After all phases have been scheduled, $versionSchedule$ contains the best schedule for a task version. The best task version is saved in the $tmpSchedule$ (Line 39). Once all versions of a task have been tried, the original $schedule$ is updated (Line 42), and the $Qdone$ list is extended (Line 44).

After all tasks in the $Qready$ list have been scheduled we check if the resulting schedule has a $makespan$ lower than the deadline D (Line 46).

Algorithm 5.3.1 Scheduling Algorithm

Input: A DAG composed of multi-version tasks (τ), a set of compute units (CU) and a deadline (D).

Output: A schedule.

```
1:  function ListSchedule(τ_x ∈ τ|τ_x = v, CU, D^C)
2:      Qready ← TopologicalSortTasks(τ)
3:      Qdone ← []
4:      schedule ← new Schedule()
5:      while t ← Qready.popFront() do
6:          tmpSched ← schedule
7:          tmpSched.makespan ← ∞
8:          foreach v ∈ t.versions do
9:              versionSchedule ← schedule
10:             foreach ph ∈ v.phases do
11:                 bestCoreSchedule ← versionSchedule
12:                 bestCoreSchedule.makespan ← ∞
13:                 oldReleaseTime ← ∞
14:                 foreach u ∈ CU do
15:                     if ph does not run on u then
16:                         continue
17:                     end if
18:                     copy ← versionSchedule
19:                     copy.phaseRT(Qdone, t, v, ph, u)
20:                     copy.updateWCET(Qdone, t, v, ph, u)
21:                     if ph.appliedCRPD() then
```

```
22:                                copy.phaseRT(Qdone, t, v, ph, u)
23:                            end if
24:                            copy.addPhase(ph)
25:                            ph.setTargetCU(u)
26:                            copy.propInterferences(Qdone, t, v, ph, u)
27:                            copy.updateMakespan()
28:                            newReleaseTime ← copy.releaseTime(ph)
29:                            if copy.makespan < bestCoreSchedule.makespan
    ∨ newReleaseTime < oldReleaseTime then
30:                                bestCoreSchedule ← copy
31:                                bestCoreSchedule.updateMakespan()
32:                                oldReleaseTime ← newReleaseTime
33:                            end if
34:                        end for
35:                        versionSchedule ← bestCoreSchedule
36:                        versionSchedule.updateMakespan()
37:                    end for
38:                    if versionSchedule.makespan < tmpSched.makespan then
39:                        tmpSched ← versionSchedule
40:                    end if
41:                end for
42:                schedule ← tmpSchedule
43:                schedule.updateMakespan()
44:                Qdone.pushBack(t)
45:            end while
46:            if schedule.makespan > D then
47:                return unschedulable
48:            end if
49:            return schedule
50:        end function
```

Proof of termination. We only loop over a finite set of tasks τ, and each task has a finite set of versions. Each version has a finite number of phases. Additionally, the number of CU is finite. Lastly, all function calls are guaranteed to terminate as they iterate over finite data structures.

Complexity. The complexity of Algorithm 5.3.1 (excluding all algorithms called in it) is $\mathcal{O}(n \times v \times ph \times u)$, where n is the number of tasks, v is the number of versions, ph is the number of phases and u is the number of CU.

5.3.2 *Phase Release Time*

Algorithm 5.3.2 Phase release time

Input: Scheduled tasks (Qdone), task to schedule (curTask), phase (ph), processor (curCU).
Output: Computes a phases' release time.

```
1:  function PHASERT(Qdone, curTask, ph, curCU)
2:      ph.ρ ← 0
3:      foreach x in curTask.predecessors do
4:          ph.ρ ← max(x.ρ + x.C, ph.ρ)
5:      end for
6:      y ← ph.previousPhase
7:      ph.ρ ← max(ph.ρ, y.ρ + y.C)
8:      change ← true
9:      while change do
10:         change ← false
11:         foreach t ∈ Qdone do
12:             if t is mapped on curCU then
13:                 if t overlaps in time with ph then
14:                     ph.ρ ← t.ρ + t.C
15:                     change ← true
16:                 end if
17:             end if
18:         end for
19:     end while
20: end function
```

Algorithm 5.3.2 sketches out the release time ρ computation of a phase. The input is the list of scheduled tasks Qdone, current task curTask, phase ph and target processor curCU. The start time / release time of a task ($ph.\rho$) must be after all predecessors of the task have completed (Lines 3 – 4). Additionally, ph must be be released after the previous phase of the same task (Lines 6 – 7). We must ensure that there is no overlap with previously scheduled tasks on the same compute unit curCU (Lines 9 – 19). If there is an overlap, ρ of ph is set to the previous tasks completion time (Line 14).

Proof of termination. Algorithm 5.3.2 is guaranteed to terminate because we only postpone ph. In the worst-case scenario, ph is moved to the end of the schedule.

Complexity. The worst case complexity of Algorithm 5.3.2 is $\mathcal{O}(n^2)$. Let us consider an application with n tasks, and all tasks but one have been scheduled onto a single core system. The last task has a single version with a single phase ρ and has no predecessors. In the worst-case scenario the phase, ph, has to be moved past all previously scheduled tasks. If ph is only postponed once per iteration of the while loop, we need a total of n^2 iterations.

5.3.3 *Updating phase WCET*

Algorithm 5.3.3 Updating the WCET of a phase

Input: Scheduled tasks (Qdone), phase (ph), processor (curCU).
Output: Add the CRPD cost to a phase in case of migration or entanglement.

```
1:  function updateWCET(Qdone, ph, curCU)
2:      foreach pp in ph.previousPhase do
3:          if curCU != pp.targetCU() && curCU.type() == pp.targetCU().type()
    then
4:              ph.applyCRPD()
5:              break
6:          else if curCU == pp.targetCU() then
7:              foreach t ∈ Qdone do
8:                  if pp == t || curCU != t.targetCU() then
9:                      continue
10:                 end if
11:                 if t between pp and ph then
12:                     ph.applyCRPD()
13:                     break 2
14:                 end if
15:             end for
16:         end if
17:     end for
18: end function
```

Algorithm 5.3.3 sketches out how the WCET of ph is updated in case of interference with respect to one of the previous phases. First, we iterate over all previous phases (Line 2 – 17) and check if ph migrates with respect to

a previous phase (Line 3). A *migration* occurs if ph executes on a different compute unit of the same type as the mapping of a previous phase. If migration occurs, we add the CRPD to the phase WCET (Line 4) and break out of the loop (Line 5) because the CRPD should only be added once. If ph does not migrate, we still need to verify that there is no *entanglement* with respect to the previous phases (Lines 6 – 16). To do so, we check if any of the other scheduled tasks (Line 7) is between ph and a previous phase (Line 11). If *entanglement* occurs, we add the CRPD (Line 12) to the WCET and break out of both loops (Line 13).

Proof of termination. Updating the WCET of a phase is guaranteed to terminate because the algorithm iterates over two finite data structures. The first is the list of the previous phases pp and the second is the list of already scheduled tasks Qdone.

Complexity. The complexity of the algorithm is $\mathcal{O}(\mathrm{ph}_{\mathrm{pp}} \times n)$, where $\mathrm{ph}_{\mathrm{pp}}$ is the number of previous phases and n is the number of tasks in Qdone.

5.3.4 *Find and propagate interference across the schedule*

Algorithm 5.3.4 sketches out how the scheduling of the current phase propagates interference through the already scheduled tasks and phases. First, we iterate over all previously scheduled tasks (Lines 3 – 22). We only check for interference if the current phase ph is not part of the previously scheduled task t (Line 4). Then we iterate over the phases in t twice (Lines 5 and 9). At this point, we only need to continue: 1. If the phases are both on curCU (Lines 6 and 10). 2. If the p1&p2 are different and p2 is scheduled after p1 (Line 13).

Then if ph is scheduled between p1 and p2 (Line 16), there is *entanglement* that impacts p2. At this point, we can break out of all three loops (Line 18) because only one previously scheduled phase can be impacted by ph.

If a previously scheduled phase is impacted (Line 24), the interference must be propagated through the rest of the already scheduled tasks/phases. This is done recursively, where all successors of the impacted task are added to the impacted list (Line 25). Additionally, all tasks/phases scheduled after the impacted phases and on the same compute unit are also added to the impacted list. This is done recursively until all successors or tasks on the same core have been added to the list. After that, the CRPD of the original impacted task/phase is used to update all impacted elements (Line 29).

Proof of termination. Algorithm 5.3.4 is guaranteed to terminate as we only increment the release time of the impacted tasks. We never decrement

Algorithm 5.3.4 Finding and propagating interference caused by the scheduling of the current phase.

Input: Scheduled tasks (Qdone), phase (ph), processor (curCU).
Output: Check if the scheduling of the current task causes any interference of tasks/phases that have already been scheduled.

```
1: function PROPINTERFERENCES(Qdone, ph, curCU)
2:     Find any phase with a longer WCET due to the scheduling of the current task.
3:     foreach t ∈ Qdone do
4:         if ph ∉ t then
5:             foreach p1 ∈ t do
6:                 if curCU ≠ p1.targetCU() then
7:                     continue
8:                 end if
9:                 foreach p2 ∈ t do
10:                    if p1.targetCU() ≠ p2.targetCU() then
11:                        continue
12:                    end if
13:                    if p1 == p2 || p2 after p1 then
14:                        continue
15:                    end if
16:                    if ph between p1 & p2 then
17:                        impactedPhase ← p2
18:                        break 3
19:                    end if
20:                end for
21:            end for
22:        end if
23:    end for
24:    if impactedPhase then
25:        impacted ← [impactedPhase]
26:        impacted.findAllImpacted()
27:    end if
28:    foreach i ∈ impacted do
29:        i.update(impactedPhase.crpd())
30:    end for
31: end function
```

the release time. Hence, in the worst-case scenario, all release times of all tasks released after the original impactedPhase must be incremented.

Complexity. The complexity of the algorithm described in Lines 3 – 22 is $\mathcal{O}(t_p^2 \times n)$, where t_p is the number of phases of task t and n is the number of tasks.

The complexity of the recursive function (impacted.findAllImpacted()) is $\mathcal{O}(n^2)$. In the worst-case scenario, the last scheduled phase ph impacts the first task in the schedule. Thus, it might happen that all tasks after the first task are impacted. In this case, all tasks have to be checked against each other.

5.4 EVALUATION

In this section, we first evaluate our hFLS heuristic against an Integer Linear Programming (ILP) formulation and then against two synchronous phase-unaware heuristics. Our experimental results are based on synthetic benchmarks.

5.4.1 *hFLS vs ILP*

Heuristic algorithms intrinsically generate approximate results but usually scale well. ILP formulations give exact results to the problem but do not scale well. We compare schedules generated by the heuristics against the ones generated by the ILP to estimate the heuristics over-approximation.

We summarise and introduce all variables and functions in Table 5.1. For conciseness, we use logical operators ($\vee$, $\wedge$). The two logical operators can be linearised as shown by Brown and Dell [25]. Our ILP formulation is as follows.

Task mapping. Eq. 37 assigns a single version a per task t.

$$\forall t \in \tau, \sum_{v \in version(t)} a_{t,v} = 1 \tag{37}$$

Phase mapping. Eq. 38 assigns one compute unit per selected ($a_{t,v} = 1$) phase p.

$$\forall t \in \tau, \forall v \in version(t), \forall p \in phase(v), \sum_{c \in \gamma} b_{p,c} = a_{t,v} \tag{38}$$

Detect identical mapping. Eq. 39 detects if two phases (p & q) are mapped to the same compute unit (c) .

Table 5.1: Summary of ILP variables.

Variable	Description
γ	Set of compute units
τ	Set of tasks
$version(t)$	Set of versions for task t
$phase(X)$	Set of phases for X where X can be a task or a version
$succ(X)$	Set of successors for X where X can be a task or a phase
$deny(p)$	Set of forbidden compute units for phase p
D_t	Absolute deadline for for task t
$C_{p,c}$	WCET of a phase p on a compute unit c
$a_{t,v}$	1 if version v is selected for task t
$b_{p,c}$	1 if phase p is mapped on compute unit c
$d_{p,q}$	1 if phase p and phase q are on the same compute unit
$e_{p,q}$	1 if phase p is scheduled before phase k
$f_{p,q}$	1 if phase p is scheduled before phase k and both phases are on the same core
ρ_p	Start time of phase p
$\omega_{p,c}$	Augmented WCET for phase p on compute unit c

$$\forall (u,t) \in \tau \times \tau, \forall p \in phase(u), \forall q \in phase(t), u \neq t, \\ d_{p,q} = \sum_{c \in \gamma} b_{p,c} \wedge b_{q,c} \tag{39}$$

Mapping restriction. Eq. 40 enforces compute unit mapping restrictions (e.g., p runs on a GPU and not on a CPU).

$$\forall t \in \tau, \forall p \in phase(t), \forall c \in deny(p), b_{p,c} = 0 \tag{40}$$

Order phases in task. Eq. 41 enforces the phase ordering within a version, i.e. q only starts after p is complete if q succeeds p.

$$\forall t \in \tau, \forall p \in phase(t), \forall q \in succ(p), \\ \forall c \notin deny(p), \rho_p + \omega_{p,c} = \rho_q \tag{41}$$

Deadline. Eq. 42 guarantees the timing constraints of the graph, i.e., all phases must complete before the deadline D.

$$\forall t \in \tau, q \in phase(t), \forall c \notin deny(q), \rho_q + \omega_{q,c} \leqslant D_t \tag{42}$$

Task dependency. Like Eq. 41, Eq. 43 enforces task dependencies, i.e., task q only starts after all its predecessor phases.

$$\begin{aligned} \forall u \in \tau, \forall t \in succ(u), \forall p \in phase(u), \forall q \in phase(t), \\ \forall c \notin deny(p), \rho_p + \omega_{p,c} \leqslant \rho_q \end{aligned} \tag{43}$$

Phase ordering. Eq. 44 sets the ordering of two successive phases. If $e_{p,q} = 1$, then phase p is scheduled before phase q. Eq. 45 detects the ordering of two successive phases on the same core ($d_{p,q} = 1$).

$$\forall (u,t) \in \tau \times \tau, \forall p \in phase(u), \forall q \in phase(t),$$

$$e_{p,q} + e_{q,p} = 1 \tag{44}$$

$$f_{p,q} = e_{p,q} \wedge d_{p,q} \tag{45}$$

Prevent overlap. Eq. 46 prevents multiple reservations of a computing unit simultaneously. It uses the common big-M ($\mathcal{M}$) notation, where $\mathcal{M}$ is the sum of all WCETs.

$$\begin{aligned} \forall (u,t) \in \tau \times \tau, \forall p \in phase(u), \forall q \in phase(t), \\ \forall c \notin deny(p), u \neq t, \\ \rho_p + \omega_{p,c} \leqslant \rho_q + (1 - f_{p,q})\mathcal{M} \end{aligned} \tag{46}$$

