Energy efficient scheduling of parallel real-time tasks on heterogeneous multicore systems

Houssam Eddine Zahaf

UNIVERSITY OF LILLE
UNIVERSITY OF ORAN 1

DOCTORAL THESIS

Energy efficient scheduling of parallel real-time tasks on heterogeneous multicore systems

Author:
Houssam-Eddine ZAHAF

Supervisor:
Dr. Richard OLEJNIK
Dr Abou-ElHassen BENYAMINA

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Computer Science

November 2,2016

Joöl GOOSSENS	Full professor, Univ-Libre Brussels, Belgium	Referee
Mohamed BENYETTOU	Full professor, USTO, Algeria	Referee
Samira CHOURAQUI	Full professor, USTO, Algeria	Examiner
Yahia LEBBAH	Full Professor, LITIO, Univ-Oran1, Algeria	Examiner
Sophie Quinton	Researcher, Inria, Grenoble, France,	Examiner
Giuseppe LIPARI	Full professor, CRIStAL, Univ-Lille1, France	Invited
Richard OLEJNIK	Senior Researcher, CNRS, CRIStAL, Univ-Lille1, France	Supervisor
A.Hassen BENYAMINA	Associate Professor, LAPECI, Univ-Oran1, Algeria	Supervisor

"Life is short and you can not realize
If you start a job you have to finalize
worthy, life is not what someone did or does
Sometimes your life is too many lies
and too many shames you have exercise
So never be late to tell your apologize
Cause life is short and you can not realize

Life is something and you should matter
So start by making it little bit better
Every thing is simple, An alphabet starts with one letter
And even one hundred miles starts with one meter
It is not hard, Try to speak, or send a letter
Cause, life is short and it does matter

A lot of people where there for me from the day I took my first breath, and they are still in my life, some has been gone, but they are still as memories, others enter and they are making memories,
Thank You, "

Houssam Eddine ZAHAF

Acknowledgements

The research lead in this thesis is the property of both University of Lille1, and University of Oran 1. The works that will be presented has been done in two research teams: Optimization dans les Reseaux de Transport Et Systemes Embarques (ORTESE) team of the laboratory "Laboratoire d'Architetures Paralleles, Embarquees et du Calcul Intensif" LAPECI of university of Oran 1 and, emeraude team of "Centre de Recherche en Informatique, Signal et Automatique de Lille" CRIStAL laboratory of university of Lille. The research were held at IRCICA, "Institut de Recherche en Composants logiciels et materiels pour l'Information et la Communication Avancee" and LAPECI laboratory.

This research has been funded in part by IRCICA and PHC Curien under the direction of Abou El Hassen BENYAMINA and Richard OLEJNIK. Giuseppe LIPARI has participated in directing the thesis and has importatnt impact on the choices taken during the last two years. I am very glad and grateful to my directors Hassan and Richard with a very special thank and admiration to Giuseppe LIPARI for his scientific and human support.

I would thank also the member of my jury. Firstly, Jöel Goossens and Benyattou Mohamed for accepting and putting time even in their very full agenda, to read and evaluate my work. I thank also Sophie Quinton, Samira Chouraqui and Lebbah Yahia to be members of my jury and accept to examine the research presented in this disseration.

I will always remember the great times that I have passed in Emeraude and ORTESE teams. I will always keep in my memories Philippe Devienne, Pierre Boulet, Clement Balabriga, Pierre Falez, Yassine SidLakhdar, Loukil Lakhdar, Abbassia Deba, Aroui Abdelkader, $\cdots$. A special thank to Antoine Bertout for discussions that we had about everything and nothing, and the great moment that we shared in playing football. A special thank to all IRCICA and IRI colleagues, Anne So, Ahmed, Xavier lepallec and all the people that I could not cite all names.

A special thought to my family, My Father Charef and my both mothers, the one who gave me life and to the one that made me the man I am today. A special thank to my brothers: Farida, Ismail, Salah, Oum Elkhier, Nassima, Hanane, Wafaa, Rajaa, my nephews, and to my friends, especially mohammed maloufi, Azzouz Joseph and Amrane Kichou. A very special thought to Sawsen, the woman that made my life different. Thank you for being there, you are a treasure for me.

Abstract

by Houssam-Eddine ZAHAF

Cyber physical systems (CPS) and Internet of Objects (IoT) are generating an unprecedented volume and variety of data that needs to be collected and stored on the cloud before being processed. By the time the data makes its way to the cloud for analysis, the opportunity to trigger a reply might be late.

One approach to solve this problem is to analyze the most time-sensitive data at the network edge, close to where it is generated. Thus, only the pre-processed results are sent to the cloud. This computation model is know as *Fog Computing* or *Edge computing*. Critical CPS applications using the fog computing model may have real-time constraints because results must be delivered in a pre-determined time window. Furthermore, in many relevant applications of CPS, the processing can be parallelized by applying the same processing on different sub-sets of data at the same time by the mean parallel programming techniques. This allow to achieve a shorter response time, and then, a larger slack time, which can be used to reduce energy consumption.

In this thesis we focus on the problem of scheduling a set of parallel tasks on multicore processors, with the goal of reducing the energy consumption while all deadlines are met. We propose several realistic task models on architectures with identical and heterogeneous cores, and we develop algorithms for allocating threads to processors, select the core frequencies, and perform schedulability analysis. The proposed task models can be realized by using OpenMP-like APIs.

Contents

Contents

List of Figures

List of Tables

List of Symbols

$\%$	The rest of the ecludienne division
$\mathsf{random}(a, b)$	generates a random number between a and b
f_j	The operating frequency of core j
f	An arbitrary operating frequency
s_j	The speed of core j
τ_i	Task i
$\mathcal{J}_{i,a}$	The a^{th} job of task τ_i
$\mathcal{A}_{i,a}$	The arrival time of a^{th} instance of task τ_i
D_i	The relative deadline of task τ_i
O_i	The offset of task τ_i
T_i	The period of task τ_i
$\mathcal{T}$	A task set
$\mathcal{T}_j$	The task set allocated on core j
act	The active state of an arbitrary core
$\mathcal{A}$	A multicore architecture
Th	An arbitrary thread
aTH	An arbitrary allocated thread
C_i	Execution time of the single thread version of task τ_i
$\mathcal{C}$	The execution time of an arbitrary thread
$\mathsf{C}_{i,j}$	The execution time of the thread of task τ_i allocated to core j in FTC model
$\mathsf{Th}_{i,j}$	The thread of task τ_i that is allocated to core j in the FTC task model
$\mathsf{tdbf}(\tau_i, t)$	The demand bound function of task τ_i for an interval of time of length t
$\mathsf{dbf}(\mathcal{T}_j, t)$	The demand bound function of task set $\mathcal{T}_j$ for an interval of time of length t
$\mathcal{D}_i$	An arbitrary decomposition of task τ_i
dbf	The demand bound function
α	The symbol of excess time in equations
$\mathsf{Th}_{i,j,k}$	Thread k of cut-point $\gamma_{i,j}$ of task τ_i
$\mathsf{aTH}_{i,j,k,z}$	Thread $\mathsf{Th}_{i,j,k}$ allocated on core j
$u^g_{i,k,z}(\mathsf{f}_{op})$	The base line utilization of thread $\mathsf{Th}_{i,k,z}$ when allocated on group g
$\vec{\xi}$	Energy coeficients
$\gamma_{i,k}$	The k^{th} cut-point of task τ_i
$\mathsf{ct}^g_{i,j,k}(\mathsf{f}_{op})$	A part of execution time of thread $\mathsf{Th}_{i,k,z}$ on group g operating at frequency f_{op}
ct	An arbitrary part of the execution time that depends on the frequency
$\mathcal{G}_g$	the group g of cores
$\mathsf{mt}^g_{i,k,z}$	The memory acces time of thread $\mathsf{Th}_{i,k,z}$ on group g
mt	An arbitrary memory acces time
Ω^g	The needed strength of group g
$\mathcal{S}^g$	The current strength of core g
$\Pi_i(t)$	All pathes that could be generated by task τ_i in any interval of time of length t
${}^pC_{\lvert S_i \rvert}$	The combination of p elements of $\lvert S_i \rvert$ elements
pdf	An arbitrary path demand function
$\mathsf{pdf}(i, j)$	The demand function of the j^{th} path of task τ_i

$\pi_k^i(t)$	the k^{th} path of task τ_i of length t
$\mathsf{G}(V, E)$	A di-graph of vertices set V and edges E
E_i	The egdes set of the graph of task τ_i in the di-graph model
$e(s, d)$	The edge starting from s and ending at d
$v_{i,j}$	The vertice j of task graph of τ_i
V_i	The vertice set of the task graph of τ_i
$\mathcal{C}_{i,j,k}^g(\mathsf{f}_{op})$	The worst case execution time of thread $\mathsf{Th}_{i,k,z}$ on group g
$\mathcal{C}_{i,k}^{\mathsf{v}}(\mathsf{f})$	The execution time of vertice $v_{i,j}$ on a core operating at f

Introduction

Context

The internet of things (IoT) is a network of physical devices, vehicles, buildings and other embedded items with electronics, software, sensors, and network connectivity that enables these objects to collect and exchange data. The IoT allows objects to be sensed and controlled remotely across existing network infrastructure, creating opportunities for more direct integration of the physical world into computer-based systems, and resulting in improved efficiency, accuracy and economic benefit. When IoT is augmented with sensors and actuators, the technology becomes an instance of the more general class of cyber-physical systems (CPS) (please refer to Baheti and Gill, 2011).

A CPS is therefore a system composed of physical entities such as mechanisms controlled or monitored by computer-based algorithms. Today, a precursor generation of cyber-physical systems can be found in areas as diverse as aerospace, automotive, chemical processes, civil infrastructure, energy, health-care, manufacturing, transportation, entertainment, and consumer appliances.

CPS is generating an unprecedented volume and variety of data, by the time the data makes its way to the cloud for analysis, the opportunity to trigger a reply might be *late*. The basic idea is to analyze the most *time-sensitive* data at the network edge, close to where it is generated instead of sending vast amounts of data to the cloud. The reply can be triggered *quick* enough to ensure the system constraints. Only, the pre-processing results are sent later to the cloud for historical analysis and longer-term storage. Thus, a CPS should (Cisco, 2015):

- Minimize latency: Milliseconds matter when you are trying to prevent manufacturing line shutdowns or restore electrical service. Analyzing data close to the device that collected the data can make the difference between averting disaster and a cascading system failure.

- Conserve network bandwidth: Offshore oil rigs generate 500 GB of data weekly. Commercial jets generate 10 TB for every 30 minutes of flight. It is not practical to transport vast amounts of data from thousands or hundreds of thousands of edge devices to the cloud. Nor is it necessary, because many critical analyses do not require cloud scale processing and storage.

- Operate reliably: CPS data is increasingly used for decisions affecting citizen safety and critical infrastructure. The integrity and availability of the infrastructure and data cannot be in question.

This computing model is known as *Fog Computing* or *Edge computing* (see Bonomi et al., 2012). Fog computing uses one or a collaborative multitude of near-user edge devices to carry out a substantial amount of storage, communication, and control, configuration, measurement and management.

The embedded platforms used for supporting fog computing are often multicore systems and many of CPS applications can be easily parallelized by distributing data across the parallel computing elements. Reducing power consumption in these systems is a very serious problem especially when processing elements operate on battery power. Even when they are connected to the electric grid, we need to keep the consumption as low as it is possible. Multicore technology can help us in achieving timeliness and low power consumption systems. In fact, even

when the computational load is not very high, multicore processors are more energy efficient than an equivalent single-core platform as reported by Wolf, 2012.

Critical processings done in CPS systems such as electrical grid control must deliver responses in a pre-determined time window. Thus, a large spectrum of CPS applications real-time applications.

Therefore, in this thesis we will be interested in parallelizing real-time application to multiprocessor architectures in the goal of reducing the energy consumption.

Contributions

In this thesis, we will be interested in particular in parallelization techniques, real-time scheduling techniques, energy reduction techniques for a set of real-time tasks expressed with different models on several types of multicore architectures. Mainly our contributions consist in proposing realistic and expressive task models and efficient feasibility tests for these task models on multicore architectures. Based on the proposed tests, we propose methodologies to reduce the energy consumption for identical, uniform and heterogeneous cores. Thus the main contributions of this thesis are:

1. The Free-To-Cut FTC task model,
2. The allocation of FTC tasks to uniform architectures: exact and sufficient feasibility tests,
3. The Cut-point task model (CPM),
4. A methodology to build a realistic timing, power and energy models
5. The allocation of cut-point tasks to heterogeneous multicore architectures: exact and sufficient feasibility tests,
6. Modelling parallel task with di-graphs,
7. The allocation of di-graph parallel tasks to identical cores platform: a sufficient feasibility test.

Organization

This thesis is structured as follows. In the first chapter, we give a quick overview on multicore systems, parallel programming and energy consumption techniques. In this chapter, we illustrate how parallel systems are seen for non-real-time systems. The second chapter will be reserved to real-time systems theory, and practice. This chapter is divided mainly into two parts: in the first, we talk mainly on real-time systems scheduling theory, and in the second part, we will show how a simple real-time task model can be implemented in a real-time operating system. We will talk later in chapter 3 about the context of our work and the motivations that pushed us to be interested in the parallelization of real-time task on multicore architecture, and we will state some work that has been done in this field of research in real-time systems. In the chapter 4, we present a simple and non-realistic parallel real-time task model that we call, Free-To-Cut task model. We propose a corresponding feasibility test, and exact scheduler and a heuristic to allocate such tasks to a uniform multicore architecture. The chapter 5, will be reserved into an extension of the model in chapter 4, to a realistic model. In addition, we will be interested on allocating a set of task of the extended model to heterogeneous multicore platform. In this chapter, we present the results of a large set of experiments that has been conducted on an ARM big.LITTLE processor to build task, architecture and energy models. We propose an other feasibility test for these tasks on heterogeneous architectures. We will show that obtaining an optimal solution of these problem is hard for only medium size problem, and

we will propose heuristics that allows to obtain quasi-optimal solutions in a reasonable time. In the last chapter, we present a very expressive parallel task model that extend the di-graph the one proposed by Stigge et al., 2011. We propose a sufficient feasibility test for the extended model to a set of identical cores.