Entanglement cost. Eq. 47 adds the CRPD cost (Section 5.2.1) if two phases are entangled by augmenting the WCET ($\omega_{r,c}$) of phase r. The entanglement cost ($CRPD_{p,r}$) is only added if phase q is mapped between phase p & r and is on the same compute unit (d).

$$\begin{aligned} \forall (u,t) \in \tau \times \tau, \\ \forall p \in phase(u), \forall q \in phase(t), \forall r \in succ(p), \\ \forall c \notin deny(p) \cap deny(q) \cap deny(r), \\ (d_{p,q} == d_{q,r} == 1 \wedge \rho_p < \rho_q < \rho_r) \\ \implies \omega_{r,c} = C_{r,c} + CRPD_{p,r} \end{aligned} \tag{47}$$

Migration delay. Eq. 48 accounts for additional migration delay mentioned in Section 5.2.1. Equations 47 and 48 are mutually exclusive as either the two phases p, and r are mapped on the same core (Eq. 47) or on different

cores (Eq. 48). The migration delay is only added if the two phases target the same compute unit type, but different compute units, e.g., LITTLE core 1 & 2.

$$\begin{aligned} &\forall t \in \tau, \forall p \in phase(t), \forall q \in succ(p), \\ &\qquad \forall c \notin deny(p) \cap deny(q), \\ &\omega_{q,c} = C_{q,c} + CRPD_{p,q}(1 - d_{p,r}) \end{aligned} \tag{48}$$

Setup. The target platform for this experiment is a homogeneous quad-core CPU with an onboard GPU. We generate 1000 task graphs using Task Graphs For Free (*TGFF*) [41]. Additionally, TGFF provides the WCET for a simple CPU version and the deadline of the graph. The WCET provided by TGFF can be generated using the *type_attrib* option. We used the following parameters: *60, -50, 0.5, 1*. We generate a three-phase version which targets the GPU. The WCET of the GPU version is based on the CPU WCET and is either lower or equal to the CPU WCET. The task graphs contain, on average, 38 tasks. Links to the datasets and code used in this chapter can be found on page 137. The CRPD of a phase is an arbitrarily chosen 5% of the WCET of the impacted phase. The same CRPD is applied to both hFLS and the ILP. The CRPD is set arbitrarily as we do not have an appropriate target platform that could be used to estimate the CRPD. The conclusion does not change when increasing the CRPD level to 100%. Executing the DAG on the CPU sequentially yields the completion time. And dividing the completion time by the deadline yields the utilisation.

To solve the ILP, we use CPLEX [1] v12.10 executed on a 16core Intel system with 96GB memory. The heuristic was developed in C++ and was executed on the same machine as the ILP. For the ILP solver, we set up a timeout of 12h and a memory limit of 90GB per DAG.

Results. Due to the time and memory limits, the ILP solver was not able to reach a decision in 21.2% of the DAGs. For the DAGs that the solver came to a decision, 54.8% were marked as unschedulable (i.e. makespan exceeding deadline). All cases marked as unschedulable by the ILP were also decided as unschedulable by the heuristic, i.e., showing that the heuristic does not come to impossible conclusions. Compared to the ILP, the heuristic marked 2.4% more DAGs as unschedulable. Thus, showing that the ILP slightly outperforms hFLS with respect to schedulability. This is also reflected by Figure 5.6, which compares the schedulability rate per utilisation range of the ILP and hFLS. Figure 5.6 only compares the DAGs where the ILP reached a decision. Thus, the DAGs for which the ILP solver could not reach

1 https://www.ibm.com/nl-en/analytics/cplex-optimizer

a decision are excluded. Each utilisation range represents the aggregated schedulability rate for all DAGs within that range, e.g. range 1 to 2 contains 123 DAGs, out of which 108 were schedulable. We can clearly see that for higher utilisation ranges (3 to 4 and 4 to 5), the ILP achieves a higher schedulability rate than hFLS.

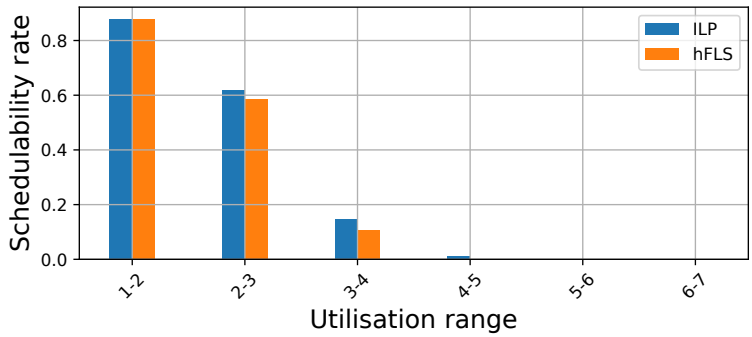


Figure 5.6: Performance of hFLS scheduler vs ILP scheduler. Each utilisation range represents the aggregated schedulability rate for all DAGs within that range, e.g. range 1 to 2 contains 123 DAGs, out of which 108 were schedulable.

Thus, why don't we use the ILP exclusively if it leads to a higher schedulability rate? This is perfectly answered by Figure 5.7, which shows the scheduling time (in seconds) with respect to the number of tasks in the DAG. We can see that the ILP does not scale well with an increasing number of tasks. The ILP solver cannot find a solution in a reasonable amount of time for most DAGs above 60 tasks. Thus, hFLS is capable of scheduling significantly larger taskgraphs.

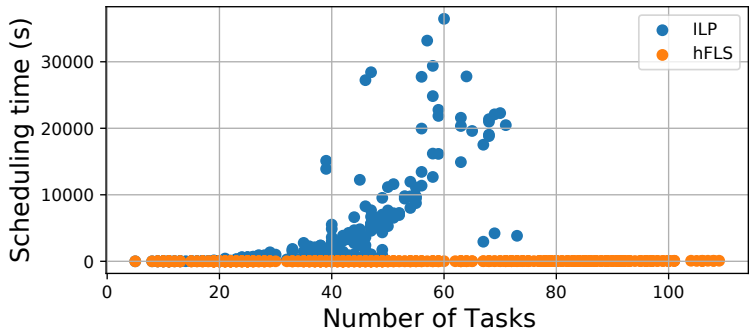


Figure 5.7: Scheduling time (s) taken with respect to the number of tasks.

5.4.2 *hFLS vs eFLS vs HEFT*

We compare our approach with our own multi-version energy-aware *eFLS* approach [148] and with HEFT [131]. For a fair comparison, we added multi-version capabilities to HEFT and modified the *eFLS* approach to focus on makespan instead of energy consumption. Both comparison heuristics use a synchronous scheduling approach, i.e., a GPU version with multiple phases blocks both the CPU and GPU for the complete execution time of the task.

Setup. We schedule 10,000 task graphs for the Odroid-XU4 [59]. The Odroid-XU4 contains an Arm big.LITTLE CPU [1] and a Mali GPU[49]. The CPU contains 4 energy-efficient, in-order LITTLE cores and 4 high-performance, out-of-order big cores. CPU and GPU share a common memory. This setup is more representative of modern high-performance embedded systems than the setup from the previous section (Section 5.4.1). We could not make use of this more realistic setup in Section 5.4.1 due to the poor scaling of the ILP.

Like in Section 4.4.3 and Section 5.4.1 TGFF [41] provides the structure of each DAG. Additionally, TGFF provides the WCET of each task on the LITTLE cores and the deadline. The task graphs contain, on average, 76 tasks. Next to the LITTLE core version, there are another 3 versions: one CPU-only version for the big cores, one 3-phase GPU version initiated on the LITTLE cores and one 3-phase GPU version initiated on the big cores. The WCET of the additional versions is based on the WCET of the LITTLE version, similar to the approach in Section 5.4.1. The *big* version always takes less time than the initial *LITTLE* version. The WCET of the *GPU* version initiated on the LITTLE cores is less or equal to the *LITTLE* CPU version. The WCET of the last version (i.e., initiated on the big cores) depends on the *GPU* version initiated on the *LITTLE* cores and only the phases that are executed on the *big* cores change. We assume that the *GPU* phase is the same for both the *LITTLE* and the *big* version. The utilisation of each task graph is calculated in the same manner as in the previous section. The only difference is that we use the LITTLE core WCET to compute the sequential schedule.

We calculate the CRPD of each phase based on the WCET of the impacted phase. The CRPD is equal to either the WCET (i.e., 100% of the WCET) or the platform's upper CRPD bound, e.g., fully reloading L2 cache with poor pre-fetching. Thus, for short phases the CRPD cost might be high, but for long-running tasks the CRPD costs might be negligible (e.g., Object Detection takes up to a second on a Jetson [130]).

The Odroid-XU4 has a 2MB big core L2 cache [59] and 64Byte cache lines [33]. Assuming a worst-case scenario of 414 cycles difference between L2 and main memory [126] (i.e., having to reload L2 cache from main memory) at a frequency of 2GHz results in an upper bound of 6.8ms (rounded to 7ms). The additional time required for datasets larger than the L2 cache would already be accounted for in the WCET.

Results. Figure 5.8 shows the schedulability rate for each utilisation range. The schedulability rate of the *hFLS* heuristic is the same as the schedulability rate of the *eFLS*, and *HEFT* approaches until a utilisation index of 4. At utilisation rates above 4, the *hFLS* approach achieves, on average, 12% higher schedulability rates than the *eFLS* approach. In the utilisation ranges 9-11, the *hFLS* approach results in a 24% higher schedulability rate. *hFLS* outperforms the multi-version HEFT by 2.5% on average and is never worse. The most noteworthy is the significantly better performance (up to 11% higher schedulability rate) in the higher utilisation ranges. In a production environment, this could make the difference between a cheaper and a more expensive SoC, as hFLS enables better hardware utilisation. Lastly, *hFLS* performs statistically significantly (McNemar's Test [93]) better than both approaches at a confidence interval of 99.9%. Note that all approaches can schedule DAGs with a utilisation above 8 (i.e., above the number of cores). Two factors cause this. First, the utilisation is calculated based on the LITTLE core WCET. However, tasks can have a lower WCET on the big cores and the GPU. Second, the GPU adds an additional "core".

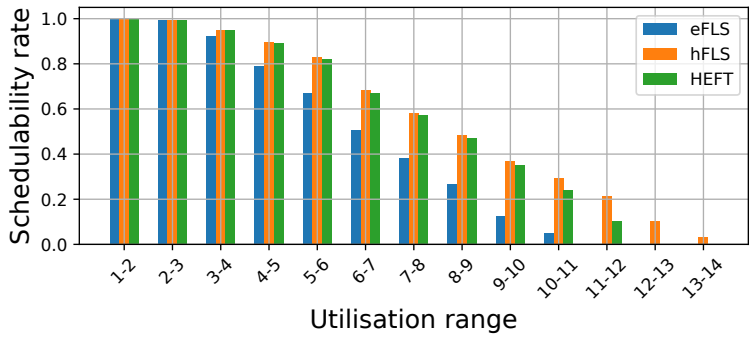


Figure 5.8: Performance of multi-phase schedules vs single-phase schedules. Each utilisation range represents the aggregated schedulability rate for all DAGs within that range, e.g., range 2 to 3 contains 1099 DAGs, of which 1039 were schedulable by HEFT and hFLS.

5.4.3 *Sorting for hFLS*

Many scheduling algorithms (e.g., HEFT, eFLS) require a total order of the tasks that need to be scheduled. A DAG provides a partial order by nature. Rouxel et al. [116] showed that no ranking strategy consistently outperforms (i.e., results in the best schedule) all other ranking strategies. In our current work on multi-phase scheduling, we observe that the HEFT ranking results in the best schedule in 73.6% of the cases. One of the two BFS sorting strategies results in the best schedule in 25.6% of the cases. In the other 0.8% of the cases, one of the DFS strategies was better. In the few instances where a DFS strategy resulted in a better schedule than either BFS or HEFT ranking, the makespan improved by only 2.6%. We observe that a DFS strategy leads to more entanglement of later phases and propagates delays throughout the schedule, i.e., more CRPD values are added to tasks, resulting in longer makespans. Thus, DFS ranking does not add much value in the case of hFLS. Therefore, DFS ranking is not necessary, and we can decrease the scheduling time of the heuristic.

5.5 RELATED WORK

Most previous research in the area of static scheduling for heterogeneous architectures focused on scheduling for heterogeneous CPUs [12, 22, 118, 131, 143]. Previous work on scheduling incorporating GPUs has either focused on workload balancing between the CPU and GPU [53, 118], introducing real-time capabilities to desktop GPUs [60] or running multiple tasks concurrently on the same GPU [104].

In [87], the authors introduce a run-time system for task and data mapping for embedded systems with a GPU. The aim is to minimise the completion time. However, a task only requires either the CPU or GPU. In contrast to our approach, they cannot handle multiple versions.

Multi-version scheduling is explored in [67], [148] and [7]. Houssam-Eddine et al. [67] introduce an online scheduling approach with an offline component for scheduling multi-version tasks. The target architecture for a task is selected offline, and then the task is scheduled online. Their approach does not support executing a task on both CPU and GPU, i.e., a GPU task does not have a CPU phase. Furthermore, in contrast to our offline approach, their approach requires loading binaries for all versions, substantially increasing the memory requirements.

Aldegheri et al. [7] also target CPU+GPU scheduling for multi-version tasks. Their approach extends HEFT by improving the ranking of *exclusive* tasks (tasks that only have one implementation). Additionally, tasks can only use either the CPU or the GPU. Our tasks are not limited to using a single compute engine, but can use multiple compute engines and switch between them.

Lastly, our multi-phase approach extends well-known concepts of multi-phase tasks (AER [90], PREM [107] or LET [63]) by allowing phases to execute on different compute units.

5.6 CONCLUSION

We extend existing task models along with a scheduling strategy to fully benefit from the capacities offered by heterogeneous hardware such as the Odroid-XU4 or the Nvidia Jetson boards. To the best of our knowledge, we are the first to propose a task model and a heuristic that interleaves CPU and GPU workloads of different tasks using a multi-version multi-phase approach.

We demonstrate that our approach can utilise the hardware better than two synchronous schedulers, improving schedulability by up to 11% and 24%, respectively, for high-utilisation DAGs. Additionally, we show that HEFT ranking or BFS sorting algorithms perform best in our approach, reducing the scheduling time by 40%. Lastly, the solutions given by our hFLS heuristic are close to the optimal with at most 2.4% degradation.

6 SCHEDULING BASED ON REINFORCEMENT LEARNING

This chapter explores offline scheduling of dependent tasks (i.e., DAGs) using a reinforcement learning approach. First, we explain the different components of our two RL schedulers and illustrate how they schedule a task. Then, we compare our RL schedulers to a FLS approach based on two different datasets. We demonstrate that our Graph Convolutional Network based scheduler produces schedules that are as good or better than the schedules produced by the FLS approach in over 85% of the cases for a dataset with small taskgraphs. The same scheduler performs very similar to the FLS scheduler (at most 5% degradation) in almost 76% of the cases for a more challenging dataset.