Part I

Context, Motivations & Related work

Chapter 1

Multiprocessors & Parallel Systems

" On dit que le temps change les choses,
mais en fait le temps ne fait que passer
et nous devons changer les choses nous-młmes. "

Andy Warhol

Contents

1.1 Introduction

Gordon E. Moore, the co-founder of Intel and Fair-child Semiconductor observed, in 1965, that the number of transistors in a dense integrated circuit is doubling every year in the number of components per integrated circuit, and projected this rate of growth would continue for at least another decade. In 1975, looking forward to the next decade, he revised the forecast to doubling every two years. This observation is called the moore's law (please refer to Aspray, 2004, Moore, 2006, Moore, 2005). In the last decade, Intel stated that the pace of advancement has slowed.

Starting from 2004, the market leaders have difficulties of satisfying Moores law greedy demand for computing power using classic single processor architectures because increasing the operating clock speed improvements slowed due to the serious heating problems and considerable power consumption (see Figure 1.1) and also due to the increasing gap between processor and memory speeds. This, in effect, pushes cache sizes to be larger in order to mask the latency of memory. This helps only to the extent that memory bandwidth is not the bottleneck in performance, the increasing difficulty of finding enough parallelism in a single instruction stream to keep a high-performance single-core processor busy.

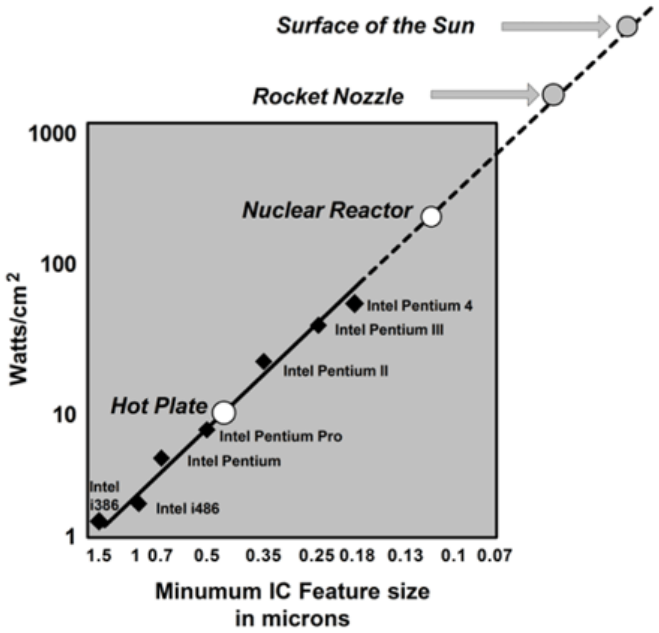


FIGURE 1.1: The power & heat increasing

In order to continue performance improvements, processor manufacturers such as Intel and AMD have turned to multicore designs. the idea is to put several computing elements on the same die and operate these cores on lower frequencies.

A multicore processor is a single computing component with more than one computing element (referred as "cores"). Manufacturers integrate cores onto a single integrated circuit. In contrast to multicore systems, the term multiprocessors refers to multiple physically separate processing-units.

1.1.1 Classification of multicore systems

Multicore processors have been pervading and several architectures of multicore systems had been proposed. For example, cores may or may not share caches, and they may implement message passing (*MPI itemplementations* 2016) or shared-memory inter-core communication (such as OpenMP Architecture Review Board, 2008). At the same time, cores may run same instructions or different ones. Cores may be interconnected by a single bus, ring, two-dimensional

mesh, and crossbar, $\cdots$. According to a criteria (Instruction set, memory, micro-architectures, $\cdots$), multicore (multiprocessor) systems can be classified to distinguish between them.

According to Flynn

Flynn, 1972 classification of architectures is based upon the number of instructions that could be run at the same time (single, multiple), and on the data streams on which instructions are applied (also single, multiple). Hence, 4 classes can be distinguished:

- **Single Instruction, Single Data (SISD)**: Also known as the Von Neumann machine, or sequential computer. In this class no parallelism is allowed, only one instruction is run at the time. A single control unit fetches a single instruction from memory. However, SISD machines can have concurrent processing characteristics. Pipelined processors and superscalar processors are common examples found in most modern SISD computers (Michael, 2003).
- **Single Instruction, Multiple Data (SIMD)**: In such architectures, the same instruction can be applied to different data at the same time. In January 8, 1997, Intel proposed the 1^{st} processor with MMX technology, The Pentium MMX (P166MX) operating at frequency 166 Mhz (P166MX) is the first SIMD machine.
- **Multiple Instruction, Single Data (MISD)** Multiple instructions operate on the same data stream. It is very uncommon architecture and only few cases can use this kind of machines.
- **Multiple Instruction, Multiple Data (MIMD)** Multiple processing elements simultaneously executing different instructions on different data. This class is the more general than all previous classes. All architectures, we use in this thesis, are from this class of multicore architectures. An example of MIMD system is Intel Xeon Phi, descended from Larrabee microarchitecture. These processors have multiple processing cores (up to 61 as of 2015) that can execute different instructions on different data (Pfister, 2008).

According to architecture and microarchitecture

According (as state in Davis and Burns, 2011) to the difference (architecture and micro-architecture) between the computing elements, the multiprocessors (multicores) can be classified to:

- **Identical**: The processors are identical; hence the execution time of a processing is the same on all processors. The odroid C2 contains an identical core platform compound of 4 ARM cortex A53 processors (HardKernel, 2016).
- **Uniform:** The rate of execution of a processing depends only on the speed of the processor. Thus, a processor of speed $\times 2$, will execute a processing at twice of the rate of a processor of speed 1. The third generation of Intel $i5$ can be considered as a unifrom architecture (Intel, 2016).
- **Heterogeneous:** The processors are different; hence the rate of execution of a processing depends on both the processor and the task. Indeed, not all tasks may be able to execute on all processors and a processing may have different execution rates on two different processors operating at the same speed. The single ISA platforms such ARM big.LITTLE allow to have the same instruction set architecture in all cores, but with different microarchitecture and architecture, thus allowing a task to be executed on any core. SAMSUNG EXYNOS 5422 is a Single ISA platform compound of 8 cores: 4 "big" cores ARM A15, and 4 little cores ARM A7 (Samsung, 2016).

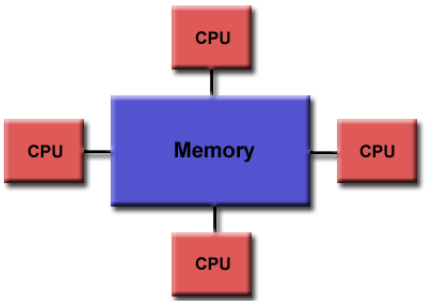


FIGURE 1.2: Shared memory model

In Chapter 4, we will be interested in uniform architectures, and in single ISA architectures, especially SAMSUNG EXYNOS 5422 processor in Chapter 5. In last chapter, we will be restricted to only identical core platforms.

According to memory architecture

- **Shared memory** A shared memory multicore (multiprocessor) offers a single memory space used by all processors. Any processors can access physically to data at any location in the memory (see Figure 1.2).
- **Distributed memory** refers to a multicore (multiprocessor) in which each processor has its own private memory. Processings can only operate physically on local data, and if remote data is required, the processing must communicate with one or more remote processors (see Figure 1.3).

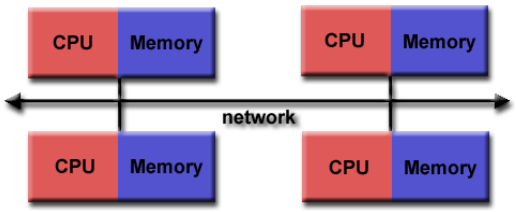


FIGURE 1.3: Distributed memory model

- **hybrid memory** refers to a multiprocessor in which each core has its own private memory, and all cores share a global memory. The largest and fastest computers in the world today employ both shared and distributed memory architectures (see Figure 1.4).

In the experiments that we present in chapter 5, we will be restricted to a platform with a shared memory.

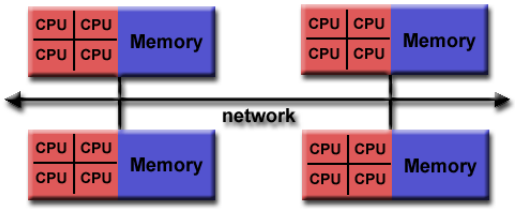


FIGURE 1.4: Hybrid memory model

1.2 Programming parallel architecture

Parallel computing is a type of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved at the same time. There are several different forms of parallel computing: bit-level, instruction-level, data, and task parallelism. Parallelism has been employed for many years, mainly in high-performance computing, but interest in it has grown lately due to cyber-physical systems. Parallel computing is closely related to concurrent computing, they are frequently used together, and often conflated, though the two are distinct: it is possible to have parallelism without concurrency (such as bit-level parallelism), and concurrency without parallelism (such as multitasking by time-sharing on a single-core CPU). In some cases parallelism is transparent to the programmer, such as in bit-level or instruction-level parallelism, but explicitly parallel algorithms, particularly those that use concurrency, are more difficult to write than sequential ones, because concurrency introduces several new classes of potential software bugs, of which race conditions are the most common. Communication and synchronization between the different subtasks are typically some of the greatest obstacles to getting good parallel program performance.

Before talking in more details about programming parallel architectures, we give a quick overview on two fundamental concepts in parallel computing: **Threads** and **Processes**.

1.2.1 Thread & Process

First, we will give basic definitions of process, thread and how a real-time task is implemented a like thread. A process is an instance of a computer program that is being executed. Each process has a Process Control Block (*PCB*) which contains information about that process. Depending on the implementation of PCB in OS[1], PCB may hold different pieces of information, commonly it contains:

- **PID**: Process Identifier
- **PPID** the Parent Process Identifier;
- **UID**: User Identifier;
- **The values of registers:** The current state of the process even if it is ready, running, of blocked;
- **Instruction Pointer (IP)** or **Program counter (PC):** it contains the address of the next instruction to execute;
- **Process addressing space**;

PCB may contains other information like the processor time, register values, $\cdots$.

A thread is the smallest sequence of instructions that can be managed independently by a scheduler. The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. If a process has only one thread, the thread is executed in sequential (single thread). If the process contains multiple threads, they execute concurrently and share resources such as memory. On uniprocessor systems, the CPU is *switched* between multiple threads/processes. When a thread/process execution is interrupted from a higher priority task, all information about the interrupted task are saved, and the information of the new one is loaded, this operation is called *Context Switch*.

[1] Operating Systems

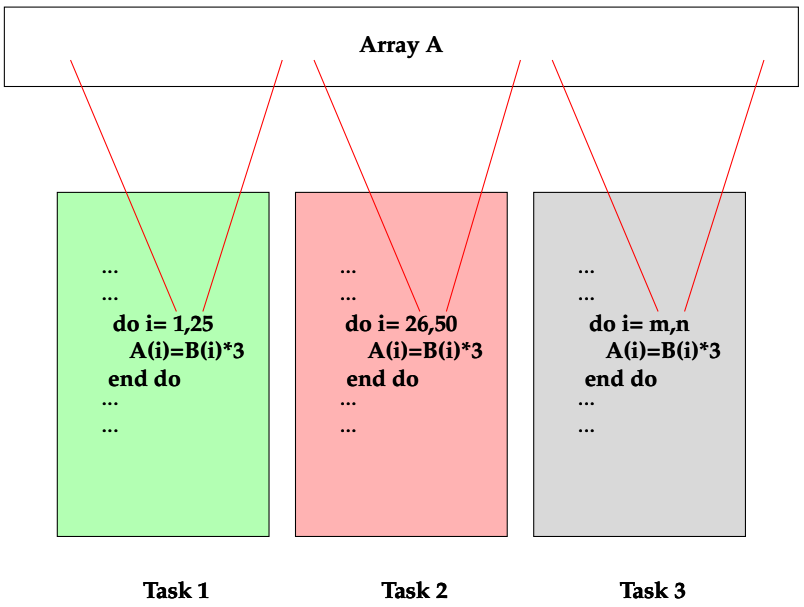


FIGURE 1.5: Example of Data parallelism

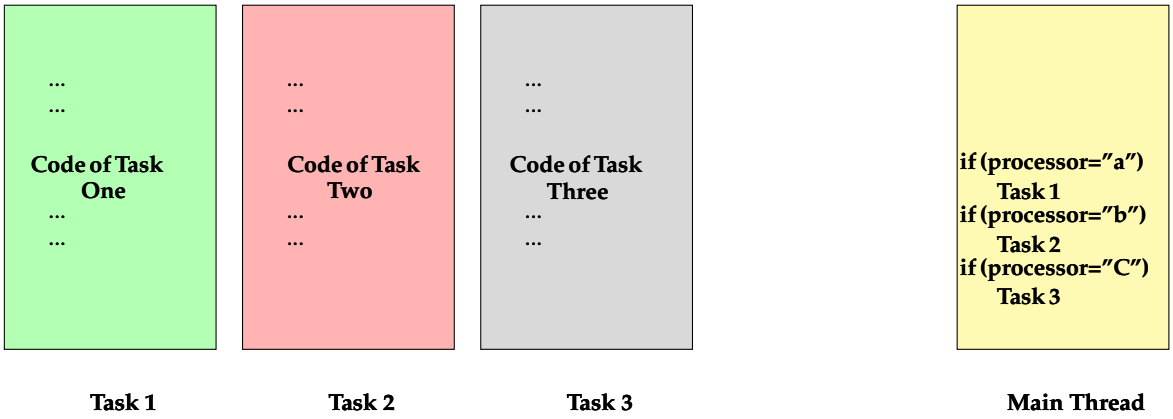


FIGURE 1.6: Example of task parallelism

1.2.2 Sources of parallelism

Data Parallelism

Data parallelism is a form of parallelization, it focuses on distributing the data across different parallel computing nodes. Data parallelism is achieved when each processor performs the same processing on different subsets of data at the same time.

In Figure 1.5, we present an example of data parallelism. Tasks **task 1**, **task 2**, $\cdots$, **task n** do the same computation is done on a sub array of Array B and results are *gathered* in Array A.

Task Parallelism

Task parallelism (control parallelism) focuses on distributing different processing across different processors. The processings may execute the same or different code.

It is of a paramount importance to consensus on programming models because the existence of different parallel computers, thereby facilitating portability of software. Parallel programming models are a bridge between hardware and software.

Classifications of parallel programming models can be divided broadly into two areas: process interaction and problem decomposition. First, we will focus on the interaction between tasks.

1.2.3 Communication models

Process interaction relates to the mechanisms by which parallel processes are able to communicate with each other. The most common forms of interaction are shared memory and message passing.

Shared memory model

Shared memory is an efficient means of passing data between processes. In a shared-memory model, parallel processes share a global address space that they read and write to asynchronously. Asynchronous concurrent access can lead to race conditions and mechanisms such as locks, semaphores and monitors can be used to avoid these. Conventional multi-core processors directly support shared memory, which many parallel programming languages and libraries, such as Cilk (Blumofe et al., 1996), OpenMP(OpenMP Architecture Review Board, 2008) and Threading Building Blocks, (TBB Pheatt, 2008), are designed to exploit.

Message passing

In a message-passing model, parallel processes exchange data through passing messages to one another. These communications can be asynchronous, where a message can be sent before the receiver is ready, or synchronous, where the receiver must be ready.

1.2.4 Decomposition & granularity of a parallel task

Granularity is the amount of real work in the parallel task. If granularity is too fine, then performance can suffer from communication overhead. If granularity is too coarse, then performance can suffer from load imbalance. The granularity of a multithreaded application greatly affects its parallel performance. When decomposing an application for multithreading, one approach is to logically partition the problem into as many parallel tasks as possible. Within the parallel tasks, next determine the necessary communication in terms of shared data and execution order. Since partitioning tasks, assigning the tasks to threads, and sharing data between tasks are not free operations, one often needs to agglomerate, or combine partitions, to overcome these overheads and achieve the most efficient implementation. The agglomeration step is the process of determining the best granularity for parallel tasks.

The granularity is often related to how balanced the work load is between threads. While it is easier to balance the workload of a large number of smaller tasks, this may cause too much parallel overhead in the form of communication, synchronization, etc. Therefore, one can reduce parallel overhead by increasing the granularity (amount of work) within each task by combining smaller tasks into a single task.

1.2.5 Limits and costs of parallel programming

Here, we present the challenges that programmer could face in the world of parallel programming:

- **Acceleration** Amdahl's law gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved.

  Amdahl's law can be formulated as:

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}} \tag{1.1}$$

  where:

  - $S_{latency}$ is the theoretical speedup in latency of the execution of the whole task;

- s is the speedup in latency of the execution of the part of the task that benefits from the improvement of the resources of the system;
- p is the percentage of the execution time of the whole task concerning the part that benefits from the improvement of the resources of the system before the improvement.

$$\begin{cases} S_{\text{latency}}(s) \leq \frac{1}{1-p} \\ \lim_{s\to\infty} S_{\text{latency}}(s) = \frac{1}{1-p}. \end{cases} \tag{1.2}$$

Furthermore, the above equation show that the theoretical speedup of the execution of the whole task increases with the improvement of the resources of the system and that regardless the magnitude of the improvement, the theoretical speedup is always limited by the part of the task that cannot benefit from the parallelization.

Amdahl's law is often used in parallel computing to predict the theoretical speedup when using multiple processors. For example, if a program needs 20 hours using a single processor core, and a particular part of the program which takes one hour to execute cannot be parallelized, while the remaining 19 hours (p = 0.95) of execution time can be parallelized, then regardless of how many processors are devoted to a parallelized execution of this program, the minimum execution time cannot be less than that critical one hour. Hence, the theoretical speedup is limited to at most 20 times (1/(1 p) = 20). For this reason parallel computing is relevant only for a low number of processors and very parallelizible programs.

- **Communication** In the general case, different threads communicate with each another as they execute. Communication usually takes place by passing data from one thread to another as part of a workflow.
- **Harder to debug** Programming multi-threaded code often requires complex coordination of threads and can easily introduce subtle and difficult-to-find bugs due to the interweaving of processing on data shared between threads. Consequently, such code is much more difficult to debug than single-threaded code when it fails.
- **Load Balancing** workload across processors can be problematic, especially if they have different performance characteristics.
- **Complexity** In general, parallel processing are much more complex than corresponding sequential processing.
- **Portability:** Thanks to standardization in several APIs such as MPI, POSIX threads and OpenMP, portability issues with parallel programs are not as serious as in past years. However, even though standards exist for several APIs, implementations differ in a number of details, sometimes requiring code modifications to ensure the portability.

1.3 Parallel models

In this section, we present two models for parallel programming, the fork-joint model and the gang model and their correspondent schedulers.

1.3.1 Fork-Join model

The forkjoin model is a way of executing parallel programs, such that execution branches alternatively between parallel and sequential execution at designated points in the program. Parallel sections may fork recursively until a certain task granularity is reached.

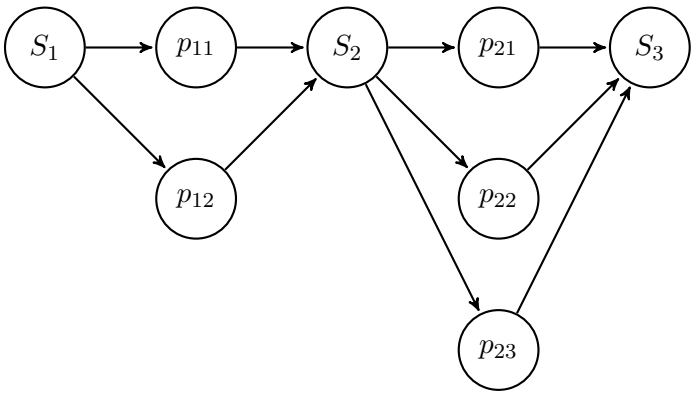


FIGURE 1.7: Example of Fork-Join Model

The fork join example given in Figure 1.7 presents an example of three consecutive forks and joins. The first sequential segment is S_1 which forks two parallel threads P_{11} and P_{12}. The second sequential segment is S_2 and it joints the two segments forked by S_1 and continue onward its execution and it forks at its turn three parallel threads P_{21}, P_{22} and P_{23}. The last task joins the threads created by S_2 and continue onward till the task ends.

Threads used in forkjoin programming will typically have a work stealing scheduler that maps the threads onto the underlying thread pool. This scheduler can be much simpler than a fully featured, preemptive operating system scheduler: general-purpose thread schedulers must deal with blocking for locks, but in the fork join paradigm, threads only block at the join point.

Fork-join is the main model of parallel execution in the OpenMP framework. although OpenMP implementations may or may not support nesting of parallel sections. It is also supported by the Java concurrency framework, the Task Parallel Library for .NET, and Intel's Threading Building Blocks (TBB).

1.3.2 Gang Model

When a critical section is used by some thread which is descheduled because its time quantum expired, then other threads attempting to access the critical section must wait until the context switch. A solution is to execute the threads of the same parallel application in parallel at the same time. Thus, task should synchronize activation, running and termination of all its threads. This scheme is called "Gang scheduling".

1.4 Designing a parallel code

Managing concurrency acquires a central role in developing parallel applications. The basic steps in designing parallel applications are:

1. **Partitioning:** The partitioning stage of a design is intended to expose opportunities for parallel execution. Hence, the focus is on defining a large number of small tasks in order to yield what is termed a fine-grained decomposition of a problem.

2. **Communication:** The tasks generated by a partition are intended to execute concurrently but cannot, in general, execute independently. The computation to be performed in one task will typically require data associated with another task. Data must then be transferred between tasks to allow computation to proceed. This information flow is specified in the communication phase of a design.

3. **revisiting:** In the third stage, development moves from the abstract toward the concrete. Developers revisit decisions which has been made during the partitioning and communication phases in so that the task will execute efficiently on some class of parallel architectures. The fine-grain decompositions can be clustered to gather to form bigger tasks and provide a smaller number of tasks.

4. **Mapping:** In the fourth and final stage of the design of parallel algorithms, the developers specify where each task will be executed.

In this thesis, we are interested in the two last aspects for parallel programming design under real-time constraints in the goal of reducing the energy consumption. Hence, in the next section, we overview two major techniques for reducing the power consumption : DVFS and DPM.

1.5 Power consumption in multiprocessor systems

Reducing power consumption in multicore systems for CPS is a very serious problem when they are operated by batteries. Even when they are connected to the electric grid, we need to keep the consumption as low as it is possible. In fact, even when the computational load is not very high, multicore processors are more energy efficient than an equivalent single-core platform Wolf, 2012.

Previously, power management on multicore architectures boils down to a simple principle "turn-off anything you do not use" in the logic of Dynamic Power Management (DPM). In modern processor, it is possible to change dynamically the *operating frequency* to reduce the power consumption, this operation is called *Dynamic Voltage and Frequency Scaling* DVFS.

1.5.1 DVFS: Dynamic voltage and frequency scaling

Increasing the frequency of a processor involves switching its transistors more rapidly, and transistors that are switched more rapidly dissipate more power. The power dissipated due to switching is called *dynamic power*.

Dynamic Voltage and Frequency Scaling (DVFS) describes the use of two power saving techniques: *dynamic frequency scaling* and *dynamic voltage scaling*. The benefit of scaling voltage and frequency is to reduce power consumption of the processor and the attached peripherals like memory. A frequency can be set on one of several available operating points. An Operating Point is a set voltage and frequency in which the processor can operate. Typically, the voltage is determined by the minimum voltage that can sustain a set processor frequency, therefore it usually does not make sense to have two different operating points at the same frequency, but at different voltages. Downscaling the frequency downgrades the timing performances of a system. Thus, calibrating the frequency to reduce the energy consumption is always coupled with a quality of service.

1.5.2 DPM: Dynamic Power Management

Even transistors that are not switching will still leak current during idle periods. This leakage current constantly dissipates power. The amount of power dissipated due to leakage current is called *static power*.

In 1996, Intel, HP, and Microsoft with Toshiba and Phoenix standardized static power management by presenting the ACPI Specification. ACPI defines which registers, piece of hardware should be available, and what information should be offered to control the processor states. The basic idea behind ACPI based power management is that unused/less used devices should be put into lower power states. Even the entire system can be set into low-power state (sleeping state) when possible. Standards designate two famillies of processor states: P-states and C-states. P-states are described as performance states; each P-state corresponds to a certain clock

speed and voltage. P-states could also be called processing states, contrary to C-states, a core in a P-state is actively processing instructions.

With the exception of C0, C-states correspond to sleep/idle states, there is no processing on a core when it is in a C-state. The ACPI standard only defines 4 CPU power states from C0 to C3: C0 is the state where the P-state transitions happen: the processor is processing. C1 halts the processor. There is no processing done but the processor's hardware management determines whether there will be any significant power savings. All ACPI compliant CPUs must have a C1 state. C2 is optional, also known as *stop clock*. While most CPUs stop "a few" clock signals in C1, most clocks are stopped in C2. C3 is also known as *sleep*, or completely stop all clocks in the CPU.

The actual result of each ACPI C-state is not defined. It depends on the power management hardware that is available on the processor. Modern processors does not only stop the clock in C3, but also move to *deeper sleep states* C4/C5/C6 and may drop the voltage of the CPU.

The total power dissipation is the sum of the dynamic and static power. The integral of dissipated power over time defines the energy consumption. In this work, we focus on minimizing the total energy dissipation.

In this thesis, several hardware architectures will be benchmarked to build a timing and power profile for both DVFS and DPM techniques.

1.6 Conclusion

In this chapter, we presented an overview on two fundamental concepts that we are going to use in this thesis: The parallelization techniques and energy saving techniques. We will be interested in the next chapters in particulary two problems of parallelization: decomposition and allocation. We will consider a set of tasks with timing constraints to be parallelized to a multicore platform with the goal of reducing the energy consumption. However before that we will give a quick overview on real-time systems in the next chapter.

Chapter 2

Introduction to real-time systems

“ Time is like a sword Either you strike it,
or it will strike you ”

Arabic proverbe

Contents

2.1 Introduction

Real-time systems are defined as those systems in which the correctness of the system depends not only on the correctness of logical result of computation, but also on the time on which results are produced Burns and Wellings, 2001. If the response time violates the timing constraints imposed by the dynamic of the processing, the system has to pay a cost for the violation. Hence, it is essential that the timing constraints of the system are guaranteed to be met. The cost of failure in a real-time system differentiates real-time systems into mainly three types of real-time systems.

If the violation of timing constraints causes the system failure, the real-time is *hard*. Missile Guidance System (MGS) and Electronic Stability Program (*ESP*) are both hard real-time systems. If the ESP is available in a car, then the car has two other sub-systems, the Anti-lock Braking System (ABS) which prevents the car wheels from blocking while braking, and Traction Control System (*TCS*) which prevents the wheels from spinning while accelerating. The ESP collects data from several sensors (wheel speed sensor, steering angle sensor, lateral acceleration sensors, etc) 25 times per second and processes the collected data. If it detects that the car is moving in an other direction than the driver guidance, it triggers several actions independently from the driver, these actions act mainly on brakes (ABS), engine, and wheel orientations in order to get the car control back. The data collected from sensors must be processed and reaction triggered within 40 milliseconds. Any delays on reaction, may cause life loss, or car damage.