This chapter is based on:

- **J. Roeder**, B. Rouxel, and C. Grelck *"Q-learning for Statically Scheduling DAGs"* [150], in *2020 IEEE International Conference on Big Data (Big Data)*.
- **J. Roeder**, A.D. Pimentel, and C. Grelck *"GCN-based reinforcement learning approach for scheduling DAG applications"* [146], in *19th Artificial Intelligence Applications and Innovations (AIAI 2023)*.

In the previous chapters, we relied on Integer Linear Programming (ILP) or heuristics to schedule applications represented by Directed Acyclic Graphs. The former provides an optimal solution, but does not scale well with large DAGs. The latter scales better with the problem size but often relies heavily on the initial ranking of tasks (see Chapter 4). Finding near-optimal static schedules for larger DAGs within a reasonable solving time is still an open problem.

Artificial intelligence (AI) may provide a feasible solution. However, supervised learning is not possible because it requires a set of problems with solutions. As shown in Chapter 4, finding an optimal solution for a taskgraph of 100 tasks would take approximatley 35 years. Thus, we could never make a large enough training dataset for more involved problems. Furthermore, as shown in Chapter 4, other AI approaches, such as evolutionary algorithms, are also not well suited for the task at hand.

Reinforcement Learning, however, may still provide a promising approach. Recent advances in Reinforcement Learning (RL) have enabled computers to find good solutions for a variety of challenges, e.g., RL methods can build near-optimal solutions (up to 100 nodes) for combinatorial problems such as the Travelling Salesmen Problem [82]. RL approaches combined with supervised learning can schedule DAG task graphs and outperform heuristics such as Heterogeneous Earliest Finish Time (HEFT), Critical-Path-on-a-Processor (CPOP) and Graphene [71, 140]. The combination of RL and supervised methods is required to stabilise the training. However, by pre-training neural networks in a supervised manner, they learn to imitate heuristics, whereas it is known that learning from scratch can outperform both heuristics and humans [124].

While RL schedulers are often used online (i.e. on the target platform at runtime), it is also possible to use them offline and generate a schedule before runtime. In an embedded or real-time setting this has two advantages. First, the schedule is known, can be validated and there is no chance that something confuses the neural networks involved and thus leads to a fatal error. Second, using the RL scheduler online in an embedded system may lead to significant overhead and take up valuable compute time. Especially the GPU time is valuable, considering that neural networks execute much faster on the GPU than on the CPU and that most embedded systems only have one GPU. In cases such as identifying poachers directly on wildlife cameras [74] this may lead to unacceptable overhead.

We propose a Q-learning based approach that learns to build offline schedules from scratch (i.e., not learning to imitate a heuristic) and incorporates graph information. We show that a Q-learning approach quickly learns static DAG scheduling. The generated schedules are, on average, slightly longer than a schedule generated by a FLS based scheduling algorithm (0.29% and 2.8% degradation for the two datasets).

6.1 BACKGROUND

Reinforcement learning is a framework for sequential decision making. The two main components of an RL approach are the environment and the agent. At each time step, the agent receives a state from the environment and returns an action. The environment processes the action and returns a reward to the agent. Over time the agent learns to maximise the cumulative expected reward by observing the states, actions and rewards. In the game

Pong[1], for example, the state is a frame from the game engine, the action is to move the paddle up or down, and the reward is 1 if the agent scores a point. In our case, the state contains the tasks, schedule and target architecture information. The action is where to schedule the next task, and the reward is the impact of the action on the makespan.

In this chapter, we consider the deep Q-learning (DQN) algorithm [95], a value-based method. The action-value function is approximated by a deep neural network which is normally unstable or divergent. The instability arises because we are dealing with time series and the training samples are correlated. That means the training sample at T and the training sample at $T+1$ are correlated as they follow each other in time. The method introduced by Mnih et al. [95] makes use of *experience replay*, where the neural network is not trained during scheduling but after. In *experience replay*, the agent's experiences (state, action, reward) are stored in a memory, and during training, the network is trained on batches of experience from memory. This has two advantages. 1) The data efficiency is improved as experiences are used several times for training. 2) The correlations between consecutive experiences are broken. The standard DQN algorithm is improved with several extensions (summarised in [65]) to speed up and stabilise the training process.

6.2 SYSTEM MODEL

Task Model. We consider applications represented as Directed Acyclic Graphs (DAG), hereafter called task graphs. In a graph, $G = (\tau, E)$ the set of nodes/vertices τ represents the tasks, and the set of edges E represents data dependencies between tasks, i.e., a source task needs to be completed before the corresponding sink task may start executing. Our task model supports multiple sources and sinks. Additionally, we support a multi-graph setup (i.e., multiple applications).

A task is a sub-part of an application that needs a certain input, then executes without additional input until it finalises and passes its output to the following tasks in the application. Each task has a timing budget. Our model does not limit the number of incoming or outgoing edges of a task. We assume that multiple tasks cannot run concurrently on one processing unit. In this chapter, in contrast to previous chapters, we do not consider multi-phase or multi-version tasks as our aim firstly is to establish whether or not RL based-scheduling is feasible for a straightforward base line case.

1 https://en.wikipedia.org/wiki/Pong

Architecture Model. Our approach is fully platform-independent and can be applied to a wide range of homogeneous system architectures. The number of CU can be altered via a parameter (the number of actions the agent can make). This is in contrast to previous chapters, where we focused on heterogeneous systems. We only focus on homogeneous systems to determine the feasibility of RL based schedulers. However, the model could easily be extended to heterogeneous systems by increasing the number of CU and including one additional feature (the runtime on the second core type).

6.3 RL SCHEDULING

This section provides a high-level overview of our RL scheduling approach. Figure 6.1 shows the interactions between the agent and the environment, following the classic RL approach described in Section 6.1.

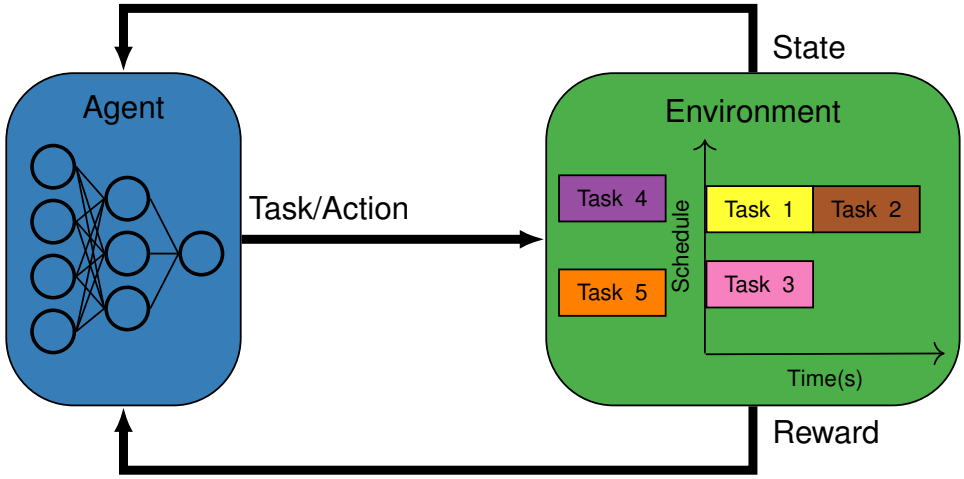


Figure 6.1: Reinforcement Learning Framework

The environment updates the ready-queue at the beginning of each scheduling step, i.e., collects all tasks that can be scheduled. Only tasks whose predecessors have already been scheduled can be scheduled. Hence, at any given point in time, we might have multiple tasks that can be scheduled.

Then the features of all tasks (states) in the ready-queue are collected and passed to the agent. The agent evaluates all eligible tasks and selects the task/action pair with the highest expected reward.

The task/action pair is returned to the environment, and the environment adds the task to the schedule in a *as-soon-as-possible* fashion, respecting both

predecessor run-times and preventing tasks from being scheduled to execute at the same time on the same CU. The environment then returns the reward for the task/action pair and the states for the next tasks in the ready-queue. This is repeated until all tasks in all applications have been added to the schedule.

6.4 RL SCHEDULER COMPONENTS

In this section, we give a short introduction to the various components of our reinforcement learning scheduler. One main part of the scheduler is the neural network agent that makes all the decisions. We have decided to investigate two different agents. The first one is based on a straightforward fully connected neural network. And the second one extends the first network by incorporating graph convolutional layers to extract additional information from the DAG. We want to investigate if the additional information improves the scheduler.

Environment. The environment contains the task graph, the schedule and a representation of the target architecture. It can evaluate the impact of different scheduling decisions and update its internal states.

Fully Connected Agent. The main backbone of our agent consists of a fully connected neural network (FCNN). The network consists of *4* layers having *2048, 2048, 4096, 4096* neurons, respectively. We use ELU activation functions after each layer [29]. This network architecture performed the best across a selection of different networks while searching the hyperparameter space. For the hyperparameter space search we use the bayesian sweep function provided by *Weights and Biases*[20].

The input to a neural network depends on the type of neural network used. For our FCNN based approach, the state is a list of features for a task (i.e. node and some global features that are common to all tasks). The node specific features are: 1) The runtime of a task. 2) The best start time at which a task can start (i.e. the end time across all predecessors). 3) The actual start time of a task if it has been scheduled. 4) The target core if a task has been scheduled. The global features are: 1) The normalised values of the min, max and mean of all tasks in the ready queue. 2) The normalised values of the min, max and mean runtime of all tasks in the done queue. 3) The number of tasks in the DAG. 4) The number of tasks available for scheduling. 5) The number of tasks that still need to be scheduled. All normalised features are normalised with the maximum runtime of any task in the graph. This results in a total of 13 features for each task.

Graph Convolutional Network Agent. As our problem is in the form of a graph it makes sense to extract additional information using a graph convolutional network (GCN) [80], which work similarly to convolutional neural networks (CNN). However, instead of applying different convolutional kernels to neighbouring pixels in an image, we apply convolutional kernels to neighbouring nodes in the graph. The input to a GCN consists of the node features and the edge information (i.e. which nodes are connected). For our purposes, we have identified 3 potential sources of useful information. First, a node's predecessors may hold important information. Second, all previously scheduled nodes may be of importance (i.e. they may hold information about gaps in the schedule). Third, all successors to a node may also hold important information. For each one of these sources, we create a four layer GCN (i.e. we have three GCNs). Each GCN consists of 4 SAGEConv [57] layers with *8, 16, 32, 64* neurons per layer respectively. Each layer is followed by an ELU activation function. The input to each GCN are the node information (runtime, best start time, actual start time and target core). However, the edge information for each GCN differ slightly depending on whether it is supposed to learn about predecessors, previously scheduled nodes or successor nodes. The output of the three GCNs is then concatenated, together with the original node information and the same global features as for the FCNN agent. All this information is fed into the same feed forward agent as above (4 layers with *2048, 2048, 4096, 4096* neurons, respectively) and ELU activation functions to return what is the most valuable action.

Besides FCNN and GCN layers, we also experiment with Long-Short-Term-Memory (LSTM) layers, 1D convolutional layers and 2D convolutional layers. The network consisting of LSTM layers does not perform well despite an extensive search of the hyper-parameter space. The 1D and 2D convolutional neural networks perform almost as good as the FCNN network.

Actions. The action is the CU on which a given task is scheduled. For example, in the case of a quad-core system, the action space is between 0 and 3. The number of possible actions depends on the target system.

Rewards. The reward function (Equation (49)) returns the value of a given state s_t. In our case, the reward is the negative release time ($-rt_0$) of the action (i.e. start time of a task) plus the expected reward of future actions, where γ is the discount rate ($[0, 1)$) of future actions. We used a γ of 0.65. There are no positive rewards; the best possible reward is 0. If a task (t_0) starts at the 5 second mark, the reward is -5 minus the expected reward of

future actions. That means if we expect the next task (t_1) to start at the 8 second mark, then the reward for t_0 is -13.

$$V(s_0) = -rt_0 + E_{t=1}^{\infty}\left[\gamma^t \times (-rt_t)\right] \tag{49}$$

RL approach description. We use a double DQN approach [61] with fixed Q-targets, where two networks (*NN1* and *NN2*) are initialised with the same weights. *NN1* and *NN2* are used to update each other. A simplified representation of a single training step is shown in Figure 6.2. During training, the environment passes a state (S_t) to *NN1*, which predicts the *expected reward* of all actions at step t. The action with the highest *expected reward* is selected and passed to the environment, which evaluates it and computes a *reward*. At the same time, the environment passes the updated state S_{t+1} to *NN2*, which predicts the *expected reward* for the new state. The *reward* at t is combined with the *expected reward* at *t+1* to form the *actual reward* at t. The *actual reward* is then used to update the weights of *NN1*. The weights of *NN2* are updated every α steps with the weights of *NN1*. *NN2* is the network used for inference.

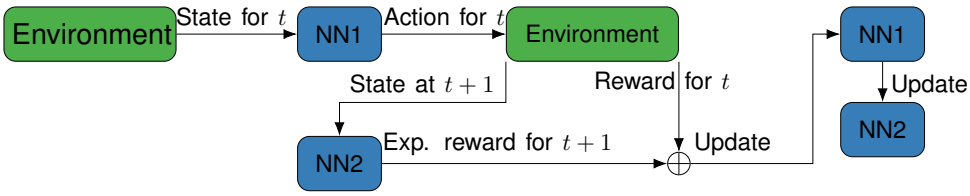


Figure 6.2: RL agent training pipeline.

We need *NN2* to compute the *expected reward* at *t+1*. If we would use *NN1* to predict both the *expected reward* at t and at step *t+1*, we would use the same neural network to calculate the *expected reward* and the *actual reward*. This would make the training process unstable. That way, the *expected reward* of *NN1* moves closer to the *actual reward* over time while preventing wide fluctuations during the training process.

Furthermore, our approach uses prioritised experience replay [119], where training samples of higher impact are more likely to be in the training batch. The impact of a sample is the absolute percentage difference between the predicted and the actual reward.

Hyperparameters. The updates between Neural Network 1 and 2 are not done at every step but at every τ steps. Our search of the hyperparameter space suggests that a value of 5 works well for this problem. Additionally, we use the Adam optimiser to minimise the L1 loss.

6.5 EXPERIMENTS

Data. We use Task Graphs For Free (*TGFF*) [41] to generate random DAG task graphs. TGFF produces both the structure of the task graphs and the runtime of the tasks. In total, there are 10000 different tasks (i.e., different runtimes) to improve the generalisation of the agent. We decided to run our experiments with two datasets. One dataset with smaller, less diverse and simple graphs, and a second dataset with large, diverse and complex graphs. Both datasets contain one test and one training dataset. The test and training datasets were generated individually with different seeds. Links to the dataset and code used in this chapter can be found on page 137.