The second type of real-time systems are *soft* real-time systems. In such systems, the violation of timing constraints does not leads to catastrophic consequences, but a bad user experience. Video streaming and multimedia are examples of soft real-time systems. In a soft real-time systems, the results may stay relevant even if the timing constraints are violated.

The firm real-time systems are not hard-real time systems, but results delivered after that time constraints ha been violated, are ignored. These systems are coupled to Quality of Service (QoS) that they deliver. However, it is still important to limit the number of timing constraint violations. For example, a video streaming of 25 FPS[1] can violates deadline 5 times per second, such that the video will have a frame rate at worst of 20 FPS.

2.2 Task Model

In this thesis, we refer to system functionalities as tasks. Tasks in real-time systems are recurrent. Any task can appear at any time in the system life. We call each task appearance a *job*. We denote $\mathcal{J}_{i,a}$ the a^{th} appearance (job) of the task τ_i. Each job $\mathcal{J}_{i,a}$ is characterized by the tuple $(\mathcal{A}_{i,a}, \mathsf{D}_i, \mathsf{C}_i)$. The job $\mathcal{J}_{i,a}$ is ready at time $\mathcal{A}_{i,a}$, and takes C_i time units to execute, and must finish its execution before time $\mathcal{A}_{i,a} + \mathsf{D}_i$. The job is said *active* in the time window between $\mathcal{A}_{i,a}$ and $\mathcal{A}_{i,a} + \mathsf{D}_i$. In the sporadic task model a minimum inter-arrival time between two consecutive releases (jobs) of the same task is defined. Thus, a sporadic task is characterized by the tuple $\tau_i = (\mathsf{O}_i, \mathsf{D}_i, \mathsf{T}_i, \mathsf{C}_i)$ where:

- Offset O_i: is the task release date, it represents the date of the first appearance of the task τ_i.
- Deadline D_i: is the task's relative deadline, it represents the time within the task have to end it's execution starting from the release date O_i. If D_i is equal to the period T_i the deadline is *implicit*. If D_i is less or equal to the period T_i the deadline is *constrained*. Otherwise, the deadline is *arbitrary*. If $\mathsf{D}_i = \mathsf{T}_i$ the deadline is *implicit*.
- Period or inter-arrival time T_i : it represents the minimum inter-arrival time between two releases of the same task. The worst case of sporadic arrivals is that the task is released

[1] Frame Per Second

at each T_i time units from its previous release. This assumption leads us to the periodic task model of Liu and Layland.

- Charge C_i: is the task execution time. It represents the time elapsed from the time task acquires the processor to the end of the task without being interrupted. In a lot of real-time works, this parameter represents the worst case execution time. The estimation of the execution time can be done dynamically by testing several inputs, however this technique underestimates the execution execution times, because we can not ensure that all inputs had been tested, and that the worst execution path was produced. An other technique is based on doing the analysis of the compiled code of the task and a processor model. We explore task code paths to generate the worst one, and use the processor model to estimate the execution time that could be generated by this path. These techniques are time consuming and in general overestimate the worst case execution time of the task and it depends mainly on the processor model and its correctness. Therefore, building a processor model is hard for modern processors. To have more information about the worst case execution time estimation please, refer to papers Puschner and Burns, 2000 Colin and Puaut, 2000.

Other task parameters can be defined such as:

- Laxity L_i : the laxity is the largest time for which the scheduler can **safely** delay the job execution before running it without any interruption. It is given by $L_i(t) = \mathcal{A}_{i,j}+\mathsf{D}_i-t-\mathsf{C}_i$
- Utilization u_i : it is given by the ratio $\frac{\mathsf{C}_i}{\mathsf{T}_i}$, it represents the processor occupation rate if the task τ_i is allocated on this processor.
- Density d_i : it is given by the ratio $\frac{\mathsf{C}_i}{\mathsf{D}_i}$ if the deadline is constrained otherwise it is equal to $\frac{\mathsf{C}_i}{\mathsf{T}_i}$.
- Worst case Response time R_i: is defined as the longest time from a job arriving to its completing.
- Hyper Period H: is defined as the least common multiple of all task periods.

You can see on Figure 2.1 a graphical representation of task parameters.

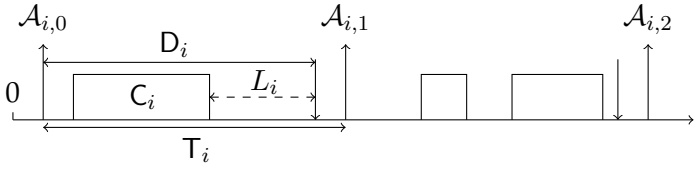


FIGURE 2.1: Periodic Task model

2.3 Priority Driven Scheduling

Scheduling real-time tasks at run-time is ordering the execution of active jobs. In a real-time operating system, a scheduler denotes the algorithms that determines which active job(s), is run on the processor(s) at each moment of time. Run-time scheduling algorithms are typically implemented as follows: at each time instant, assign a priority to each active job, and allocate the available processors to the highest-priority jobs (Sanjoy Baruah, 2015). Different scheduling algorithms differ one from another in the manner in which priorities are assigned.

A scheduling algorithm is said to be a priority driven scheduling algorithm if and only if it satisfies the condition that for every pair of jobs $\mathcal{J}_{i,a}$, $\mathcal{J}_{i\prime,a\prime}$, ,if $\mathcal{J}_{i,a}$ has higher priority than $\mathcal{J}_{i\prime,a\prime}$

at some instant in time, then $\mathcal{J}_{i,a}$ always has higher priority than $\mathcal{J}_{i\prime,a\prime}$ (Goossens, Funk, and Baruah, 2003).

According to the priority, we can classify scheduling algorithms into 3 categories:

- **Fixed Task Priority:** A task has a fixed priority during the whole system life, and all jobs of the same task, has the same priority. Rate Monotonic (**RM** Liu and Layland, 1973a) and Deadline Monotonic (**DM** Leung and Whitehead, 1982) are fixed task priority scheduling algorithms.
- **Fixed Job Priority, EDF:** A job has a fixed priority during its execution, but jobs of the same task may have different priorities. Earliest Deadline First (**EDF** Liu and Layland, 1973a) represents this class of scheduling algorithm.
- **Dynamic Job Priority:** A job priority changes at each moment of time. These algorithms are hard to implement, and have a high complexity. Least Laxity First (**LLF** Dertouzos, 2002) is a dynamic job priority scheduling algorithm.

2.3.1 Scheduling characteristics

Before detailing different scheduling algorithms, it is necessary to give more definitions.

Preemption

Preemption is the act of interrupting an executing job and invoke a scheduler to determine which process should execute next. Therefore, allowing higher priority jobs to acquire the preemptible resource.

Feasibility

A task set is said to be feasible with respect to a given system if there exists some scheduling algorithm that can schedule all possible sequences of jobs that may be generated by the task set on that system without missing any deadlines.

Optimality

A scheduling algorithm is referred as optimal if it can schedule all of the task sets that can be scheduled by any other algorithm. In other words, all of the feasible task sets.

Sufficient tests

A schedulability test is termed sufficient, with respect to a scheduling algorithm and a system if all of the task sets that are deemed schedulable according to the test are in fact schedulable.

Necessary tests

Similarly, a schedulability test is termed necessary if all of the task sets that are deemed unschedulable according to the test are in fact unschedulable.
A schedulability test that is both sufficient and necessary is referred to as exact test.

Schedulability

A task is referred to as schedulable according to a given scheduling algorithm if its worst-case response time under that scheduling algorithm is less than or equal to its deadline. Similarly, a task set set is referred to as schedulable according to a given scheduling algorithm if all of its tasks are schedulable.

Predictability

A scheduling algorithm is referred to as predictable if the response times of jobs cannot be increased by decreases in their execution times, with all other parameters remaining constant. Predictability is an important property, as in real systems task execution times are almost always variable up to some worst-case value

Comparability

In comparing the task sets that can be scheduled by two different scheduling algorithms A and B, there are three possible outcomes.

1. Dominance. Algorithm A is said to dominate algorithm B, if all of the task sets that are schedulable according to algorithm B are also schedulable according to algorithm A, and task sets exist that are schedulable according to A, but not according to B
2. Equivalence. Algorithms A and B are equivalent, if all of the task sets that are schedulable according to algorithm B are also schedulable according to algorithm A, and vice versa.
3. Incomparable. Algorithms A and B are incomparable, if there exist task sets that are schedulable according to algorithm A, but not according to algorithm B and vice versa

Sustainability

A scheduling algorithm is said to be sustainable with respect to a task model, if and only if schedulability of any task set compliant with the model implies schedulability of the same task set modified by (i) decreasing execution times, (ii) increasing periods or inter-arrival times, and (iii) increasing deadlines. Similarly, a schedulability test is referred to as sustainable if these changes cannot result in a task set that was previously deemed schedulable by the test becoming unschedulable. We note that the modified task set may not necessarily be deemed schedulable by the test. A schedulability test is referred to as self-sustainable, if such a modified task set is always deemed schedulable by the test.

2.4 Uniprocessor Scheduling

In this section, we present the uniprocessor scheduling algorithms and their corresponding schedulability tests.

2.4.1 Rate Monotonic RM

Rate Monotonic scheduling algorithm is a fixed task priority algorithm. It assigns a priority according to the task's period : the shorter the period is, the higher is the priority.

$\mathcal{T}$	C	T
τ_1	3	10
τ_2	2	15

TABLE 2.1: Example of Rate Monotonic:Task set details

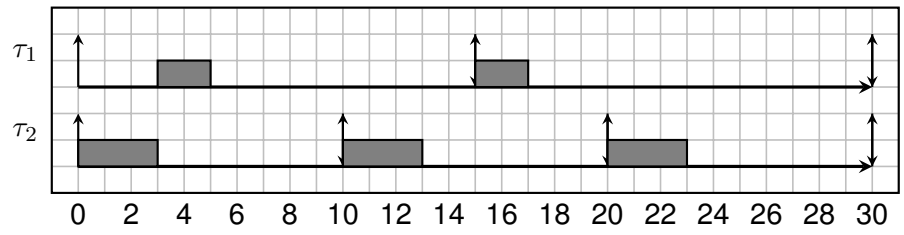


FIGURE 2.2: Example of scheduling with rate monotonic

Example: Let $\mathcal{T}$ a set of 2 periodic tasks with implicit deadlines (see table 2.1 for task details).
The total task set utilization is :

$$U = \frac{3}{10} + \frac{2}{15} = 0.43 \tag{2.1}$$

The utilization is less than 0.82, Thus the task set is schedulable under RM. Notice the same results in Figure 2.2 where a schedule is presented for the same task set in the time interval $[0, 30]$.

2.4.2 Deadline Monotonic DM

Deadline monotonic is a scheduling algorithm from fixed task priority scheduling class. With deadline monotonic, tasks priorities are assigned according to their deadlines; the highest priority is assigned the task with the shortest deadline. In contrary of Rate Monotonic, DM considers tasks with constrained deadlines. DM is optimal in its class.

Audsley et al., 1990 proposed a sufficient schedulability task under the hypothesis that tasks are sorted in a non-decreasing order of deadlines (Equation (2.2)).

$$\forall i, (1 \leq i \leq n) : \frac{\mathsf{C}_i}{\mathsf{D}_i} + \frac{I_i}{\mathsf{D}_i} \leq 1, where I_i = \sum_{j=1}^{i-1} \left\lceil \frac{\mathsf{D}_i}{\mathsf{T}_j} \right\rceil \mathsf{C}_j \tag{2.2}$$

Where I is the parameter describing the interference from higher priority tasks.

An other exact test based on the worst response time analysis can be performed. The critical instance of a task i of priority p happens when all other tasks with higher priority are active at the same time. For a synchronous task set, these moment happen at time 0. Joseph et al. proposed a test based on response time analysis R_i. The worst case response time is given by the smallest (positive) value that satisfies the recursive equation in Equation (2.3) where hp(i) denotes the set of tasks with a higher priority than τ_i;

$$R_i = \mathsf{C}_i + \sum_{j=hp(i)} \left\lceil \frac{R_i}{\mathsf{T}_j} \right\rceil \mathsf{C}j \tag{2.3}$$

We can use the following recurrent equations to calculate the worst case response time (Audsley et al., 1993).

$$R_0 = C_i \tag{2.4}$$

$$R_i^{k+1} = \mathsf{C}_i + \sum_{j=hp(i)} \left\lceil \frac{R_i^k}{\mathsf{T}_j} \right\rceil \mathsf{C}j \tag{2.5}$$

Example Let $\mathcal{T}$ a set of 3 periodic tasks with implicit deadlines (see Table 2.2, P is the task priority).

By using Equations (2.4) and (2.5), we obtain the results of the columns R in Table 2.2. We can notice that task τ_3 worst case response time is 29 which is greater that it's deadline which is

$\mathcal{T}$	C_i	T_i	D_i	P_i	R_i
τ_1	5	10	9	2	9
τ_2	4	15	7	3	4
τ_3	6	30	15	1	29

TABLE 2.2: Example of Deadline monotonic: Task set details

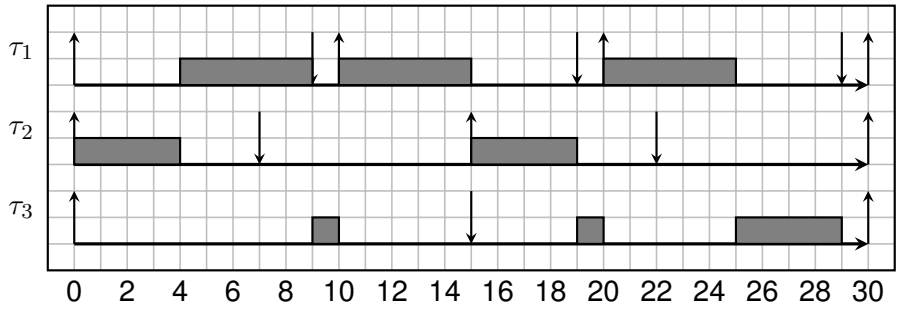


FIGURE 2.3: Example of scheduling with Deadline Monotonic

15. Thus, the task is not schedulable. This is confirmed in the scheduling simulation on Figure 2.3.

2.4.3 Earliest Deadline First

Earliest deadline first (**EDF**) is a fixed job priority scheduling algorithm. At each scheduling event (job termination or arrival), the scheduler assigns the job with the smallest value of $\mathcal{A}_{i,a} + \mathsf{D}_i$ the highest priority. EDF is an optimal scheduling algorithm on preemptive uniprocessor.

For implicit deadline task, EDF has a utilization bound of 100%. Thus, an exact schedulability test can be driven based on utilization as following if all deadlines are implicit:

$$U = \sum_{i=1}^{n} \frac{\mathsf{C}_i}{\mathsf{T}_i} \leq 1 \tag{2.6}$$

For constrained deadlines tasks, lot of sufficient testes where driven. The response time based analysis are hard to perform (Guan and Yi, 2014,Spuri, 1996), since that the critical moment does not arrive at time 0 like for DM. In the following, we present a demand based exact schedulability test for synchronous tasks for uniprocessor EDF.

Demand Bound Analysis

Let $\mathcal{T}_{=}\{\tau_i, i \in \{1 \dots n\}\}$ be a set of synchronous periodic tasks and let t be a non-negative integer. The demand bound function $\mathsf{dbf}(\mathcal{T},t)$ denotes the maximum cumulative execution requirement that could be generated by jobs of $\mathcal{T}$ that have both ready times and deadlines within any time interval of length t. Equation (2.7) describes the formula for computing the dbf:

$$\mathsf{dbf}(\mathcal{T},t) = \sum_{\tau_i \in \mathcal{T}} \left\lfloor \frac{t + \mathsf{T}_i - \mathsf{D}_i}{\mathsf{T}_i} \right\rfloor \mathsf{C}_i \tag{2.7}$$

Baruah, Rosier, and Howell, 1990 proposed the following exact schedulability test for a set of synchronous task sets:

$$\forall t \leq t^*, \mathsf{dbf}(\mathcal{T}_j, t) \leq t \tag{2.8}$$

where t^* is a constant that depends on the utilization of the task set (see Baruah, Rosier, and Howell, 1990 for more details on the analysis algorithm). In this work, we consider $t^* = h$,

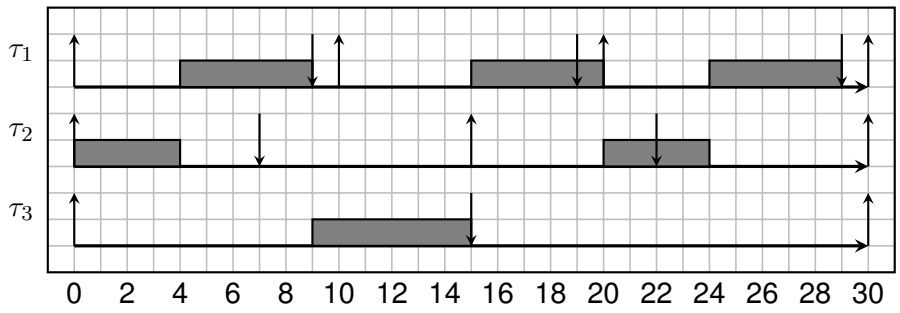


FIGURE 2.4: Example of scheduling with EDF

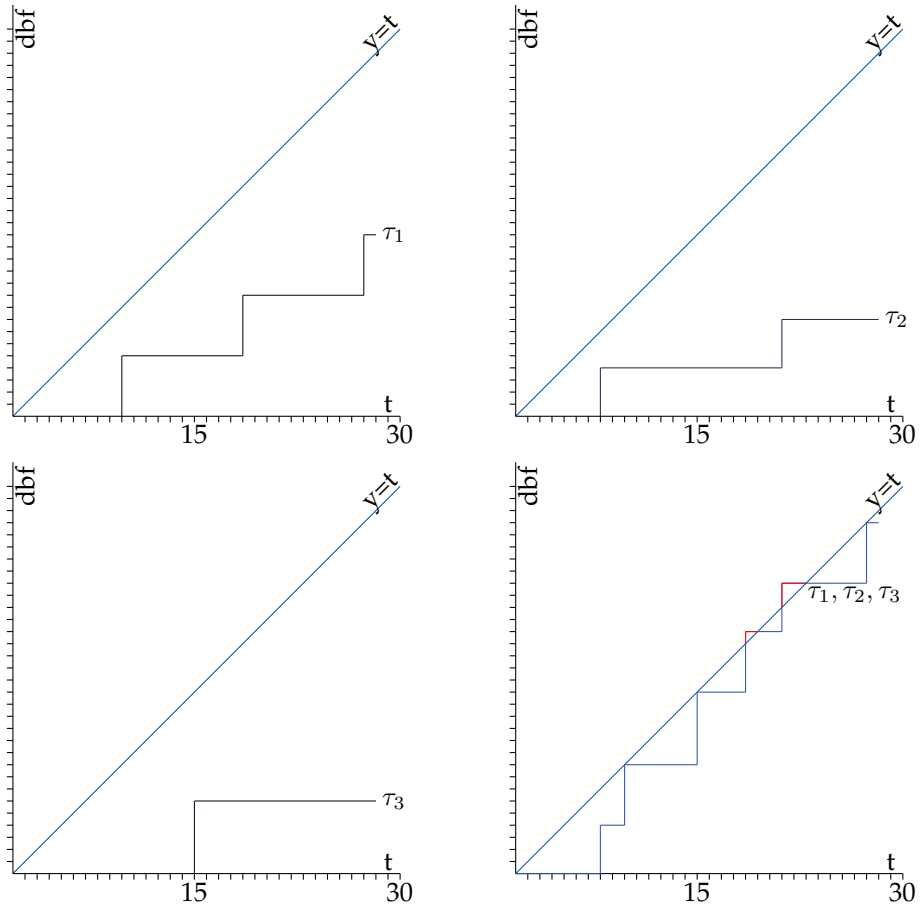


FIGURE 2.5: Demand Bound Function

where h is the hyper period of tasks, it is computed as least common multiple of all periods of tasks.

Example Let $\mathcal{T}$ be the same task set as the example 2.2.

The simulation of scheduling this task set (see Figure 2.4) shows that the task set is not schedulable under EDF, and that at moments $t = 19$ and $t = 22$ deadlines are missed. This could be also driven by using the demand bound function analysis on Figure 2.5, where we can see that at the same moments (19,22), the value of the demand bound function ($\mathsf{dbf}(\mathcal{T},19) = 20, \mathsf{dbf}(\mathcal{T},22) = 24$).

2.5 Multiprocessor Scheduling

With the revolutionary impact of multiprocessor architecture onto systems design, a lot of work concerning real-time multiprocessor scheduling has been proposed. Mainly, two approaches had been followed: the first one try to extend uniprocessor scheduling techniques into multiprocessors, and the second one proposes completely new techniques dedicated for multiprocessor architectures. Extending an optimal algorithm of uniprocessor does not allow to have an optimal multiprocessor algorithms (please refer to Levin et al., 2010). In this section, we will focus on global, partitioned and semi partitioned scheduling techniques. scheduling. The classification of the multiprocessor scheduling algorithms, that we present here, is based on ability of jobs/taks to migrate from a processor to another (see Figure 2.6).

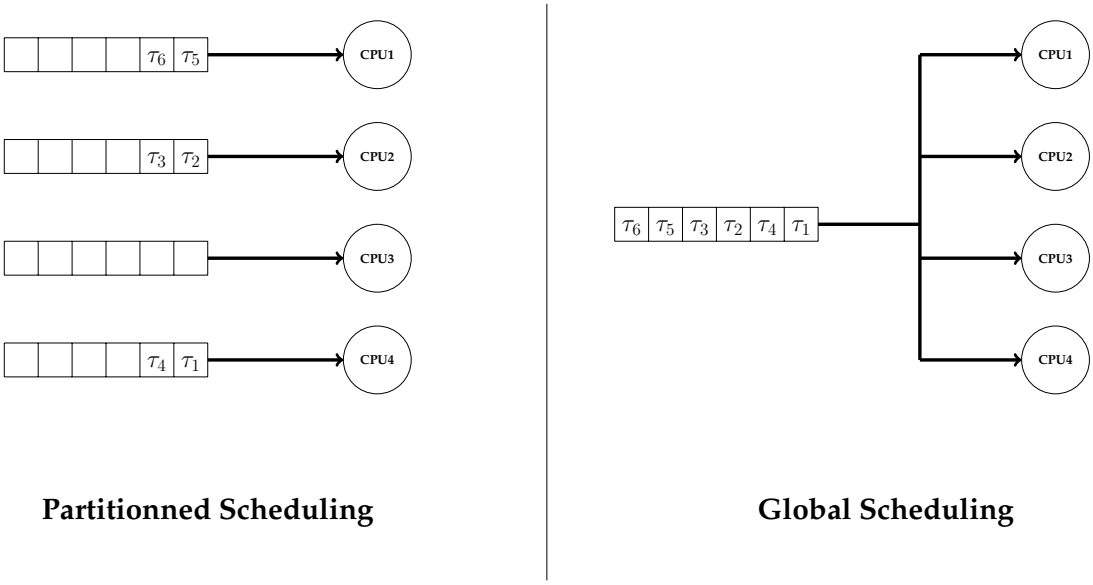


FIGURE 2.6: Partitioned Scheduling vs Global Scheduling

2.5.1 Partitioned Scheduling

Partitioning tasks among processors is transforming the problem of allocation to m processors to m uniprocessor problems. Most partitioning algorithms passes trough three steps:

1. Sort tasks in order of some criteria (period, deadline, density, utilization, etc.);
2. Assign tasks in the order of step 1 to a processor where it will always meet all deadlines when assigned to that processor, and it does not cause another previously-assigned task to miss a deadline. If a task verifies these conditions, the task *fits* on this processor. This step is performed using schedulability tests.
3. After each task has been assigned to a processor, we use the well-known uniprocessor scheduling algorithm on each processor to schedule the processor's respective tasks.

Bin-Packing problem

The problem of step 1 and 2 is a Bin-Packing Problem (BPP). In the bin-packing problem, objects of different volumes v_i must be packed into a finite number containers each of volume V in a way that it minimizes some objectives. In computational complexity theory, it is a combinatorial NP-complete problem. A variant of bin packing that occurs in practice is when items can share space when packed into a bin. Specifically, a set of items could occupy less space when packed together than the sum of their individual size, our research on real-time systems

are classed in this category of bin-packing problem. If items can share space in arbitrary ways, the bin packing problem is hard to even approximate. In the next chapters, we will present LP, ILP, INLP formulations of bin-packing problem in our context. In real-time scheduling, items are tasks, and containers are processors.

The fact that the bin packing problem has an NP-hard computational complexity, lead to develop many heuristics like First Fit, Best Fit, Worst Fit.

First Fit (FF) algorithm provides a fast but often non-optimal solution, involving placing each task into the first processor in which it will fit. It requires $\theta(n \log n)$ time, where n is the number of tasks to be allocated. The algorithm can be made much more effective by first sorting the list of elements into decreasing order (sometimes known as the first-fit decreasing algorithm) **FFD**, although this still does not guarantee an optimal solution, and for longer lists may increase the running time of the algorithm. It is known, however, that there always exists at least one ordering of items that allows first-fit to produce an optimal solution.

Best Fit (BF) is driven from **FF** algorithm by assigning the current task to the feasible processor that have the smallest residual capacity (utilization). It has been proved that in terms of reducing the number of used cores, the lower-bound of BF is FF.

Worst Fit (WF) is driven from **BF** algorithm, instead off assigning the current task to the feasible processor that have the smallest residual capacity (utilization), it is assigned to the one with the largest residual capacity.

2.5.2 Global Scheduling

In contrast to partitioned scheduling, the global scheduling allows task migration. All core have the same ready-queue and the m highest priority jobs are run at the same time on m processors. Global scheduling has several advantages compared to partitioned scheduling because it allows fewer context switches/preemption. This is because the scheduler will only preempt a task when there are no processors idle When a task executes for less than its worst-case execution time, the slack time of the task can be utilized by all other tasks, not just those on the same processor. If a task overruns its worst-case execution time budget, then there is arguably a lower probability of deadline failure as worst-case behavior of the entire system, with all tasks taking worst-case execution times, worst-case phasing occurring, etc., is less likely across multiple processors than it is on a single processor. In global scheduling we find two kinds of migrations:

- **Job level migration:** In this kind of migration, a job can start its execution of a processor and be interrupted to pursue its execution in an other.
- **task level migration:** Here, a job is executed on only one core, but jobs of the same task may be executed on different processors.

A natural adaptation of the uniprocessor scheduling into global multiprocessor architecture is proposed, Global EDF, Global DM, $\cdots$ are example of these adaptations. At each algorithm the priority is assigned to a task in the same logic like uniprocessor system but the m highest priority tasks(jobs) are run in parallel and at the same time. When a task arrives and no processor is idle, the lowest priority task is interrupted, and the new job is run on the freed processor.