The main difference between the two datasets is the number of tasks per graph. The small DAGs (Dataset 1) are set to 10 tasks with a multiplier of 1. This does not mean that all graphs have 10 tasks, as the number of tasks also depends on other characteristics. The larger taskgrahs (Dataset 2) are set to an average of 20 tasks with a multiplier of 5. This results in a wider distribution of the number of tasks in the graphs. Additionally, the two parameters 'series_len' and 'series_wid' differ. The 'series_len' parameter sets the number of tasks in a chain/series. Whereas, 'series_wid' sets the parallelism of a taskgraph. Both of these parameters are larger for the large DAG dataset. The 'series_len' parameter is set to [5, 2] (average, multiplier) and [8, 2] for the small and large DAG datasets respectively. The 'series_wid' parameter is set to [6, 2] and [8, 2] for the small and large DAG datasets respectively. This means that the larger DAGs are more challenging as they, for example, contain more potential parallelism, which is especially important as the target system only has 4 cores. All datasets are roughly uniformly distributed with respect to the number of tasks in a DAG.

The dataset of smaller DAGs consists of a training dataset with 10000 task graphs and a test dataset containing 1000 task graphs. Table 6.1 contains a summary of the graph statistics. The two datasets do not differ much with respect to the number of tasks in the DAGs.

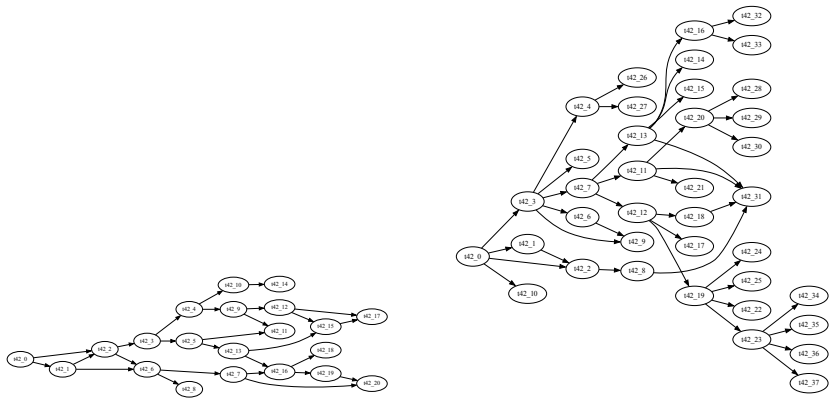
Table 6.1: The table shows the statistics with respect to the number of tasks in the train and test sets that contain the small DAGs.

	Mean	Min.	Max.	Std.
Small Train	12.7	6	24	6.3
Small Test	13.0	6	24	6.4

The dataset of larger DAGs consists of 10000 training graphs and 1000 test graphs. Table 6.2 summarises the statistics for the dataset containing larger graphs. The difference between the type of graphs generated for the two datasets can be well seen when comparing Figure 6.3a and Figure 6.3b.

Table 6.2: The table shows the statistics with respect to the number of tasks in the train and test sets that contain the large DAGs.

	Mean	Min.	Max.	Std.
Large Train	25.5	9	55	13.1
Large Test	25.5	9	54	13.4



(a) Example: DAG 42 from the small DAG test dataset.

(b) Example: DAG 42 from the large DAG test dataset.

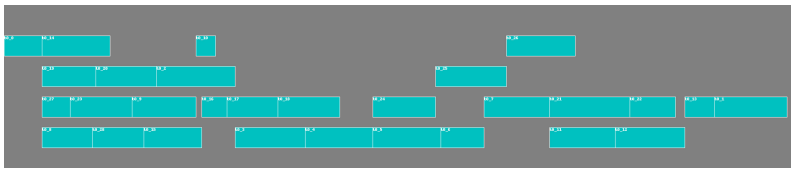
Comparison. We compare the RL generated schedules to Forward List Scheduler (FLS) generated schedules [32]. FLS first orders the tasks and then adds them one by one to the schedule without backtracking. FLS iteratively computes the impact on the makespan of scheduling a task on a specific CU and greedily selects the best CU with respect to the makespan. The performance of FLS heavily depends on the initial ranking of the tasks. Thus, it is common practice to try multiple ranking algorithms as none consistently outperforms the others [116]. In this case, we use 3 different rankings: BFS, DFS and BFS with Laxity.

Target System. In this chapter, we consider a system with 4 processing units as a proof of concept and to keep the amount of experimentation within reasonable bounds.

6.6 RESULTS

In general, the RL scheduler learns to schedule DAGs quickly. Figure 6.4a shows the schedule produced by an untrained, randomly initialised RL agent. We can see that all 27 tasks from the original graph are just mindlessly put after each other on a single core. However, after some training the situation is much improved. Figure 6.4b shows a schedule produced for the same graph by the same RL agent after some training. The decisions are not necessarily optimal but we can clearly observe that the scheduler learns that distributing tasks over different cores is better (i.e. increases its rewards).

(a) Schedule of a taskgraph with 27 tasks produced by one of our untrained RL schedulers.



(b) Schedule of a taskgraph with 27 tasks produced by one of our trained RL schedulers.

In Sections 6.6.1 to 6.6.4, we will discuss the performance of the two different schedulers (FCNN and GCN) with regard to the two different datasets (Dataset 1 & Dataset2). All four combinations were allowed to train for a similar number of epochs and the best performing neural network was selected.

6.6.1 *Dataset 1 - FCNN Agent*

The FCNN agent performs quite well on the dataset consisting of smaller DAGs. The degradation distribution between the FCNN agent and the FLS scheduler can be seen in Figure 6.5. In 69.6% of the cases the FCNN agent produces schedules that are the same or better. In 90.0% of the DAGs the FCNN agent results in schedules that perform similarly (at most 5% degradation) or better. The average degradation is 1.2% and at best the resulting schedule is 6.9% shorter than the schedule generated by the FLS approach. At worst the FCNN scheduler results in a 19.7% higher makespan.

Despite this good performance the FCNN scheduler had a L1Loss of 35.9 which is quite high compared to the L1Loss of the GCNN scheduler.

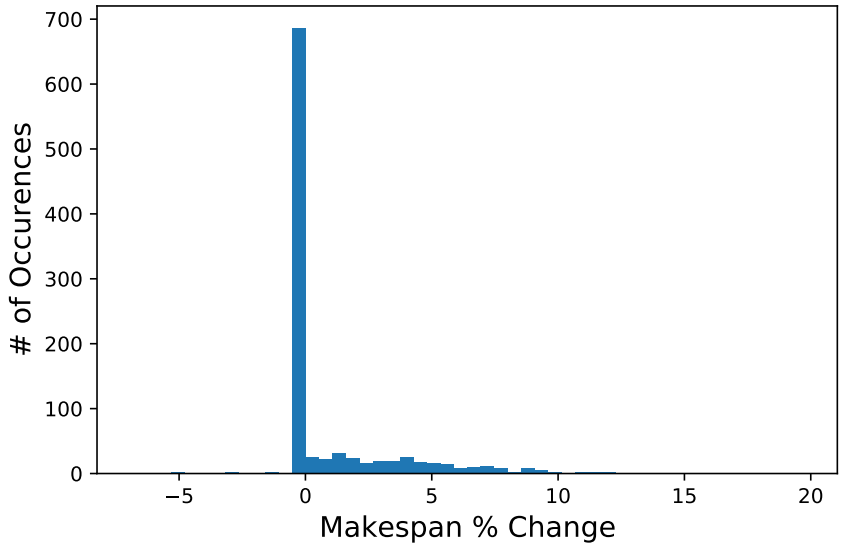


Figure 6.5: Makespan degradation of the small DAG test dataset between the FCNN generated schedules and the FLS schedules.

6.6.2 *Dataset 2 - FCNN Agent*

The FCNN agent performs significantly worse for the dataset containing larger DAGs than for the dataset of small DAGs. The degradation spread is shown in Figure 6.6. Overall, the FCNN agent only manages to produce schedules that are the same or better in 18.9% of the DAGs. Additionally, it finds schedules that perform similarly (at most 5% degradation) or better in only 42.3% of the cases. Overall, the degradation is 7.1%. And at best, the generated schedule results in 6.5% lower makespan but at worst we see a degradation of 35.3%. The L1Loss (45.5) is higher than the L1Loss in Section 6.6.1. Showing that the additional complexity of the large DAGs and possibly the larger variance of DAGs may require a more advanced approach.

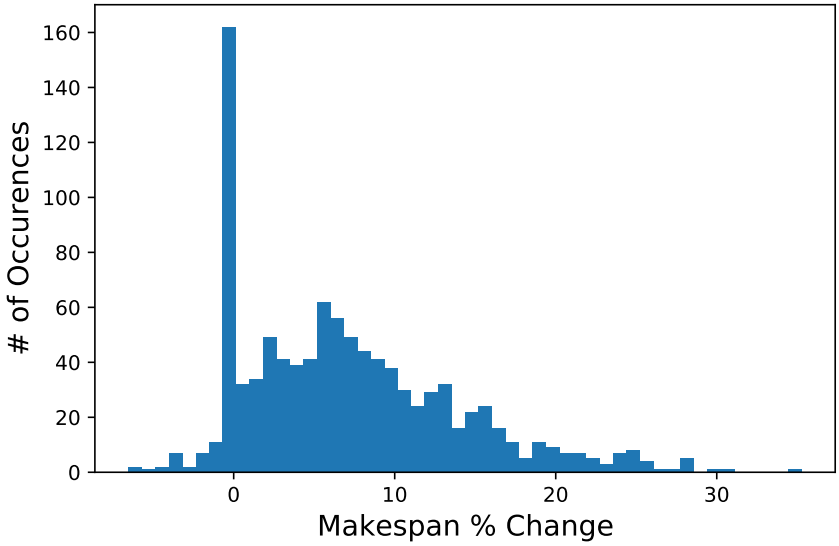


Figure 6.6: Makespan degradation of the large DAG test dataset between the FCNN generated schedules and the FLS schedules.

6.6.3 *Dataset 1 - GCN Based network*

The GCN based agent performs better than the FCNN agent. However, the difference is not as significant as the performance drop between Section 6.6.1 and Section 6.6.2. The degradation is shown in Figure 6.7. The distribution looks similar to the one shown in Figure 6.5. Overall, the GCN approach generates schedules that are the same or better in 85.2% of the cases. And it finds schedules that perform similarly (at most 5% degradation) or better in 98.1% of the cases. The average degradation is 0.29%. At best the schedule is 6.9% shorter and at worst the found schedule has a 20.4% longer makespan. One more difference between the FCNN scheduler and the GCN scheduler is the much lower L1Loss, which dropped to 5.9. This clearly shows that the three GCNs provide valuable information, even though, the information do not appear to add much value in the case of the smaller DAG dataset.

6.6.4 *Dataset 2 - GCN Based network*

We can see a clear improvement in the schedules generated by the GCN scheduler in comparison to the FCNN scheduler for the dataset of large DAGs. This improvement can also be seen when comparing the degradation

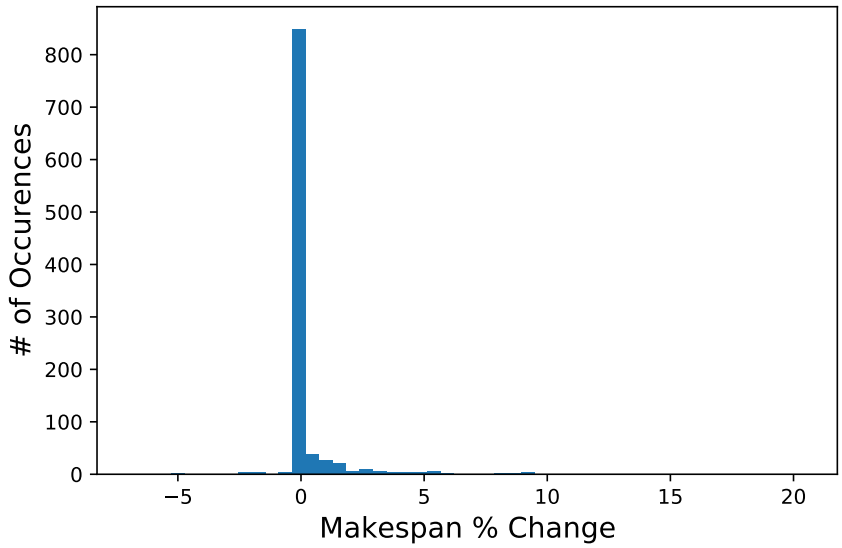


Figure 6.7: Makespan degradation of the small DAG test dataset between the GCN based RL scheduler and the FLS scheduler.

distributions in Figure 6.8 (GCN scheduler) and Figure 6.6 (FCNN scheduler). In total, we find that the GCN agent generates schedules that are the same or better in 38.7% of the cases. Furthermore, the GCN scheduler finds schedules that perform similarly (at most 5% degradation) or better in 75.6% of the cases. The average degradation drops from 7.1% for the FCNN agent to 2.8% for the GCNN agent. At best we see schedules that are 11.2% shorter and at worst the schedules are 34.4% longer than the FLS generated schedules. The final L1Loss is 11.0. In comparison, to the GCN approach on smaller DAGs this L1Loss is slightly higher. However, the L1Loss is also significantly lower than the L1Loss of the FCNN agent. This clearly shows that the additional information provided by the GCN layers is valuable.

Across all four combinations we do not observe any differences in how well or poorly a RL scheduler does with respect to the number of tasks in the taskgraph, i.e. a larger taskgraph does not necessarily lead to a higher degradation.

6.7 RELATED WORK

Wu et al. [140] use the REINFORCE agent [139] from 1992 to schedule DAG taskgraphs. The paper shows that this approach outperforms Heterogeneous

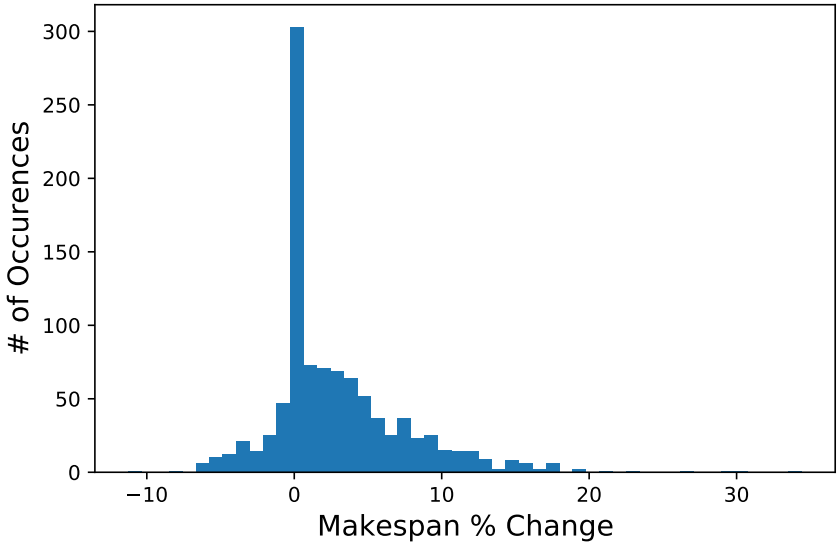


Figure 6.8: Makespan degradation of the large DAG test dataset between the GCN based RL scheduler and the FLS scheduler.

Earliest Finish Time (HEFT) and Critical-Path-on-a-Processor (CPOP) by up to 25%. However, REINFORCE agents tend to be unstable in the training process. More modern approaches like our approach address this stability issue. Furthermore, the approach by Wu et al. depends on the original ranking of the tasks in the task graph.