2.5.3 Semi-Partitioned

Andersson and Tovar, 2006 aimed at addressing the fragmentation of spare capacity in partitioned systems is by splitting a small number of tasks between processors. Thus, they proposed

an approach to scheduling periodic task sets with implicit deadlines, based on partitioned scheduling, but splitting some tasks into two components that execute at different times on different processors.

Andersson, Bletsas, and Baruah, 2008 developed the idea of job splitting to cater for sporadic task sets with implicit deadlines. In this case, each processor p executes at most two split tasks, one also executed by processor $p1$ and one also executed by processor $p + 1$. Later, they extended this approach to task sets with arbitrary deadlines.

2.6 Programming Real-time systems

In this section, we address the real-time systems programming. We focus on the link between programming real-time systems with PosixThreads (Pthread) in C language and the periodic task model of Liu and Layland. Periodic tasks are implemented as an infinite loop of the task code. At each iteration, the task sleeps from its end until its next wake-up period. A typical code of a real-time task in C programming language can be :

```
void  rtTask()
{
  while (true)
  {


    // task code

    WaitForNextPeriod();
  }
}
```

waitForNextPeriod is implemented by using **clock_nanosleep** syscall which takes 4 parameters. The first one is the clock type offered by Linux Kernel. We use **CLOCK_REALTIME** clock because it has a good compromise between clock accuracy and latency. The second parameter is the type of time, here we use absolute time. The third parameter is the desired awaking time. In the case of periodic tasks, this parameter is set to the next period and it is computed at each iteration. The last parameter is the remained time in the case of sleeping failure due to a wake-up event. Thus, the typical real-time task code in C will become :

```
void rtTask(void *arg) {
  while(true)
  {
    struct timespec next,begin;
    clock_gettime(CLOCK_REALTIME, &begin);
    next=TimeAddOperation(begin, taskPeriod);


    // task code


    clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &rem,
    NULL); } }
```

A real-time task can be implemented using a process or a thread in the same manor. However, in most RTOS[2] such FreeRTOS(Barry et al., 2008), RTAI(Mantegazza, Dozio, and Papacharalambous, 2000), VxWorks (Guide, 1999), tasks are implemented as threads. In the next section, we show how to use POSIX threads on Linux kernel to implement real-time threads.

Traditionally, applications that have real-time constraints, use custom-built hardware and RTOS to meet their real-time requirements. However, for some soft real-time requirements,

[2]Real-time Operating Systems

this may be an expensive solution. With the advent of the PREEMPT_RT patch-set led by Ingo Molnar (Molnar, 2009), a number of modifications (such as the scheduler, interrupt handling, locking mechanism, $\cdots$) were made to the general-purpose Linux kernel to make Linux a viable choice for real-time systems.

2.6.1 Real-time Scheduling policies In LINUX Kernel

The standard Linux kernel provides two real-time scheduling policies, SCHED_FIFO and SCHED_RR (Garg, 2009). The main real-time policy is SCHED_FIFO. It implements a first-in, first-out scheduling algorithm. When a SCHED_FIFO task starts running, it continues to run until it voluntarily yields the processor, blocks or is preempted by a higher-priority real-time task. It has no time slices like Linux CPS[3]. All other tasks of lower priority will not be scheduled until it frees the CPU. Two equal-priority SCHED_FIFO tasks do not preempt each other. SCHED_RR is similar to SCHED_FIFO, except that such tasks are allotted time slices based on their priority and run until they exhaust their time slice. Non-real-time tasks use the SCHED_NORMAL scheduling policy (older kernels had a policy named SCHED_OTHER).

In the standard Linux kernel, real-time priorities range from zero to (MAX_RT_PRIO-1), inclusive. By default, MAX_RT_PRIO is 100. Tasks with real-time priorities are organized on a run-queue in a priority-indexed array of type struct rt_prio_array. An rt_prio_array consists of an array of sub-queues and only one sub-queue is available per priority level.

2.6.2 POSIX Threads

Usually referred to as Pthread (Nichols, Buttlar, and Farrell, 1996), allows a program to control multiple different concurrent flows. Each flow is referred as a thread. Threads creation and control is achieved by making calls to the Pthread API. This API is defined by the standard POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995). Implementations of Pthread are available on many Unix-like POSIX-conferment operating systems such as Linux.

Pthread defines a set of C programming language types, functions and constants. It is implemented with a “pthread.h” header and a thread library. There are around 100 Pthread procedures, all prefixed “pthread_” and they can be categorized into four groups:

- Thread management;
- Mutexes;
- Condition variables;
- Synchronization with locks and barriers.

The POSIX semaphore API works with POSIX threads but is not part of threads standard, having been defined in the POSIX.1b, Real-time extensions (IEEE Std 1003.1b-1993) standard. It is used to synchronize threads.

The following table resumes the Pthread primitives that we use in the rest of this thesis to benchmark hardware architectures:

All these primitives and scheduling policies are used in the next chapters to benchmark a specific hardware architecture and study the impact of several factors on the execution of a parallel real-time tasks.

2.7 Conclusion

In this Chapter, we overviewed the theory of scheduling in real-time systems for uniprocessor and multiprocessor systems. We presented several scheduling policies and their corresponding

[3]Completely Fair Scheduler

Primitive	role
pthread_create	create a thread
pthread_join	join a thread
pthread_attr_init	intialize pthread attributes
pthread_setaffinity_np	allow to allocate a task to a processor or a set of processors
pthread_self	allow to a thread to refer to it self
pthread_attr_setschedpolicy	define the linux scheduling policy
pthread_attr_setschedparam	define the scheduling parameters such as priority
pthread_mutex_lock	It uses a mutex semaphore to protect a critical section
pthread_mutex_unlock	allows to unlock a critical section lock with a mutex lock
pthread_cond_wait	allows to lock a section waiting for a signal
pthread_cond_signal	free a section locked by a wait

TABLE 2.3: The used Pthread primitives and their role

feasibility tests. We had show also how a real-time of Liu and Layland model is implemented in real-time operating systems with a special focus on linux kernel.

At the end of this chapter, all the ingredients are ready to cook the problem that we address in this thesis.

Chapter 3

Parallel Real-time: Related work

“ Il n’est rien de plus precieux que le temps,
puisque c’est le prix de l’ternit. ”

Louis Bourdaloue

Contents

3.1 CPS Systems Needs

3.1.1 Real-time needs

The rapid development of information technologies in the past decades has resulted in wide availability of embedded computing platforms and communication capabilities in almost all types of objects. Such large-scale and parvasive deep embedding of the cyber intelligence into the physical world has created unprecedented amount of data.

Cyber-physical systems (CPS) are a promising class of systems featuring tightly integration between the cyber intelligence and the physical world that allow to cope with the increasing needs of computation power. Enabled by the ubiquitous availability of computation and communication capabilities (Rajkumar et al., 2010), the CPS covers an important applications ranging from macro-scale infrastructure based systems, such as smart grid (Karnouskos, 2011), data centers (Parolini et al., 2012 Rao et al., 2012), transportation systems, to micro-scale systems, such as intelligent medical devices (Lee et al., 2012). Almost every device today has a controller that reads inputs through sensors, does some processing and then performs actions through actuators. Examples include controller systems in cars (Anti-lock Brake System, Cruise Controllers, Collision Avoidance, etc.), automated manufacturing, smart homes, robots, etc. These controllers are discrete digital systems whose inputs are physical quantities (e.g., time, distance, acceleration, temperature, etc.) and whose outputs control physical devices.

The computational components coordinate and control the physical operations to satisfy the overall requirements. Most of computational components in CPS applications are control systems which are good at managing and regulating the behaviors of physical components.

In cyber-physical systems (CPS) the passage of time becomes a central feature of system behavior, in fact it is one of the important constraints distinguishing these systems from distributed computing in general. Time is central to predicting, measuring, and controlling properties of the physical world. Thus, Most CPS must deal with real-time constraints. That is, a computational task needs to be accomplished correctly in terms of not only functionality but also time. A delayed reaction can lead to unsatisfied customers (such as in a video streaming) or total catastrophy (such as in a vehicle anti-lock braking). The real-time requirements of CPS are generally described by deadlines associated with tasks. A task is typically executed repeatedly. Each instance of a task is a job in the real-time scheduling area. Since a computational component in CPS almost always handle multiple real-time tasks, the execution order of such tasks plays a big role in meeting the deadlines. CPS is required to have predictable and reliable behaviors in terms of meeting the time requirements of all tasks instead of completing a single task fast. The predicability and reliability of CPSs can be satisfied by employing certain real-time scheduling methods in the CPS design.

Cyber physical systems are the need of today and hence they must be made compatible enough to handle various types of events may be periodic or aperiodic whose validation is critical to the correct behavior of the system. e.g., in an industrial plant monitoring, an aperiodic alert may be generated when a series of periodic sensors readings meet certain detection criteria. Varying user inputs and certain other parameters may trigger other real-time aperiodic events.

Thus, classic real-time task models may not be able to express properly CPS real-time applications. Thus, we focus on expressing expressive task models in this thesis.

3.1.2 Parallel computing needs

With this trend of CPS systems, embedded real-time systems are essential in order to sense the physical environment, process data in real-time and control the actuators. Autonomous driving is concrete example of applying real-time into CPS technology. In an autonomous car, motion planning, sensor fusion, computer vision and other articial intelligence algorithms must run in real-time. Most algorithms for autonomous driving are parallelizable (Kim et al., 2013). In order for the vehicle to understand its surroundings, the perception subsystem should be able

to process massive amounts of data from various types of sensors. This lead to a very fine grain parallesim for both data and task parallelism. Automotive industry has already started moving towards the multicore processors for higher performance. AUTOSAR (Fennel et al., 2006), a widely used automotive software infrastructure, supports multicore processors. There has been relatively little research on tackling chalenges in modeling and scheduling parallel real-time tasks. We will reports the works that has been done in this field in the next section, and later we focus on some of the energy-efficient real-time scheduling works.

3.1.3 CPS and energy consumption

The issue of energy is important. The greedy need of computing power trends increase the processor operating frequency, this factor yields an exponential increase in power needs. Moreover in 2006, data centers in the US used 59 billion kilowatthours of electricity, costing the US 4.1$ billion and 864 million metric tons in carbon dioxide emissions; this accounted for 2 percent of the total USA energy budget. The increasing cost of power press for developping energy-aware scheduling algorithms. The energy consumption must be controlled and optimized especially when the computational elements are operated by battery power.

3.2 This work

In this work, we tacle three of the discussed CPS issues, which are:

1. garanteeing that all timing requirements will be respected,
2. by usings parallel computing techniques,
3. while reducing the energy consumption.

We tacle the first problem by focusing on real-time scheduling theory for multiprocessor architectures. In the real-time system literature, and as stated earlier, there are mainly two approches for multiprocessor systems scheduling: partitioned and global scheduling. Thus, the question that must be revealed is: *what to select?*:

- Partitioned or global scheduling?
- and what scheduling algorithm?

The second issue to tacle is :

- How are we going to adress the problem of parallizing real-time tasks?
- and at which level of parallelizing process?

Finally,

- what energy saving techniques are we going to use?
- and what is the more efficient way to do it ?

3.2.1 Global or Partitionned?

Partitioned and global scheduling are incomparable, since some task sets are feasible with partitioned scheduling that are not feasible in global and some task sets are feasible with global scheduling and that are not feasibile in partitioned scheduling. Here we present a short comparaison between both :

As you can notice in Table 3.1, Global schedulability analysis are not mature and the critical instance is unknown. In contrast, partitioned scheduling aims to convert the problem into a

Global Scheduling	Partitionned scheduling
Automatic load balancing Lower avg. response time Simpler implementation Optimal schedulers exist More efficient reclaiming	Supported by automotive industry No migrations Isolation between cores Mature scheduling framework
Migration costs Inter-core synchronization Loss of cache affinity Weak scheduling framework	Rescheduling not convenient NP-hard allocation Cannot exploit unused capacity

TABLE 3.1: Comparaison between global and partitioned scheduling

uniprocessor problem, thus the schedulability tests for this classe of algorithms are well-known and mature. Another important aspect about partitioned scheduling is migration. Let assume a preempted thread, that was previously executing on an ARM-Cortex A7 core, resumes it execution on a ARM Cortex A15. There is no-way to evaluate the rest of its execution time on that core. Thus, the analysis are very hard to be done for such architectures where migration is allowed. Hence, we decided to choose partitioned scheduling.

As partitioned scheduling converts the problem of scheduling real-time tasks into a set of uniprocessor scheduling problems, we decided to select earliest deadline first (EDF) scheduler which is known to be an optimal algorithm for single processor architectures.

3.2.2 What kind of parallelism to do and where?

As we stated in Section 1.4 of Chapter 1, designing a parallel code needs typically 4 steps. The two first steps are generally made by the programmer were he decides to express a very fine grain parallelism and the task communication mechanism. The two last steps consist in defining how sub-tasks are grouped to gather in threads, and where these grouped threads are allocated. In this thesis, we focus exactly on the two last steps. We propose a methodology that allows to revise the decisions of the programmer by grouping threads to gather and allocating these grouped threads on cores in the goal of reducing the energy consumption while all deadlines are met.

As the programmer defines the parallelisable segments of its code, we address both task and data parallelism in the same way. Thus, in the next chapter we propose a task model adapted to data parallelism. In Chapter 5, the model is more realistic and allow to express both task and data parallelism. In both Chapter 4 and 5 the parallelism grain is defined off-line and can not change during the execution. In the last chapter, we propose a general model that allow to express both data and task parallelism and we allow the grain of parallelism to change during execution.

3.2.3 What energy saving techniques are we going to use?

In order to achieve an optimal decomposition with respect to the energy consumption for a set of tasks on heterogeneous multicore platform we need to (i) decompose each task into a set of parallel threads (if possible/desirable); (ii) perform a schedulability analysis and allocate the threads on the cores to guarantee that each thread completes before its deadline, and (iii) set the operating frequency of cores and their state to reduce the energy consumption. As stated in Chapter 1, we address the energy consumption problem by using DVFS or DPM techniques. As you can notice in Figure 3.1 (Maxfield, 2012) which represents the static and dynamic power consumption for different architectures when cores are operating at their maximum performances, different architectures consume different static and dynamic power. But

in general dynamic the dynamic power is greater than the static one. Hence, for such architectures it is better to calibrate the operating frequency to reduce energy consumption but, as we will show in Chapter 5, when the frequency is very low, the static energy is higher and it is more convenient to turn "off" some cores.

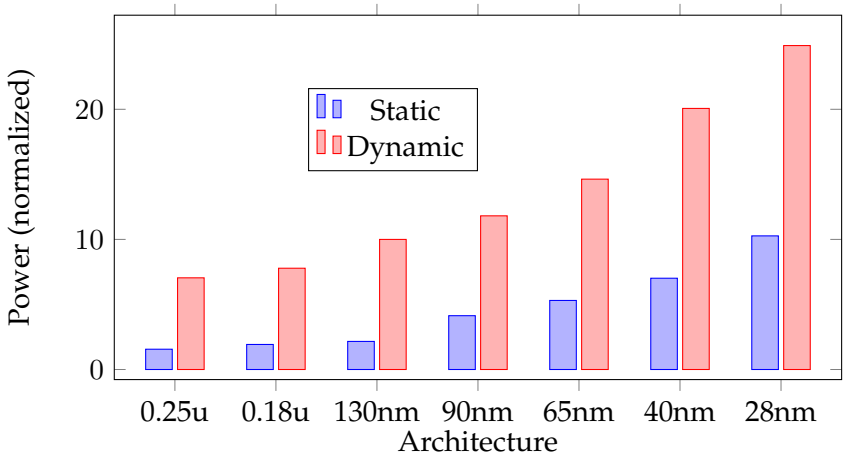


FIGURE 3.1: Comparaison between static and dynamic energy in different technologies

Moreover, it is expected that the static power take the lead to the dynamic power for close futur architectures as it is show in Figure 3.2 (Salman and Qi, 2011). Hence, it is important to model the architecture and adapt the allocation techniques according to different architectures. Thus, we will benchmark two hardware architectures in Chapter 5 and show how our heuristic can be easily adapted to different hardware platforms.

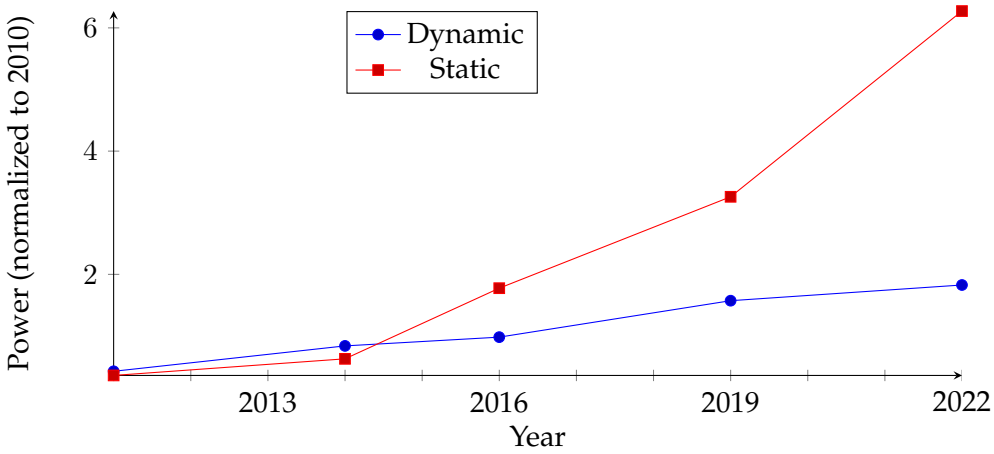


FIGURE 3.2: The memory power dissipation by one little and big core

3.3 Related work

Now that we presented what motivate the research reported in this thesis, we overview the different works done in the area of our interest.

3.3.1 Taxonomy on parallel real-time tasks

Parallel real-time task models are classified according to the way the level of parallelism is specified and to the moment of this specification. We can distinguish three types of models:

- *rigid*: the number of processors assigned to a task is specified independently before the scheduling analysis and does never change;
- *moldable*: the number of processors assigned to a task is specified off-line using an *off-line analysis algorithm*, for example during pre-processing or compilation;
- *malleable*: similar to moldable, but the number of processors assigned to a task may dynamically change during execution.

According to Drozdowski, 2004; Mounie and Drozdowski, 2004, most parallel applications in the real world are moldable. Hence, in chapter 4,5 we focus on task models and their scheduling. In Chapter 6, we propose a task model that can express both moldable and malleable tasks.

In parallel computing, a computational task is typically broken down in several subtasks that can be processed *independently* and whose results are combined afterwards, upon completion. Task decomposition is a well-known problem in the parallel programming community. Several API had been proposed to help the programmer to write a parallel code such as OpenMP, Cilk, TBB, $\cdots$.

3.3.2 OpenMP

OpenMP Chandra, 2001 is an API specification for parallel programming. The section of code that is meant to run in parallel is marked with a pre-compiler directive. One example of such directive is the *pragma parallel for*. In the OpenMP parallel model, a parallel *for* loop preceded by the STATIC schedule clause is implemented by distributing the N iteration on p threads into approximately $\frac{N}{p}$ iterations per each thread (if no further specification is implied). After the execution of the parallelized code, the threads join back into the master thread, which continues onward to the end of the program. An example of this directive is presented in listing below, which represents the addition of two 2D arrays. If $N = 16$ and $p = 4$, the first thread will sum the rows between 0 to 3, the second from 4 to 7, and so on.

```
int i, j;
int [][] mat1, mat2; // NxM matrix
int [][] res;

//Load mat1 and mat2
#pragma omp parallel for
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    res[i][j] = mat1[i][j] + mat2[i][j];
```

The example in the listing shows how data parallelism is expressed by OpenMP. OpenMP offers also support for task parallelism with parallel section pragma. With such pragma, different parallel sections threads execute different code and the parallel threads are executed separatly.

However, this way of decompositing a task does not take into account the task scheduling, the respect of the real-time and energy constraints and the processor frequency definition. In the rest of this chapter, we will present several related works that has a focus on task parallelization for real-time task with and without energy-aware scheduling.

3.4 Parallel Real-time tasks & scheduling

Han and Lin, 1989 analyzed the effect of job parallelism on the complexity of multiprocessor scheduling of hard real-time systems. They considered a system which consists of a set of independent parallel jobs and they proved that parallel fixed priority scheduling on multiprocessor systems is NP-Hard and an exact schedulability analysis of job parallelism is intractable. Based on the characteristics of parallel tasks and their internal structure, parallel scheduling is divided into two approaches: Gang Model and multithread Model.

3.4.1 Gang Model

Gang scheduling and coscheduling was proposed in Ousterhout, 1982; Feitelson and Rudolph, 1992; Gehringer, Siewiorek, and Segall, 1987 in order to grant the processors to the threads of the same task at the same time quantum. The difference between coscheduling and Gang scheduling is that the first allows tasks to continue its execution even if the number of cores needed is less than the number of available cores. This may imply the change of the parallelism level at run-time. Gang model is more strict and does not allow a task to execute if the number of processor required is less than the number of available cores. Figure 3.3 allows to see the difference between both considering a platform with 4 processors. Notice that in coscheduling, task τ needs 2 processors to run. At time 0, it is has the highest priority, thus it acquires two processors until a more priority task arrives (blue one). The new task requires 3 processors and acquires them. Thus, task τ adapts its parallelism level, and continue its execution in only one thread. At the end of the blue task, task τ continue its execution but this time with two thread until the arrival of a new task (the green task). The green task requires only two cores, and notice that in this case, both tasks are running at the same time. When a higher priority task arrives (gray task) and it requires all processors, task τ is suspended, and continue its execution after the end of the gray task. Notice that for the same scenario, that the task τ was suspended at the arrival of the blue task. Thus coscheduling considers malleable tasks, and gang scheduling the moldable tasks. Thus, gang scheduling, then, is defined as the scheduling in which all parallel threads are forced to execute simultaneously on multiple processors.

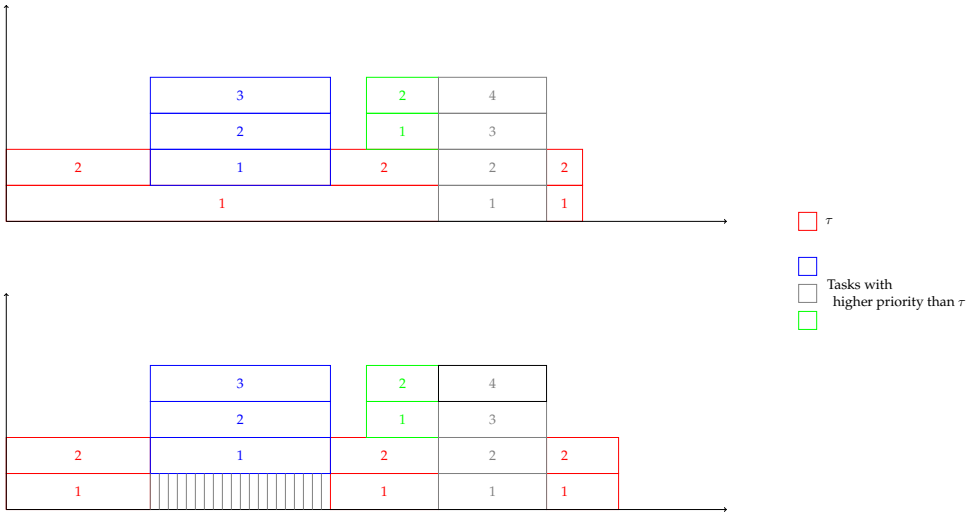


FIGURE 3.3: Difference between Gang and Co scheduling

Kato and Ishikawa, 2009 adapted the gang task model to a gang real-time task model and proposed the Gang EDF scheduling algorithm and the corresponding feasibility test. This feasibility test has been proven to be wrong in the paper of Goossens, et al. 2016. In Kato's task model, a task is defined by the number of processors used simultaneously, its execution time, period, and constrained deadline. All threads of a parallel section in a gang model have to be run in parallel (they must start and end the execution on their processors simultaneously).

However, this model is difficult to implement, because the scheduler needs to synchronize the execution of the threads among different cores and when a thread is preempted, we need to preempted all parallel threads of the same job, actually operating systems does not support this kind of preemption, hence modifications should be ported to operating system to support this kind of preemption. Kato et al. considered an identical core platform with m processing units. Gang EDF is basically Global EDF. In contrast to classic global EDF, in gang EDF a high priority task at time t may not be run at that time if the number of available processors is less than the number of the parallel threads of this task. However, they take into account the number of cores required by a task, and the number of available cores while in Global EDF, the m ready tasks are selected to be run.

Goossens and Berten, 2010 provided an exact schedulability condition for FTP Gang scheduling on identical processor.

3.4.2 Multithread Model

In contrast to the Gang model, parallel threads in multithreaded task models are scheduled independently and may start and end at different times. Multi-thread model is easier than Gang model to implement by the mean of different parallelization APIs such as OpenMP, Cilk, TBB, etc. Here we describe the multithreaded task models proposed in the literature of real-time systems

Fork-join model

Lakshmanan, Kato, and Rajkumar, 2010 introduced *fork-join* model for parallel real-time tasks: each task is a sequence of alternating parallel and sequential *segments* (See Figure 3.4). They considered a set of periodic implicit-deadline tasks to allocate to an identical core platform.

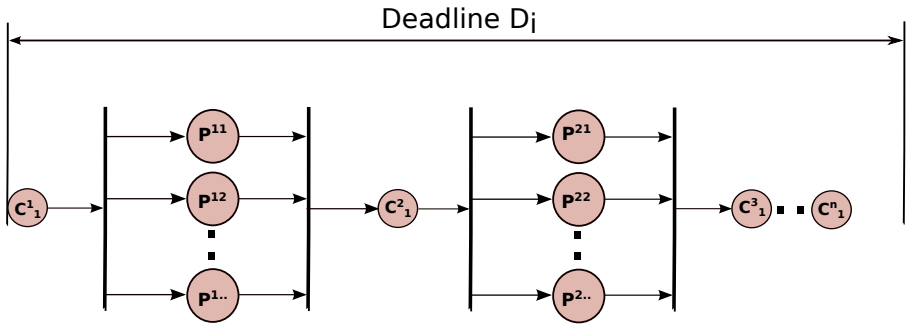


FIGURE 3.4: Fork-join Model

A task starts by executing the master thread and when a fork event happens, the master thread splits into a number of threads that execute in parallel which forms a parallel segment. When all of the threads of a parallel segment terminate their execution, they join back the master thread to resume it execution.

Lakshmanan et al. proposed an divided tasks in two classes :

- **Low utilization tasks** whose utilization is not greater than 1. These tasks are executed sequentially in order to reduce the costs of parallelization.
- **High utilization tasks** whose utilization is greater than 1, this thread must execute in parallel and one processor is not enough to be feasible. The basic idea of the allocation algorithm proposed in the work of Lakshmanan is to fill the slack of by fractions of threads from the parallel segments of the task and try to have the maximum utilization per already loaded cores. The remaining parallel threads are then forced to execute within fixed execution intervals.

Lakshmanan et al. used also a partitioned preemptive Deadline Monotonic algorithm to schedule the constrained-deadline parallel threads resulting. They proved that they can have a resource augmentation bound equal to 3.42.

The scheduling algorithm proposed by Lakshmanan et al. suffers from overheads due to the migration of a thread from a processor to an other (when trying to load at maximum a processor)

Lakshmanan et al. stated in their work that this task model is not convenient from the "perspective of schedulability, it is therefore desirable to avoid such task structures as much as possible", as they stated.