Hu, Tu and Li [71] have proposed a new approach (called *Spear)* that uses Monte Carlo Tree Search (MCTS) combined with RL. *Spear* outperforms the Graphene heuristic by 20%. *Spear* determines the ranking of the tasks, i.e., it determines in what order the tasks are scheduled, whereas we use RL to schedule the task end-to-end. Additionally, *spear* initialises the network with supervised learning, i.e., it learns to imitate the behaviour of a heuristic. This means that the agent might learn undesirable behaviour from the heuristic. And is exactly the opposite of what we want, as it has been shown that RL agents are capable of learning strategies on their own and, in some cases outperforming both humans and heuristics [124].

Mao et al. [91] use Reinforcement Learning to schedule independent tasks, whereas we focus on dependent tasks. Hu et al. [70] introduce an RL agent for online scheduling of dependent tasks. Our approach focuses on offline scheduling as online scheduling can incur a high overhead on high-performance embedded systems.

6.8 CONCLUSION

Finding near-optimal static schedules for large DAGs in a reasonable solving time is still an open problem. To the best of our knowledge, we are the first to use DQN Reinforcement Learning to tackle this problem in an end-to-end fashion.

We show that RL-based schedulers can outperform FLS-based schedulers. The resulting schedules are up to 11.2% shorter than the corresponding FLS generated schedules. For the small DAG dataset (Dataset 1) our GCN approach generates schedules that are at most 5% worse in 98.1% of the cases. Furthermore, our experiments show that the additional information obtained by the GCN layers add value to our RL-based scheduler. However, this additional information only seems to result in significantly better schedules (on average) if the target dataset is more diverse or contains larger taskgraphs. Furthermore, we show that the selected reward function works (i.e. lower loss = better performance).

For the future we plan to investigate the use of sparse rewards, i.e. the RL scheduler is only rewarded at the end of the scheduling. Additionally, we plan to experiment with policy learning instead of action-value learning. And lastly we plan a detailed analysis on which of the three GCNs adds most value.

7 CONCLUSION

Single-board computers that use heterogeneous MPSoC's have become widely available in recent years. The hardware of such single-board computers is similar to modern smartphones. MPSoC offer an excellent energy-to-performance ratio. However, MPSoC are also more challenging to program than simple embedded systems as one must engineer concurrent applications to take advantage of the multiple heterogeneous compute units. We model applications as DAG's and assume that they can be broken down into different parts (i.e. different tasks) for concurrent execution. The scheduling of the different tasks can heavily influence the final application's execution characteristics (e.g. energy consumption, runtime). In this thesis, we propose several novel scheduling methods and tackle essential system criteria such as energy consumption and energy modelling.

7.1 CONTRIBUTIONS

The main contributions of this thesis are as follows:

Sampling frequency analysis of energy-measurement setups. Our first main contribution is the analysis of a large set of power traces collected from the Odroid-XU4 using the Qoitech Otii. Using these power traces, we analysed the minimum sampling frequency required to measure energy consumption accurately. Links to the collected dataset and the analysis scripts can be found on page 137.

Multi-version task model. Our second contribution is the introduction of a new multi-version task model. Multi-version tasks are required to take full advantage of heterogeneous hardware due to binary incompatibility of different compute units. By supporting a multi-version approach our scheduler can decide between different hardware units depending on the requirements of the application. Additionally, multi-version tasks can be used to, for example, decide between two different yet functionally-equivalent algorithms during scheduling.

Fine-grained energy model. Our third contribution is our new energy model which considers the impact of the specific DVFS settings available on the target platform. Furthermore, our energy model uses different dynamic

energy consumption per task, instead of assuming a one-size-fits-all energy consumption.

Energy-aware scheduling and ranking approach. Our fourth contribution is the introduction of a new energy-aware scheduling method in conjunction with our new ranking approach. Additionally, this new scheduler also incorporates the multi-version task model and our energy model. Links to the datasets used and the full implementation of both the scheduler and the connected ILP formulation can be found on page 137.

Multi-phase task model. Our fifth contribution is the extension of our multi-version task model to a multi-phase multi-version task model. This extension allows a much finer grained scheduling of applications that contain tasks that can target accelerators such as GPUs. As our multi-phase approach also supports multiple versions a task can still target either the CPU or both a CPU and an accelerator. This more fine-grained scheduling allows us to successfully schedule applications with high hardware utilisation levels. Links to the datasets used and the implementation of both the scheduler and the connected ILP formulation can be found on page 137.

Reinforcement Learning based scheduler. Our last contribution is the introduction of a RL based scheduler which for now targets a straightforward DAG task model and a homogenous quad-core system. Links to the datasets used and the implementation of the RL agent can be found on page 137.

7.2 ANSWERS TO THE RESEARCH QUESTIONS

We can now answer the research questions from Chapter 1 using these contributions and new methods.

- *RQ 1: What is the impact of the sampling frequency on energy measurement accuracy?*

Reducing energy consumption of high-performance embedded systems is a popular research topic. However, researchers often dedicate surprisingly little space and attention to their energy measurement setup. Chapter 3 demonstrates the importance of a well-designed energy measurement setup. We collect over 42000 power traces and, using rigorous statistical analysis, show that the sampling frequency can significantly impact the accuracy. Figures 3.4 and 3.5 perfectly visualise the impact of a low sampling frequency. For the applications and the device tested, a low sampling rate (1Hz) can lead to up to 80% error in the energy measurement. Additionally, we need

to measure energy consumption at least at 350Hz to achieve a measurement equivalent (less than 1% error) to a high-frequency measurement (4kHz).

- *RQ 2: How can we improve the energy consumption of DAG applications for heterogeneous high-performance embedded systems through scheduling?*

Improving the energy efficiency of high-performance embedded systems requires a multi-facet approach. First, as explained in Chapter 2, DVFS is a crucial component of a modern high-performance embedded system, and different parts of an MPSoC have different possible DVFS settings. Furthermore, as shown in Figure 2.2, different applications have different DVFS sweet spots. Thus, we need to consider DVFS to reduce energy consumption. Second, we need a multi-task approach to target the different binary incompatible compute units. Our new task model is presented in Section 4.2.2. Lastly, we need an energy model that can predict the impact of various scheduling decisions to make good scheduling decisions. We propose a new fine-grained energy model in Section 4.2.3.

Our new scheduling approach considers the DVFS settings of different compute units and can model energy consumption during scheduling. In Chapter 4, we detail the different components required for reducing energy consumption through scheduling. Furthermore, based on extensive experimentation, we can show that our novel approach outperforms the state-of-the-art scheduling methods ARSH-FATI and HEFT.

- *RQ 3: What is the importance of the ranking algorithm used in our energy-aware scheduling approach?*

Throughout this thesis, we represent applications as Directed Acyclic Graphs. By nature, DAG's provide a partial order of tasks. However, many scheduling approaches (e.g., HEFT, ARSH-FATI, FLS, eFLS) establish or require a total order of tasks. Our ranking strategy analysis in Section 4.5 shows that different ranking strategies can lead to more than 20% difference in energy consumption. We demonstrate that no one ranking algorithm always leads to the best result. Furthermore, we show that our new energy-aware ranking strategy (HER) outperforms established ranking strategies such as BFS and DFS with respect to energy consumption by up to 23%. Additionally, our new ranking strategy also outperforms our energy-aware multi-version HEFT approach by 1.7%.

- *RQ 4: How can we increase the hardware utilisation of heterogeneous high-performance embedded systems?*

In order to further increase hardware utilisation, we need a more fine-grained approach to scheduling tasks that require both the CPU and GPU-style accelerators. Hence, we introduce a new multi-phase task model in Chapter 5. Having multiple phases means splitting tasks targeting a GPU-style accelerator into smaller chunks. In general, tasks targeting the GPU normally start on the CPU which calls a GPU kernel and then are finalised on the CPU. If we split a task into phases we can alternate between the CPU and GPU instead of just reserving both compute units for the full execution time. Splitting a task into phases, thus, allows for better utilisation of the underlying hardware.

This raises the question: Why can we not just split tasks into multiple tasks instead of complicating the task model? The answer is relatively straightforwards: In a multi-version model, splitting a task into multiple tasks could result in a mix of multiple versions, which would be erroneous. Our novel multi-phase approach significantly increases the schedulability of task graphs with high hardware utilisation compared to eFLS and HEFT. For instance, in the utilisation range, 11 to 12 our method can successfully schedule twice as many task graphs as HEFT as can be seen in Figure 5.8.

- *RQ 5: To what extent can we use reinforcement learning to replace traditional scheduling methods?*

Our novel RL-based approach in Chapter 6 quickly learns to schedule DAG based applications offline. One significant advantage over traditional schedulers is that the RL scheduler can evaluate all possible scheduling options at once. This avoids the necessity to first rank all the tasks in a DAG and can genuinely make the most greedy decision. We show that our RL approach can produce schedules that are up to 11% shorter than schedules generated by a greedy heuristic. Additionally, we show that across a large number of DAGs (1000 test DAGs) the RL-generated schedules perform similarly or better in over 75% of the cases in comparison to a greedy heuristic. The RL-generated schedules are on average 2.8% longer. Furthermore, we demonstrate that graph convolutional networks can extract useful information and incorporating them into the RL-based scheduler improves the schedules generated for large taskgraphs.

7.3 FUTURE WORK

Recently we have seen an increasing number of heterogeneous computer architectures. Nowadays, we see smartphones with up to 3 different CPU

core types. Additionally, this advancement is not only limited to small mobile devices. For example, the latest ARM-based MacBook Pro models use a big.LITTLE approach with multiple different accelerators (e.g., GPU, neural engine). Intel also introduced a hybrid/heterogeneous architecture with their 12th Gen Intel Core processors. Hence, deciding when and where a task should be executed will only gain importance. Thus, we need to continue working on using these systems efficiently.

7.3.1 *Energy Measurements*

Concerning energy measurements, we would like to establish an easy-to-follow community-based set of guidelines for energy measurements in the high-performance embedded systems area to avoid problems and confusion. Establishing guidelines could be beneficial for groups and people who are new to measuring energy consumption and would like to contribute to the research on energy efficiency. Presenting good guidelines also includes analysing more systems in depth and extending the test methodology to, for example, include multi-core and deep learning workloads (e.g. computer vision).

7.3.2 *Scheduling Heuristics*

On the scheduling side, we would like to combine our work presented in Chapters 4 and 5. Thus, we would like to incorporate our multi-phase approach into the energy-aware scheduler and test the impact on energy consumption. Furthermore, we want to extend our approach to account for more system properties, such as: memory, cache, bus occupancy and temperature. Taking additional system properties into account could improve our energy consumption prediction. Incorporating the bus and other properties could decrease contention or at least lead to a more accurate prediction of contention.

On the application side, we would like to extend our application model to account for different states / modes, where each state could potentially have its own DAG. That would allow applications to switch between states depending on the current environment and thereby change the behaviour of the application. For example, a drone on a search and rescue mission might switch to a tracking state if it finds people in the water. This tracking state might have a different DAG representing it. Thus, it would need a separate schedule. This extension could be seen as an overarching approach to our

scheduling methods, where the different states need their own schedule and the application can switch between states and schedules at runtime depending on the environment.

7.3.3 *RL for Scheduling*

Next to exploring and extending traditional scheduling algorithms, we also plan to continue our work on RL for scheduling. First, we want to analyse for which types of DAG our current method performs well. Additionally, we plan to investigate the impact of target platform complexity (e.g., heterogeneity) on the quality of generated schedules. Extending our approach to a heterogeneous system is possible by altering the state the agent receives from the environment and the size of actions passed back to the environment. Lastly, we want to explore the robustness of the schedule with changing task-graph size and complexity.

7.4 VISION AND OUTLOOK

Overall we expect that scheduling applications for high-performance embedded systems will only continue to become more complicated due to multiple reasons. First, the Odroid-XU4 considered in this work is only the beginning of heterogeneity. Compute platforms are becoming more and more heterogeneous. The NVidia Jetson AGX Orin 64GB includes three different CPU-core clusters (12 cores in total), two powerful NVidia internal GPU's (iGPU), a separate NVidia deep learning accelerator, various hardware encoders and decoders, an optical flow accelerator and more [100]. Second, we have not yet integrated all crucial design factors into our scheduling methods (e.g. cache, bus, temperature). The increasing complexity of heterogeneous systems and the need to consider more design factors means that the scheduling state space is exploding.

A state space that is even larger than the one we considered in this work means that ILPs will not be solvable in a reasonable amount of time for any problem size. The run-time of heuristics will also increase to the point where it will impact the development of products using modern MPSoC. Thus, we envision a future where schedulers are even more automated to the point where the end-user (e.g. self-driving car developer) simply lets an automatic scheduling system analyse a new piece of hardware. The analyses then result in accurate models for energy consumption, temperature, bus contention etc. without any human interaction. The end-user can then make use of

the resulting models and learned behaviour to quickly generate schedules. Part of such a system may be heuristics. However, we can also imagine that such scheduling systems will employ machine learning methods such as reinforcement learning.

Despite the fact that our RL-based approach does not yet generate better schedules across the board in comparison to an FLS scheduler, we still see it as a promising way forward with plenty of room for improvement. Thus, reinforcement learning seems to be one of the best ways to realise a fully automatic scheduling system which does not require expensive human interaction. We imagine a RL agent that is first trained in a simplistic simulator, and then is refined on real hardware. The agent generates a new schedule, which is executed on hardware and then the agent learns from the result until it contains an accurate model of the hardware. The model of the hardware can then be used to schedule new applications. Additionally, we could retrain the model for new hardware (e.g. a newer version of the same platform) and obtain another accurate hardware model without fully modelling and reverse-engineering the new hardware. Ideally retraining the RL agent for new platforms would be a plug-and-train approach resulting in an RL agent that generates competitive schedules with minimal human interaction.

BIBLIOGRAPHY

[1] ARM Ltd. "White Paper: big.LITTLE Technology : The Future of Mobile." In: (2013).

[2] F. Adelantado, X. Vilajosana, P. Tuset-Peiro, B. Martinez, J. Melia-Segui, and T. Watteyne. "Understanding the limits of LoRaWAN." In: *IEEE Communications magazine* 55.9 (2017), pp. 34–40.

[3] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis. "An empirical survey-based study into industry practice in real-time systems." In: *RTSS*. 2020.

[4] K. Al-Kodmany. "Tall buildings and elevators: A review of recent technological advances." In: *Buildings* 5.3 (2015), pp. 1070–1104.

[5] L. Al-Sharif, Z. Yang, A. Hakam, and A. Abd Al-Raheem. "Comprehensive analysis of elevator static sectoring control systems using Monte Carlo simulation." In: *Building Services Engineering Research and Technology* 39.5 (2018), pp. 518–539.

[6] S. Albers and A. Antoniadis. "Race to idle: new algorithms for speed scaling with a sleep state." In: *ACM Transactions on Algorithms (TALG)* 10.2 (2014), pp. 1–31.

[7] S. Aldegheri, N. Bombieri, and H. Patel. "On the task mapping and scheduling for DAG-based embedded vision applications on heterogeneous multi/many-core architectures." In: *DATE*. 2020, pp. 1003–1006.

[8] S. Altmeyer and C. M. Burguière. "Cache-related preemption delay via useful cache blocks: Survey and redefinition." In: *J. of Systems Architecture* 57.7 (2011), pp. 707–719.

[9] S. U. Amin and M. S. Hossain. "Edge intelligence and internet of things in healthcare: a survey." In: *IEEE Access* 9 (2020), pp. 45–59.

[10] IoT Analytics. *State of IoT 2022: Number of connected IoT devices growing 18% to 14.4 billion globally*. https://iot-analytics.com/number-connected-iot-devices/. Accessed: 2022-08-09. 2022.

[11] A. S. G. Andrae. "New perspectives on internet electricity use in 2030." In: *Engineering and Applied Science Letters* 3.2 (2020), pp. 19–31.

[12] H. Arabnejad and J. G. Barbosa. "List scheduling algorithm for heterogeneous systems by an optimistic cost table." In: *IEEE TPDS* 25.3 (2013), pp. 682–694.

[13] S. Arar. *Resistive Current Sensing: Low-Side vs. High-Side Sensing*. Accessed on 21.04.2022. URL: https://www.allaboutcircuits.com/technical-articles/resistive-current-sensing-low-side-versus-high-side-sensing/.

[14] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, et al. "The landscape of parallel computing research: A view from berkeley." In: (2006).

[15] E. Badidi. "Edge AI and Blockchain for Smart Sustainable Cities: Promise and Potential." In: *Sustainability* 14.13 (2022), p. 7609.

[16] A. Balsini, L. Pannocchi, and T. Cucinotta. "Modeling and simulation of power consumption and execution times for real-time tasks on embedded heterogeneous architectures." In: *ACM SIGBED Review* 16.3 (2019), pp. 51–56.

[17] M. Bambagini, M. Marinoni, H. Aydin, and G. Buttazzo. "Energy-Aware Scheduling for Real-Time Systems." In: *TECS* 15.1 (2016). ISSN: 15399087.

[18] Y. Bapin, K. Alimanov, and V. Zarikas. "Camera-driven probabilistic algorithm for multi-elevator systems." In: *Energies* 13.23 (2020), p. 6161.

[19] A. Bhuiyan, Z. Guo, A. Saifullah, N. Guan, and H. Xiong. "Energy-efficient real-time scheduling of dag tasks." In: *TECS* 17.5 (2018). ISSN: 15583465.

[20] L. Biewald. *Experiment Tracking with Weights and Biases*. Software available from wandb.com. 2020. URL: https://www.wandb.com/.

[21] S. Birrell, J. Hughes, J. Y. Cai, and F. Iida. "A field-tested robotic harvesting system for iceberg lettuce." In: *Journal of Field Robotics* 37.2 (2020), pp. 225–245.

[22] L. F. Bittencourt, R. Sakellariou, and E. R. M. Madeira. "Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm." In: *IEEE PDP*. 2010, pp. 27–34.

[23] E. Black and L. Kolodny. *This robot can pick tomatoes without bruising them and detect ripeness better than humans*. https://www.cnbc.com/2019/05/11/root-ai-unveils-its-tomato-picking-robot-virgo.html. Accessed: 02-09-2022.

[24] N. Brouwers, M. Zuniga, and K. Langendoen. "Neat: A novel energy analysis toolkit for free-roaming smartphones." In: *SenSys*. 2014, pp. 16–30.

[25] G.G. Brown and R.F. Dell. "Formulating integer linear programs: A rogues' gallery." In: *ITE* 7.2 (2007).

[26] M. Buschhoff, C. Günter, and O. Spinczyk. "MIMOSA, a highly sensitive and accurate power measurement technique for low-power systems." In: *Real-World Wireless Sensor Networks*. Springer, 2014, pp. 139–151.

[27] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. "Rodinia: A benchmark suite for heterogeneous computing." In: *IISWC*. Ieee. 2009, pp. 44–54.

[28] J. F. Clancy. "Automatic leveling device for elevators." Pat. 2044152.

[29] D. Clevert, T. Unterthiner, and S. Hochreiter. "Fast and accurate deep network learning by exponential linear units (elus)." In: *arXiv preprint arXiv:1511.07289* (2015).

[30] M. F. Cloutier, C. Paradis, and V. M. Weaver. "A raspberry pi cluster instrumented for fine-grained power measurement." In: *Electronics* 5.4 (2016), p. 61.

[31] A. Colin, A. Kandhalu, and R. Rajkumar. "Energy-Efficient Allocation of Real-Time Applications onto Single-ISA Heterogeneous Multi-Core Processors." In: *J. of Signal Processing Systems* 84.1 (2016). ISSN: 19398115.

[32] K. D. Cooper, P. J. Schielke, and D. Subramanian. "An Experimental Evaluation of List Scheduling." In: *TR98* 326 (1998).

[33] *Cortex-A15 Technical Reference Manual*. https://developer.arm.com/documentation/ddi0438/d/Level-2-Memory-System/About-the-L2-memory-system. Accessed: 2021-05-31.

[34] R. D'Agostino and E. S. Pearson. "Tests for departure from normality. Empirical results for the distributions of $b^2$and $\sqrt{b^1}$." In: *Biometrika* 60.3 (1973), pp. 613–622.

[35] X. Dai, I. Spasić, B. Meyer, S. Chapman, and F. Andres. "Machine learning on mobile: An on-device inference app for skin cancer detection." In: *2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC)*. IEEE. 2019, pp. 301–305.

[36] M. Dangana, S. Ansari, Q. H. Abbasi, S. Hussain, and M. A. Imran. "Suitability of NB-IoT for indoor industrial environment: A survey and insights." In: *Sensors* 21.16 (2021), p. 5284.

[37] T. Darwish and M. Bayoumi. "Trends in low-power VLSI design." In: *The Electrical Engineering Handbook* (2005), pp. 263–280.

[38] R.I. Davis and A. Burns. "A survey of hard real-time scheduling algorithms for multiprocessor systems." In: *ACM Computing Surveys* (2011).

[39] Y. De Bock, S. Altmeyer, T. Huybrechts, J. Broeckhove, and P. Hellinckx. "Task-set generator for schedulability analysis using the TACLeBench benchmark suite." In: *ACM SIGBED Review* 15.1 (2018).

[40] K. De Vogeleer, G. Memmi, P. Jouvelot, and F. Coelho. "The energy/frequency convexity rule: Modeling and experimental validation on mobile devices." In: *PPAM*. Springer. 2013, pp. 793–803.

[41] R.P. Dick, D.L. Rhodes, and W. Wolf. "TGFF: task graphs for free." In: *6th CODES/-CASHE*. IEEE. 1998.

[42] DieselDucy. *Historic Manually Controlled Self Leveling Otis Elevator @ Hotel Lawrence Dallas TX*. https://www.youtube.com/watch?v=qEphygZ2hAc. Accessed: 2022-31-10.

[43] M. E. M. Diouri, M. F. Dolz, O. Glück, L. Lefèvre, P. Alonso, S. Catalán, R. Mayo, and E. S. Quintana-Ortí. "Solving some mysteries in power monitoring of servers: Take care of your wattmeters!" In: *EE-LSDS*. Springer. 2013, pp. 3–18.

[44] A. Djupdal, B. Gottschall, F. Ghasemi, and M. Jahre. "Lynsyn and LynsynLite: The STHEM power measurement units." In: *Towards Ubiquitous Low-power Image Processing Platforms*. Springer, 2021, pp. 93–114.

[45] Z. Dong, Y. Lu, G. Tong, Y. Shu, S. Wang, and W. Shi. "WatchDog: Real-Time Vehicle Tracking on Geo-Distributed Edge Nodes." In: *ACM Trans. Internet Things* (2022). Just Accepted. ISSN: 2691-1914. DOI: 10.1145/3549551. URL: https://doi.org/10.1145/3549551.

[46] EEMBS. https://www.eembc.org/ulpmark/. Accessed: 18.11.2022.

[47] EEMBS. https://www.eembc.org/adasmark/. Accessed: 18.11.2022.

[48] EEMBS. https://www.eembc.org/mlmark/. Accessed: 18.11.2022.

[49] *Exynos 5 Octa 5422 Processor: Specs, Features: Samsung Exynos*. 2019. URL: https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-5-octa-5422/.

[50] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R.B. Sørensen, P. Wägemann, and S. Wegener. "TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research." In: *16th WCET*. Vol. 55. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016.

[51] M. Garrett. "Powering down." In: *Communications of the ACM* 51.9 (2008), pp. 42–46.

[52] M.E.T. Gerards, J.L. Hurink, and P.K.F. Hölzenspies. "A survey of offline algorithms for energy minimization under deadline constraints." In: *J. of Scheduling* 19.1 (2016). ISSN: 10946136.

[53] D. Grewe and M. F. P. O'Boyle. "A static task partitioning approach for heterogeneous systems using OpenCL." In: *CC*. 2011, pp. 286–305.

[54] I. Griva, S.G. Nash, and A. Sofer. *Linear and nonlinear optimization*. Vol. 108. Siam, 2009.

[55] Z. Guo, A. Bhuiyan, D. Liu, A. Khan, A. Saifullah, and N Guan. "Energy-efficient real-time scheduling of DAGs on clustered multi-core platforms." In: *RTAS* 2019-April (2019). ISSN: 15453421.

[56] Z¿ Guo, A. Bhuiyan, Di Liu, A. Khan, A. Saifullah, and N. Guan. "Energy-efficient real-time scheduling of DAGs on clustered multi-core platforms." In: *RTAS*. IEEE. 2019, pp. 156–168.

[57] W. Hamilton, Z. Ying, and J. Leskovec. "Inductive representation learning on large graphs." In: *Advances in neural information processing systems* 30 (2017).

[58] J. Han, M. Lin, D. Zhu, and L. T. Yang. "Contention-aware energy management scheme for NoC-based multicore real-time systems." In: *TPDS* 26.3 (2014), pp. 691–701.

[59] Hardkernel Co., Ltd. *Odroid-XU4*. https://wiki.odroid.com/odroid-xu4/odroid-xu4. Accessed: 2019-09-06.

[60] C. Hartmann and U. Margull. "Gpuart-an application-based limited preemptive gpu real-time scheduler for embedded systems." In: *J. of Systems Architecture* 97 (2019), pp. 304–319.

[61] H. Hasselt. "Double Q-learning." In: *24th NIPS* 23 (2010), pp. 2613–2621.

[62] S. Henn. *Remembering When Driverless Elevators Drew Skepticism*. https://www.npr.org/2015/07/31/427990392/remembering-when-driverless-elevators-drew-skepticism?t=1661773736778. Accessed: 2022-08-10. 2015.

[63] T. A. Henzinger, C. M. Kirsch, M. A. A. Sanvido, and W. Pree. "From control models to real-time code using Giotto." In: *IEEE Control Systems Magazine* 23.1 (2003), pp. 50–64.

[64] A. Hergenröder and J. Furthmüller. "On energy measurement methods in wireless networks." In: *ICC*. IEEE. 2012, pp. 6268–6272.

[65] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. "Rainbow: Combining Improvements in Deep Reinforcement Learning." In: *arXiv:1710.02298* (2017).

[66] G. Hoover. *Two Billion Passengers a Day: The Otis Story*. https://americanbusinesshistory.org/two-billion-passengers-a-day-the-otis-story/. Accessed: 2022-08-10. 2021.

[67] Z. Houssam-Eddine, N. Capodieci, R. Cavicchioli, G. Lipari, and M. Bertogna. "The HPC-DAG task model for heterogeneous real-time systems." In: *IEEE Transactions on Computers* (2020).

[68] B. Hu, Z. Cao, and Z. Zhou. "Energy-minimized scheduling of real-time parallel workflows on heterogeneous distributed computing systems." In: *IEEE Trans. Serv. Comput.* (2021).

[69] J. Hu, Y. Shin, N. Dhanwada, and R. Marculescu. "Architecting voltage islands in core-based system-on-a-chip designs." In: *Proceedings of the 2004 international symposium on Low power electronics and design*. 2004, pp. 180–185.

[70] Y. Hu, C. de Laat, and Z. Zhao. "Learning workflow scheduling on multi-resource clusters." In: *2019 IEEE International Conference on Networking, Architecture and Storage (NAS)*. IEEE. 2019, pp. 1–8.

[71] Z. Hu, J. Tu, and B. Li. "Spear: Optimized Dependency-Aware Task Scheduling with Deep Reinforcement Learning." In: *39th ICDCS*. IEEE. 2019, pp. 2037–2046.

[72] P. Huang, P. Kumar, G. Giannopoulou, and L. Thiele. "Energy efficient DVFS scheduling for mixed-criticality systems." In: *EMSOFT* 354 (2014).

[73] T. Ilsche, D. Hackenberg, S. Graul, R. Schöne, and J. Schuchart. "Power measurements for compute nodes: Improving sampling rates, granularity and accuracy." In: *IGSC*. IEEE. 2015, pp. 1–8.

[74] Argonaut — Archangel Imaging. https://www.archangel.im/product-page/argonaut. Accessed: 2022-08-10.

[75] C. Imes, D. HK. Kim, M. Maggio, and H. Hoffmann. "POET: a portable approach to minimizing energy under soft real-time constraints." In: *RTAS*. IEEE. 2015, pp. 75–86.

[76] S. Isuwa, S. Dey, A. K. Singh, and K. McDonald-Maier. "Teem: Online thermal-and energy-efficiency management on cpu-gpu mpsocs." In: *DATE*. IEEE. 2019, pp. 438–443.

[77] X. Jiang, P. Dutta, D. Culler, and I. Stoica. "Micro power meter for energy monitoring of wireless sensor networks at scale." In: *IPSN*. IEEE. 2007, pp. 186–195.

[78] D. Kang, J. Oh, J. Choi, Y. Yi, and S. Ha. "Scheduling of deep learning applications onto heterogeneous processors in an embedded device." In: *IEEE Access* 8 (2020), pp. 43980–43991.

[79] O. Khan and S. Kundu. "A self-adaptive scheduler for asymmetric multi-cores." In: *GLSVLSI* (2010).

[80] T. N. Kipf and M. Welling. "Semi-supervised classification with graph convolutional networks." In: *arXiv preprint arXiv:1609.02907* (2016).

[81] F. Kluge, S. Uhrig, J. Mische, B. Satzger, and T. Ungerer. "Optimisation of energy consumption of soft real-time applications by workload prediction." In: *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*. IEEE. 2010, pp. 63–72.

[82] W. Kool, H. van Hoof, and M. Welling. "Attention, Learn to Solve Routing Problems!" In: *7th ICLR*. 2019.

[83] D. E. Lackey, P. S. Zuchowski, T. R. Bednar, D. W. Stout, S. W. Gould, and J. M. Cohn. "Managing power and performance for system-on-chip designs using voltage islands." In: *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*. 2002, pp. 195–202.

[84] S. Latif, M. Driss, W. Boulila, Z. E. Huma, S. S. Jamal, Z. Idrees, and J. Ahmad. "Deep learning for the industrial internet of things (iiot): A comprehensive survey of techniques, implementation frameworks, potential applications, and future directions." In: *Sensors* 21.22 (2021), p. 7518.

[85] Lattepanda. *Lattepanda 3 Delta - A pocket sized SBC with Windows and Arduino*. https://www.lattepanda.com/lattepanda-3-delta. Accessed: 09-09-2022.

[86] E. Le Sueur and G. Heiser. "Dynamic voltage and frequency scaling: The laws of diminishing returns." In: *Proceedings of the 2010 international conference on Power aware computing and systems*. 2010, pp. 1–8.

[87] Z. Li, Y. Zhang, A. Ding, H. Zhou, and C. Liu. "Efficient algorithms for task mapping on heterogeneous CPU/GPU platforms for fast completion time." In: *J. of Systems Architecture* 114 (2021), p. 101936.

[88] F. Liang, W. Yu, X. Liu, D. Griffith, and N. Golmie. "Toward edge-based deep learning in industrial Internet of Things." In: *IoT-J* 7.5 (2020), pp. 4329–4341.

[89] D. Liu, J. Spasic, G. Chen, and T. Stefanov. "Energy-efficient mapping of real-time streaming applications on cluster heterogeneous mpsocs." In: *ESTIMedia*. IEEE. 2015, pp. 1–10.

[90] C. Maia, L. Nogueira, L. M. Pinho, and D. G. Pérez. "A closer look into the AER model." In: *ETFA*. 2016, pp. 1–8.

[91] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. "Resource management with deep reinforcement learning." In: *Proceedings of the 15th ACM workshop on hot topics in networks*. 2016, pp. 50–56.

[92] N. Mathur, G. Paul, J. Irvine, M. Abuhelala, A. Buis, and I. Glesk. "A practical design and implementation of a low cost platform for remote monitoring of lower limb health of amputees in the developing world." In: *IEEE Access* 4 (2016), pp. 7440–7451.

[93] Q. McNemar. "Note on the sampling error of the difference between correlated proportions or percentages." In: *Psychometrika* 12.2 (1947), pp. 153–157.

[94] S. Minakova, E. Tang, and T. Stefanov. "Combining task-and data-level parallelism for high-throughput CNN inference on embedded CPUs-GPUs MPSoCs." In: *International Conference on Embedded Computer Systems*. Springer. 2020, pp. 18–35.

[95] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. "Playing Atari with Deep Reinforcement Learning." In: *arXiv:1312.5602* (2013).

[96] NCSS, LLC. *NCSS 2022 Statistical Software*. Kaysville, Utah, USA, ncss.com/software/ncss. 2022.

[97] Z. Nakutis. "Embedded systems power consumption measurement methods overview." In: *MATAVIMAI* 2.44 (2009), pp. 29–35.

[98] V. A. Nguyen, D. Hardy, and I. Puaut. "Cache-conscious offline real-time task scheduling for multi-core processors." In: *ECRTS*. 2017.

[99] S. R. Nichols. "The evolution of elevators: physical-human interface, digital interaction, and megatall buildings." In: *Frontiers of Engineering: Reports on Leading-Edge Engineering from the 2017 Symposium*. National Academies Press. 2018.

[100] *Nvidia Jetson Orin*. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/. Accessed: 2023-01-15.

[101] *Nvidia Jetson*. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/. Accessed: 2023-01-21.

[102] U. Odyurt, **J. Roeder**, A. D. Pimentel, I. G. Alonso, and C. de Laat. "Power passports for fault tolerance: Anomaly detection in industrial CPS using electrical EFB." In: *2021 4th IEEE International Conference on Industrial Cyber-Physical Systems (ICPS)*. IEEE. 2021, pp. 152–157.

[103] A. V. Oppenheim, J. R. Buck, and R. W. Schafer. *Discrete-time signal processing. Vol. 2*. Upper Saddle River, NJ: Prentice Hall, 2001.

[104] N. Otterness, M. Yang, S. Rust, E. Park, J. H. Anderson, F. D. Smith, A. Berg, and S. Wang. "An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads." In: *RTAS*. 2017, pp. 353–364.

[105] J. Pallister, S.J. Hollis, and J. Bennett. "Identifying compiler options to minimize energy consumption for embedded platforms." In: *Comput. J.* 58.1 (2015). ISSN: 14602067.

[106] S. Park, J. Park, D. Shin, Y. Wang, Q. Xie, M. Pedram, and N. Chang. "Accurate modeling of the delay and energy overhead of dynamic voltage and frequency scaling in modern microprocessors." In: *Trans. Comput.-Aided Des. Integr. Circuits Syst* 32.5 (2013), pp. 695–708.

[107] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. "A predictable execution model for COTS-based embedded systems." In: *RTAS*. 2011, pp. 269–279.

[108] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin. "Power-performance modeling on asymmetric multi-cores." In: *CASES*. IEEE. 2013, pp. 1–10.

[109] B. Pu, K. Li, S. Li, and N. Zhu. "Automatic fetal ultrasound standard plane recognition based on deep learning and IIoT." In: *IEEE Trans. Industr. Inform.* 17.11 (2021), pp. 7771–7780.

[110] J. P. Queralta, J. Raitoharju, T. N. Gia, N. Passalis, and T. Westerlund. "Autosos: Towards multi-uav systems supporting maritime search and rescue with lightweight ai and edge computing." In: *arXiv preprint arXiv:2005.03409* (2020).

[111] E. Quertemont. "How to statistically show the absence of an effect." In: *Psychologica Belgica* 51.2 (2011), pp. 109–127.

[112] *Raspberry Pi*. https://www.raspberrypi.org/. Accessed: 2020-08-01.

[113] H. Rihani, M. Moy, C. Maiza, R.I. Davis, and S. Altmeyer. "Response time analysis of synchronous data flow programs on a many-core processor." In: *24th International Conference on Real-Time Networks and Systems*. ACM. 2016.

[114] S. Rostedt and D. V. Hart. "Internals of the RT Patch." In: *Proceedings of the Linux symposium*. Vol. 2. Citeseer. 2007, pp. 161–172.

[115] B. Rouxel and I. Puaut. "STR2RTS: Refactored StreamIT benchmarks into statically analyzable parallel benchmarks for WCET estimation & real-time scheduling." In: *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*. Vol. 57. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017.

[116] B. Rouxel, S. Skalistis, S. Derrien, and I. Puaut. "Hiding communication delays in contention-free execution for SPM-based multi-core architectures." In: *31st ECRTS19*. 2019.

[117] M. Sadegh Norouzzadeh, A. Nguyen, M. Kosmala, A. Swanson, M. S. Palmer, C. Packer, and J. Clune. "Automatically identifying, counting, and describing wild animals in camera-trap images with deep learning." In: *PNAS* 115.25 (2018), E5716–E5725. URL: https://www.pnas.org/doi/abs/10.1073/pnas.1719367115.

[118] R. Sakellariou and H. Zhao. "A hybrid heuristic for DAG scheduling on heterogeneous systems." In: *IPDPS*. 2004, p. 111.

[119] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. "Prioritized Experience Replay." In: *arXiv:1511.05952* (2015).

[120] S. Schneider, G. W. Taylor, S. Linquist, and S. C. Kremer. "Past, present and future approaches using computer vision for animal re-identification from camera trap data." In: *MEE* 10.4 (2019), pp. 461–470.

[121] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott. "Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling." In: *Proceedings Eighth International Symposium on High Performance Computer Architecture*. IEEE. 2002, pp. 29–40.

[122] S. S. Shapiro and M. B. Wilk. "An analysis of variance test for normality (complete samples)." In: *Biometrika* 52.3/4 (1965), pp. 591–611.

[123] A. Z. Sheikh and M. A. Pasha. "Energy-efficient multicore scheduling for hard real-time systems: A survey." In: *TECS* 17.6 (2018), pp. 1–26.

[124] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. "Mastering the game of Go without human knowledge." In: *Nature* 550 (2017).

[125] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel. "Mapping on multi/many-core systems: Survey of current and emerging trends." In: *2013 50th ACM/EDAC/IEEE DAC*. IEEE. 2013, pp. 1–10.

[126] S. Stepanovic, G. Georgakarakos, S/ Holmbacka, and J. Lilius. "Quantifying the Interaction Between Structural Properties of Software and Hardware in the ARM Big.LITTLE Architecture." In: *IEEE PDP*. 2018, pp. 138–144.

[127] T. Strang and C. Bauer. "Context-aware elevator scheduling." In: *21st International Conference on Advanced Information Networking and Applications Workshops (AINAW'07)*. Vol. 2. IEEE. 2007, pp. 276–281.

[128] R. Sykora. *Moving up from hardwired relay logic*. https://www.controleng.com/articles/moving-up-from-hardwired-relay-logic/. Accessed: 06-09-2022.

[129] M. Thammawichai and E.C. Kerrigan. "Energy-efficient real-time scheduling for two-type heterogeneous multiprocessors." In: *Real-Time Syst.* 54.1 (2018). ISSN: 15731383.

[130] N. Tijtgat, W. Van Ranst, T. Goedeme, B. Volckaert, and F. De Turck. "Embedded real-time object detection for a UAV warning system." In: *ICCVW*. 2017, pp. 2110–2118.

[131] H. Topcuoglu, S. Hariri, and M. Wu. "Performance-effective and low-complexity task scheduling for heterogeneous computing." In: *IEEE TPDS* 13.3 (2002), pp. 260–274.

[132] R. S. Turgel. *Sampling Techniques for Electric Power Measurement*. Vol. 870. US Department of Commerce, National Bureau of Standards, 1975.

[133] U. Ullah Tariq, H. Ali, L. Liu, J. Panneerselvam, and X. Zhai. "Energy-efficient Static Task Scheduling on VFI based NoC-HMPSoCs for Intelligent Edge Devices in Cyber-Physical Systems." In: *TIST* 1.1 (2019). ISSN: 2157-6912.

[134] L. Van, Y. Lin, T. Wu, and T. Chao. "Green elevator scheduling based on IoT communications." In: *IEEE Access* 8 (2020), pp. 38404–38415.

[135] E. Vasilakis, I. Sourdis, V. Papaefstathiou, A. Psathakis, and M.G.H. Katevenis. "Modeling energy-performance tradeoffs in ARM big. LITTLE architectures." In: *PATMOS*. IEEE. 2017, pp. 1–8.

[136] E. Vasilakis, I. Sourdis, V. Papaefstathiou, A. Psathakis, and M.G.H. Katevenis. "Modeling energy-performance tradeoffs in ARM big.LITTLE architectures." In: *27th PATMOS* (2017).

[137] F. Wilcoxon. "Individual comparisons by ranking methods." In: *Breakthroughs in statistics*. Springer, 1992, pp. 196–202.

[138] M. Willi, R. T. Pitman, A. W. Cardoso, C. Locke, A. Swanson, A. Boyer, M. Veldthuis, and L. Fortson. "Identifying animal species in camera trap images using deep learning and citizen science." In: *MEE* 10.1 (2019), pp. 80–91.

[139] R. J. Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning." In: *Machine learning* 8.3-4 (1992), pp. 229–256.

[140] Q. Wu, Z. Wu, Y. Zhuang, and Y. Cheng. "Adaptive DAG Tasks Scheduling with Deep Reinforcement Learning." In: *19th ICA3PP*. Springer. 2018.

[141] H.E. Zahaf, A.E.H. Benyamina, R. Olejnik, and G. Lipari. "Energy-efficient scheduling for moldable real-time tasks on heterogeneous computing platforms." In: *J. of Systems Architecture* 74 (2017). ISSN: 13837621.

[142] G. Zeng, T. Yokoyama, H. Tomiyama, and H. Takada. "Practical energy-aware scheduling for real-time multiprocessor systems." In: *RTCSA*. IEEE. 2009, pp. 383–392.

[143] H. Zhao and R. Sakellariou. "An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm." In: *Euro-Par*. 2003, pp. 189–194.

[144] **J. Roeder**, S. Altmeyer, and C. Grelck. "Can we trust our energy measurements? A study on the Odroid-XU4." In: *15th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT at ECRTS 2022)*. 2022, p. 33.

[145] **J. Roeder**, S. Altmeyer, and C. Grelck. "Energy Measurements of 9 Rodinia Benchmarks executed on the Odroid-XU4." In: *Dataset published on UvA figshare* (June 2022). DOI: 10.21942/uva.19665564.v1. URL: https://uvaauas.figshare.com/articles/dataset/Energy_Measurements_of_9_Rodinia_Benchmarks_executed_on_the_Odroid-XU4_/19665564.

[146] **J. Roeder**, A.D. Pimentel, and C. Grelck. "GCN-based reinforcement learning approach for scheduling DAG applications." In: *19th Artificial Intelligence Applications and Innovations (AIAI 2023)*. Springer. 2023.

[147] **J. Roeder**, B. Rouxel, S. Altmeyer, and C. Grelck. "Interdependent Multi-version Scheduling in Heterogeneous Energy-aware Embedded Systems." In: *13th Junior Researcher Workshop on Real-Time Computing (JRWRTC at RTNS 2019)*. 2019, p. 1.

[148] **J. Roeder**, B. Rouxel, S. Altmeyer, and C. Grelck. "Energy-aware scheduling of multi-version tasks on heterogeneous real-time systems." In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing (SAC 2021)*. 2021, pp. 501–510.

[149] **J. Roeder**, B. Rouxel, S. Altmeyer, and C. Grelck. "Scheduling multi-version tasks on heterogeneous IoT systems using energy-aware ranking." In: *Under review*. 2022.

[150] **J. Roeder**, B. Rouxel, and C. Grelck. "Q-learning for Statically Scheduling DAGs." In: *2020 IEEE International Conference on Big Data (Big Data)*. IEEE Computer Society. 2020, pp. 5813–5815.

[151] **J. Roeder**, B. Rouxel, and C. Grelck. "Scheduling DAGs of Multi-version Multi-phase Tasks on Heterogeneous Real-time Systems." In: *14th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC 2021), Singapore. IEEE*. 2021.

ACRONYMS

MPSoC	Multiprocessor System-on-Chip
DAG	Directed Acyclic Graphs
eFLS	energy Forward List Scheduling
HER	Heterogeneous Energy-aware Ranking
GA	Genetic Algorithm
FLS	Forward List Scheduling
hFLS	heterogeneous Forward List Scheduling
DVFS	Dynamic Voltage and Frequency Scaling
DL	Deep Learning
HEFT	Heterogeneous Earliest Finish Time
COTS	Commercially-Off-The-Shelf
FPGA	Field-Programmable Gate Array
IoT	Internet of things
IIoT	Industrial Internet of things
CPS	cyber-physical systems
RTS	real-time systems
ILP	Integer Linear Programming
RL	Reinforcement Learning
SoC	System-on-Chip
WCET	Worst Case Execution Time
EA	Evolutionary Algorithm
CU	compute unit
UAV	Unmanned Aerial Vehicles
GAG	Gerichte Acyclische Graaf

PUBLICATIONS

[1] U. Odyurt, **J. Roeder**, A. D. Pimentel, I. G. Alonso, and C. de Laat. "Power passports for fault tolerance: Anomaly detection in industrial CPS using electrical EFB." In: *2021 4th IEEE International Conference on Industrial Cyber-Physical Systems (ICPS)*. IEEE. 2021, pp. 152–157.

[2] B. Rouxel, **J. Roeder**, A. Sebastian, and G. Clemens. "A time, energy and security coordination approach." In: *10th International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS 2019)*. 2019.

[3] A. Seewald, U. P. Schultz, **J. Roeder**, B. Rouxel, and C. Grelck. "Component-based computation-energy modeling for embedded systems." In: *SPLASH*. 2019, pp. 5–6.

[4] **J. Roeder**, S. Altmeyer, and C. Grelck. "Can we trust our energy measurements? A study on the Odroid-XU4." In: *15th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT at ECRTS 2022)*. 2022, p. 33.

[5] **J. Roeder**, S. Altmeyer, and C. Grelck. "Energy Measurements of 9 Rodinia Benchmarks executed on the Odroid-XU4." In: *Dataset published on UvA figshare* (June 2022). DOI: 10.21942/uva.19665564.v1. URL: https://uvaauas.figshare.com/articles/dataset/Energy_Measurements_of_9_Rodinia_Benchmarks_executed_on_the_Odroid-XU4_/19665564.

[6] **J. Roeder**, A.D. Pimentel, and C. Grelck. "GCN-based reinforcement learning approach for scheduling DAG applications." In: *19th Artificial Intelligence Applications and Innovations (AIAI 2023)*. Springer. 2023.

[7] **J. Roeder**, B. Rouxel, S. Altmeyer, and C. Grelck. "Interdependent Multi-version Scheduling in Heterogeneous Energy-aware Embedded Systems." In: *13th Junior Researcher Workshop on Real-Time Computing (JRWRTC at RTNS 2019)*. 2019, p. 1.

[8] **J. Roeder**, B. Rouxel, S. Altmeyer, and C. Grelck. "Towards Energy-, Time-and Security-Aware Multi-core Coordination." In: *International Conference on Coordination Languages and Models*. Springer. 2020, pp. 57–74.

[9] **J. Roeder**, B. Rouxel, S. Altmeyer, and C. Grelck. "Energy-aware scheduling of multi-version tasks on heterogeneous real-time systems." In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing (SAC 2021)*. 2021, pp. 501–510.

[10] **J. Roeder**, B. Rouxel, S. Altmeyer, and C. Grelck. "Scheduling multi-version tasks on heterogeneous IoT systems using energy-aware ranking." In: *Under review*. 2022.

[11] **J. Roeder**, B. Rouxel, and C. Grelck. "Q-learning for Statically Scheduling DAGs." In: *2020 IEEE International Conference on Big Data (Big Data)*. IEEE Computer Society. 2020, pp. 5813–5815.

[12] **J. Roeder**, B. Rouxel, and C. Grelck. "Scheduling DAGs of Multi-version Multi-phase Tasks on Heterogeneous Real-time Systems." In: *14th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC 2021), Singapore. IEEE*. 2021.

TEACHING AND SUPERVISION

TEACHING EXPERIENCE

- MSc. Computer Science, *Programming Multi-core and Many-core Systems*, 2021
- ASCI, *A Programmer's Guide for Modern High-Performance Computing*, 2020
- MSc. Computer Science, *Programming Multi-core and Many-core Systems*, 2020
- MSc. Computer Science, *Programming Multi-core and Many-core Systems*, 2019

THESIS SUPERVISION

- Henok Ghebrenigus, *Evaluating evolutionary algorithms for static scheduling with energy consumption reduction on heterogeneous embedded systems*, BSc. Computer Science 2022
- Stan Bergevoet, *Performance and Power Consumption Accuracy Evaluation of the Sniper Multi-Core Simulator*, BSc. Computer Science 2019
- Dennis Wind, *Run time and energy consumption trade-offs for ARM big.LITTLE*, BSc. Computer Science 2019

SOURCE CODE AND DATASET

- The source code for Chapter 3 can be found in the following bitbucket repository: https://bitbucket.org/uva-sne/energymeasurementanalysis/src.

  The source code for Chapters 4 and 5 can be found in the following bitbucket repositoy: https://bitbucket.org/uva-sne/coordinationcompiler/src.

- The source code for Chapter 6 can be found in the following bitbucket repository: https://bitbucket.org/jroeder/simple_rl_scheduling/src/.

- The data used for Chapters 3 and 4 can be found at the following URL: https://uvaauas.figshare.com/articles/dataset/Energy_Measurements_of_9_Rodinia_Benchmarks_executed_on_the_Odroid-XU4_/19665564/1.

- The data used for Chapters 5 and 6 is generated using the TGFF tool [41]. The TGFF settings and generated datasets can be found https://bitbucket.org/jroeder/tgff_multi_phase/ and https://bitbucket.org/jroeder/gnn_tgff_data/ respectively.

SUMMARY

Heterogeneous high-performance embedded systems (i.e. single board computers employing an MPSoC) have become ubiquitous in recent years. Applications range from identifying potential poachers with camera traps in national parks to literal "on-the-fly" image analysis on unmanned aerial vehicles. Often these applications must be executed locally because the environmental situation imposes some restrictions. For example, the image analysis must be done on the fly as a quick reaction is required. The camera traps identifying poachers communicate via LoRaWAN, thus, have limited bandwidth, but poachers must be identified quickly. Additionally, both applications are battery-powered. Therefore, energy consumption is crucial.

In this thesis, we focus on improving the hardware utilisation of the MPSoC found on high-performance embedded systems with respect to energy consumption and time. High-performance embedded systems, such as the Odroid-XU4, are single-board computers with multiple CPU cores, possibly with different architectures (e.g., ARM big.LITTLE) and an onboard accelerator such as a GPU. Throughout this work, we propose and investigate different static scheduling methods (i.e. determining offline where and when a task should be executed).

To improve energy consumption, we first must measure energy consumption accurately. Thus, first, we explain the measurement setup used throughout this dissertation. Next, we detail a set of experiments that determines how important the sampling rate is for accurate energy measurement of high-performance embedded systems. This includes a visual representation of the error introduced if one measures at a low sampling rate. We further investigate the minimum required sampling rate that can be considered equivalent to the original power trace (equivalence testing).

Next, we propose a new energy model, system model and energy-aware scheduler. Combining all three results in a new scheduling method for MPSoC that supports Dynamic Voltage and Frequency Scaling, multiple-version tasks, heterogeneous CPUs and GPU-like accelerators. Multiple versions allow the schedulers to target different binary-incompatible compute units for a task. We can further exploit a multi-version approach and support versions resulting from, e.g., different compiler flags or functionally-equivalent

algorithms etc. We show that our approach outperforms two state-of-the-art schedulers, Heterogeneous Earliest Finish Time (HEFT) and ARSH-FATI.

All our methods target applications that can be represented as Directed Acyclic Graphs (DAG). DAG applications provide a partial order of tasks. However, many schedulers (e.g., HEFT, ARSH-FATI, the Forward List Scheduling family of schedulers) require a total order of tasks. We show that the ranking used can have a significant impact on the performance of the final application. Additionally, we introduce a new ranking algorithm and compare its performance against a set of base ranking algorithms.

Tasks that run on a GPU need to be controlled by the CPU. Hence, a task that targets the GPU first runs on the CPU. The CPU part of the task prepares the data for the GPU and then launches the GPU portion of the task. Existing scheduling methods reserve both the CPU and GPU for the entirety of the task. However, this is inefficient as valuable CPU, and GPU compute time is lost unnecessarily. Therefore, we propose a scheduling method that divides tasks into multiple phases. This allows for a more fine-grained separation of workloads and targeting, e.g., the CPU and GPU. We show that our scheduler leads to a higher schedulability rate than HEFT.

Lastly, as many schedulers heavily rely on the ranking of tasks (i.e., total order), we explore a ranking independent Reinforcement Learning (RL) based scheduler. We perform experiments with the RL scheduler for DAG applications and compare it against a traditional greedy heuristic. The RL scheduler generates schedules with a makespan similar to a greedy heuristic and including graph convolutional neural networks improves the resulting schedules significantly.

To conclude, the work in this thesis presents different strategies and methodologies focused around scheduling dependent tasks to high-performance embedded systems. The aim is to improve the utilisation of applications deployed to high-performance embedded systems with respect to different aspects such as energy consumption and run-time. In summary, the main topics explored in this thesis include accurate energy measurements, improving energy consumption through static scheduling, increasing schedulability by better utilising the available hardware and exploring ranking independent scheduling strategies. Throughout this thesis, we propose techniques to use less powerful hardware for solving a given problem, hence, saving money and resources. Alternatively, the same techniques can be used to solve more advanced problems with the same hardware.

SAMENVATTING

Heterogene high-performance geëmbedde systemen (bijvoorbeeld single-board computers met een MPSoC) zijn tegenwoordig niet meer weg te denken. Applicaties omvatten onder andere systemen die in natuurreservaten met cameravallen stropers identificeren en systemen die letterlijk tijdens het vliegen fotoanalyse uitvoeren vanuit onbemande luchtvaartuigen. Dit soort applicaties zijn doorsnee aan omgevingsrestricties onderhevig waardoor ze enkel lokaal uitgevoerd kunnen worden. Zo moet de fotoanalyse bijvoorbeeld dusdanig snel uitgevoerd worden om nog tijdens de vlucht op een stroper te reageren. De cameravallen die de stropers identificeren communiceren met LoRaWAN, een protocol met een lage doorvoersnelheid, wat een extra obstakel vormt in het snel kunnen identificeren van de stropers. En daar komt nog bovenop dat beide applicaties van energie voorzien wordt door middel van een accu, waardoor het energieverbruik in de perken houden ook nog een essentiële eis is.

In dit proefschrift richten wij ons op het verbeteren van de hardware-utilisatie van een MPSoC, zoals gevonden in een high-performance geëmbed systeem met betrekking op energieverbruik en looptijd. High-performance geëmbedde systemen, zoals de Odroid-XU4, zijn single-board computers met meerdere processorkernen van mogelijk verschillende architecturen (bijvoorbeeld ARM big.LITTLE), en een geïntegreerde accelerator zoals een GPU. In dit werk introduceren en evalueren wij verschillende statische schedulingsmethodes (statisch wil zeggen het van te voren vaststellen van waar een wanneer een taak uitgevoerd moet worden).

Om het energieverbruik te verbeteren moet het energieverbruik eerst exact gemeten worden. Daarom lichten wij eerst de energiemeetopstelling toe die wij in de rest van dit proefschrift gebruiken. Daarna beschrijven wij een aantal experimenten die vaststellen wat het belang van de meetfrequentie is voor het accuraat vaststellen van het energieverbruik van high-performance geëmbedde systemen. Dit doen we onder andere aan de hand van een visuele representatie van de ontstane fout voortgebracht uit het gebruik van een te lage meetfrequentie. Wij zetten het onderzoek voort door vast te stellen wat de laagste vereiste meetfrequentie is die als equivalent gezien kan worden aan het originele powertrace (door middel van equivalentietesten).

Daarnaast stellen we een nieuw energiemodel, een systeemmodel en een energiebewuste scheduler voor. Het combineren van deze drie resultaten

leidt tot een nieuwe schedulingsmethode voor MPSoC met ondersteuning voor Dynamic Voltage and Frequency Scaling, taken met meerdere versies, heterogene processors, en GPU-achtige acceleratoren. Ondersteuning voor taken met meer dan één versie stelt de schedulers in staat om verschillende binary-incompatibele verwerkingseenheden (zoals processors en GPU's) te ondersteunen. We kunnen deze techniek verder exploiteren door ondersteuning te bieden voor bijvoorbeeld versies met verschillende compileropties, of functioneel-equivalente versies gebruikmakend van verschillende algoritmes. We laten zien dat onze aanpak beter presteert dan twee eerder gepubliceerde schedulers, namelijk Heterogeneous Earliest Finish Time (HEFT) en ARSH-FATI.

Onze technieken richten zich op applicaties die gestructureerd kunnen worden als Gerichte Acyclische Graaf (GAG). GAG-applicaties kennen een partiële orde toe aan de taken. Echter, veel schedulers (zoals HEFT, ARSH-FATI, en Forward List Scheduling-gebaseerde schedulers) hebben een totale orde nodig. Wij laten zien dat de gebruikte orde een significant impact heeft op de presentaties van de uiteindelijke applicatie. Daarnaast introduceren wij een nieuw ordeningsalgoritme en vergelijken de prestaties met die van een aantal standaard ordeningsalgoritmes.

Taken die op de GPU draaien moeten beheerd worden door de processor. Hierdoor draait een taak voor de GPU eerst op de processor. Het processordeel van de taak bereidt de data voor, waarna het GPU-deel uitgevoerd wordt. Bestaande schedulingsmethodes reserveren zowel de GPU als de processor voor de volledige duur van de taak, maar dit is inefficiënt omdat waardevolle processortijd onnodig verloren gaat. Daarom stellen wij een schedulingsmethode voor die de taak in meerdere fasen splitst. Deze fasen stellen ons in staat om efficiënter de hardware te reserveren voor taken die een GPU en processor gebruiken. Wij laten zien dat onze scheduler in significant meer gevallen een geldige schedule kan produceren dan HEFT.

Als laatste, aangezien veel schedulers erg afhankelijk zijn van de orde van de taken (namelijk de totale orde) verkennen wij een orde-onafhankelijke Reinforcement Learning-gebaseerde (RL) scheduler. We experimenteren met de RL-scheduler voor GAG-applicaties, en vergelijken deze met een doorsnee gulzige heuristiek-gebaseerde scheduler. De RL-scheduler produceert schedules met een totale looptijd vergelijkbaar met die geproduceerd d.m.v. gulzige heuristiek. Het toevoegen van neurale netwerken op basis van graafconvolutie verbetert de schedules significant.

In conclusie, het werk in dit proefschrift presenteert verschillende strategieën en methodes gericht op het schedulen van taken met onderlinge afhan-

kelijkheden voor high-performance geëmbedde systemen. Het doel is om de hardwarebezetting te verbeteren voor applicaties op high-performance geëmbedde systemen met betrekking op verschillende aspecten zoals energieverbruik en looptijd. De hoofdonderwerpen verkend in dit proefschrift omvatten het nauwkeurig meten van energieverbruik, het verbeteren van het energieverbruik door middel van statische schedulingsmethodes, en het vermeerderen van de gevallen waarin een geldige schedule bepaald kan worden. In dit proefschrift stellen wij technieken voor die minder krachtige hardware nodig hebben om een gegeven probleem op te lossen, ter besparing van geld en hardware. Op dezelfde manier kunnen onze methodes ook worden gebruikt om geavanceerdere problemen op te lossen met dezelfde hardware.