Generalized model

Saifullah et al., 2013 proposed a parallel task model where a task is composed of segments, and each segments consists of a set parallel threads with the same real-time characteristics (e.g. release time, execution time, deadline). At the end of each segment, parallel threads must synchronize their termination. They also proposed a general method to express models proposed in Lakshmanan, Kato, and Rajkumar, 2010; Courbin, Lupu, and Goossens, 2013 in their own. They use global EDF and partitioned DM for scheduling. They assume that their work can be used on uniform processors with different speeds, and they assume that the thread execution time scales inversely on speed. However, they ignore the memory effect on execution time variation.

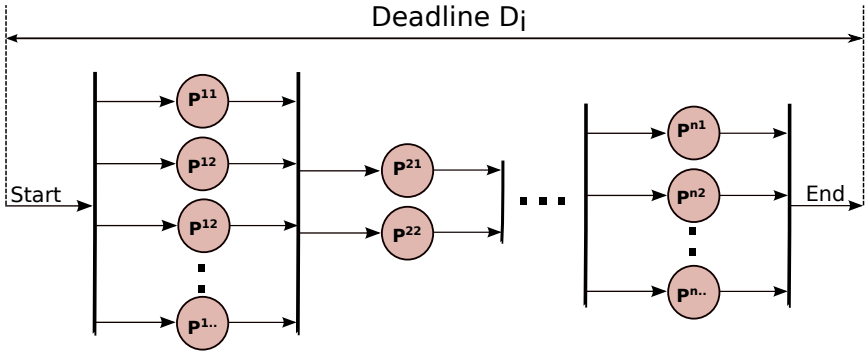


FIGURE 3.5: Generalized Model of Saifullah

Authors simplified the scheduling problem of parallel tasks by using a *decomposition* algorithm which assigns intermediate offsets and deadlines for each segment. These intermediate parameters are used by real-time algorithms to schedule tasks after they are transformed into a set of independent sequential subtasks. They insure the intermediate deadline of a segment insures that the density of any segment can not be greater than $\frac{2 \cdot C_i}{T_i}$. They then divide a task according to certain threshold into heavy and light segments. Hence, in the same task we can find only heavy segments, only light segments or heavy and light segments.

- If all segments are light, then the task deadline is split proportionally among all segment according to the WCET of each segment.
- If all segments are heavy, then task deadline is split to proportionally based on the total execution time of each segment.
- if a task contains a mixture of heavy and light segments. Here, a larger deadline is divided between heavy segments, and a shorter for light segments.

Multi-Phase-Multi-Thread model (MPMT)

Courbin, Lupu, and Goossens, 2013 proposed, multi-phase multi-thread model), a less restrictive model where parallel sub-tasks of each *phase*[1] have the same real-time characteristics except for the execution time. They state that the model proposed in Saifullah et al., 2013 is a specific case of their model. They also proposed an algorithm to assign real-time parameters to adapt fork-join model to their own.

Threads of each phase (equivalent to segment in the previous model) share the same deadline (Figure 3.6) and each thread is a sequential process that requires a single processor. In this work, the authors define two different classes of real-time schedulers for this model which are summarized as follows:

- Hierarchical schedulers manage tasks with a task-level rule and use a second rule to schedule threads within each task (thread-level rule).
- Global thread schedulers use a single scheduling rule to assign priorities to threads regardless of their tasks.

An exact schedulability condition is provided for each class of schedulers. The performance of these schedulers is also compared with Gang scheduling.

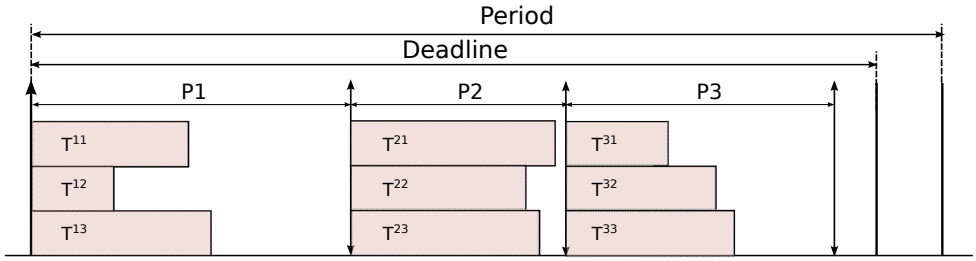


FIGURE 3.6: Multi-Phase Multi-Thread Model

3.4.3 Federated scheduling

Li et al., 2014 proposed a federated scheduling approach for parallel real-time task scheduling. They considered a general task model (DAG to express the data dependency) with implicit deadlines. The federated scheduling algorithm proposes to divide tasks into two disjoint sets, the high-utilization tasks ($u \geq 1$) and the low-utilization tasks ($u < 1$). The low-utilization tasks are run in competition with each other on shared cores using well-know multiprocessor scheduling algorithms. The high-utilization tasks are run each on a dedicated core. This is not always the *optimal* choice in the case of optimizing the energy consumption.

3.5 Related work to energy consumption

Works in (Courbin, Lupu, and Goossens, 2013; Saifullah et al., 2013; Lakshmanan, Kato, and Rajkumar, 2010; Li et al., 2014) only address the scheduling problem of parallel task, without considering the energy consumption and assume a fixed decomposition of a task to a set of parallel threads. However, parallel threads *size* may be adjusted. For example, a parallel **for** loop with 100 iteration may be decomposed into two threads with each 50 iteration, or the first with 80 iteration and the second 20 iteration or any other decomposition that ensures that the sum of iterations is 100. This may allow more flexibility on scheduling. The models that we

[1] A phase is equivalent to a *segment*

will present in this thesis allow having numerous alternative parallel decompositions without any particular restriction.

Regarding the energy consumption problem for parallel tasks, Paolillo et al., 2014 defined the *optimal* frequency for minimizing energy consumption on homogeneous platforms with gang *malleable* tasks. They target homogeneous processor platforms with the ability of turning off some processors. They considered sporadic implicit deadline tasks and use a canonical parallel scheduler proposed in Collette, Cucu, and Goossens, 2008, which is an optimal scheduling algorithm for sporadic tasks on homogeneous multiprocessor platforms.

Colin, Kandhalu, and Rajkumar, 2015 proposed a partitioned-EDF heuristic to allocate implicit deadline tasks on Single-ISA heterogeneous multicore architecture such ARM big.LITTLE architecture, with the goal of minimizing the energy consumption. They show several experiments on EXYNOS 5410,5420,5422 big.LITTLE processors. They assume that the power dissipated by executing a task depends only on the hardware architecture. As we will show later, this assumption is not valid in general, but different tasks dissipate energy in a different way (see Section 5.2.6).

based their work on a similar power dissipation model as the one proposed in Colin, Kandhalu, and Rajkumar, 2015 for an architecture compound of a set of islands. Each island contains a set of cores that share the same frequency characteristics. In contrast to our interests, they focus only on task sequential execution and consider also that the execution time of a task scales in the same way on all island (two cores of two different islands are identical but they may operate at different frequencies). We use the same methodology as Pagani et al. in deriving task allocation and frequency selection.

3.6 Conclusion

In this chapter, we first presented our motivations regarding the CPS needs. We show that it is of a paramount importance to focus on the parallelization under real-time constraints with a special interest in reducing the energy consumption. Secondly we overviewed the different related work that have been published in our interest domain. In the rest of this thesis, we will present three tasks models on three different types of multicore architectures. We will be focus especially on multithreaded models because they are easier to implement and does not need any change on the operating system. All the models we present later can be expressed by an OpenMP-look-like API.

Part II

Contributions

Chapter 4

Free-to-Cut task model On Uniform Computing Architectures

"Le temps est du mouvement sur de l'espace. "

Joseph Joubert

Contents

4.1 Introduction

In parallel computing, many computations are carried out simultaneously by *decomposing* the solution of a large problem into smaller ones which can cooperate to solve at the same time the same problem. In many cases, the computational task is typically broken down in several similar subtasks that can be processed independently and whose results are combined afterwards, upon completion. Practically, the programmer implements such processing by the mean of **for** loops, and APIs such as OpenMP allow to parallelize task code at compilation by interpreting the user compilation directives. However, new aspects that are not considered in sequential programming must be taken into account when parallelizing a task such as: sub-task creation, termination, communication and synchronization These aspects always bring an overhead to the parallel execution of a task compared to the sequential execution of the same task. In this chapter, we ignore the overhead brought by the task parallelization.

4.2 System overview

In this chapter, we consider a set of tasks where each task can be decomposed (*cut*) at any point of its code. For example, a *for* loop with 100 independent iterations can be decomposed into two thread where the 1^{st} thread can handle 20 iterations and the 2^{nd} handles 80 iterations, or it can be decomposed into 3 threads such as the 1^{st} and the 2^{nd} thread can perform 35 iterations for each and the last one only 30 Iterations, or any decomposition to any arbitrary number of threads whose the sum of the performed iterations is equal to 100. However, we do not allow this decomposition to change at run-time, and it is defined at pre-compilation phase. Hence, we are interested only by moldable tasks in this chapter. We focus on allocating such tasks on a compound set of cores with the same characteristics and operate at different fixed frequencies. The aim of the allocation is to reduce the energy consumption, while guaranteeing that all deadlines will be met.

4.3 Architecture Model

In this chapter, we consider an uniform core platform compound of m cores. Every core j has only one operating frequency point f_j. Any core may select one state : active state where it is operating at frequency f_j or *deep power state*. The core state is defined offline and does not change at run-time. We denote by s_j the **speed** of core j and it is computed as the ratio of its operating frequency by the maximum frequency in the platform:

$$\mathsf{s}_j = \frac{\mathsf{f}_j}{\mathsf{f}_{max}} \tag{4.1}$$

Thus, the speed of core j is bounded by :

$$\frac{\mathsf{f}_{min}}{\mathsf{f}_{max}} \leq \mathsf{s}_j \leq 1 \tag{4.2}$$

4.4 Task model

Let $\mathcal{T}_{=}\{\tau_1, \tau_2, \ldots, \tau_n\}$ be a task set. Every task τ_i is characterized by the tuple $\tau_i = (\mathsf{C}_i(1), \mathsf{D}_i, \mathsf{T}_i)$, where:

- $\mathsf{C}_i(1)$ is the worst case execution time of task τ_i on a core operating at the maximum speed. It represents the execution time of the sequential version of the task (single thread). We

assume that the execution time of task τ_i on core j is the ratio of the execution time of the task at speed equal to 1 by the speed of core j:

$$\mathsf{C}_i(\mathsf{s}_j) = \frac{\mathsf{C}_i(1)}{\mathsf{s}_j} \tag{4.3}$$

- D_i is the task relative deadline. We consider tasks with constrained deadline ($\mathsf{D}_i \leq \mathsf{T}_i$);
- T_i: is the task period, it represents a fixed time between releases of two consecutive instances of task τ_i.

In this chapter, we assume that a task can be decomposed into several parallel threads at any point of its code. For example, a task with an execution time $\mathsf{C}_i(1) = 11$, can be decomposed into three parallel threads with an execution time that equals 3, 3 and 5, or any arbitrary decomposition to any number of threads that verifies that the sum of the execution times of the parallel threads is equal to the execution time of the sequential $\mathsf{C}_i(1)$ (Equation (4.4)), we call this model "Free-to-Cut" (FTC) task model . In this model, parallel threads are independent from each other whereas threads belonging to the same task and running on different cores need to synchronize their activation and deadlines. Notice that the model, we consider in this chapter, does not take into account the overheads due parallelization (thread creation, synchronization, $\cdots$). We refer by thread τ_i the thread responsible for sequential execution of task τ_i.

Definition 1. *Let τ_i be a task. We define $\mathsf{Th}_{i,j}$ as the thread of task τ_i that is allocated on the core j. We define also $\mathsf{C}_{i,j}(\mathsf{s}_j)$ as the worst case execution time of thread $\mathsf{Th}_{i,j}$ on core j.*

Definition 2. *Decomposition $\mathcal{D}_{\tau_i} = \{\mathsf{Th}_{i,j}, i \in \{1, \cdots, n\}, j \in \{0, \cdots, m\}\}$ is a correct decomposition of the task τ_i according to FTC model, if and only if:*

$$\sum_{j}^{m} \mathsf{C}_{i,j}(\mathsf{s}_j) \cdot \mathsf{s}_j = \mathsf{C}_i(\mathsf{s}_{max}) \tag{4.4}$$

Notice here, that only one thread of task τ_i can be allocated on a core. Thus, a task can never be decomposed to a number of thread that exceeds m the number of cores.

4.5 Power & Energy Models

As stated in Chapter 1, total energy is the integral of the sum of static and dynamic power. In this chapter, we consider cores with one operating frequency point, thus the dissipated dynamic power does not change at run-time and we assume that cores in deep power state does not dissipate power.

4.5.1 Power Model

Mei et al., 2013 defined the dynamic power dissipation by a CMOS circuit as the product of a constant coefficient ξ that depends on the technology used to manufacture the chip, by the square of the voltage, and by the frequency.

$$P = \xi \times V^2 \times \mathsf{f} \tag{4.5}$$

They also defined the frequency (f) as the ratio of the difference between the actual voltage V and V_{Th} raised to the power of a, where V_{th} is the threshold voltage, by the product of a constant K and the logic depth L_d. a and K are constants that depend on ξ.

$$\mathsf{f} = \frac{(V - V_{\mathsf{Th}})^a}{K \times L_d} \tag{4.6}$$

By combining Equation (4.5) and Equation (4.6), the dynamic power P can be expressed as a polynomial of frequency of degree λ, where λ has been set equal to 3 in most of the papers focusing on energy consumption.

In this chapter, We compute the dissipated dynamic power by core j as:

$$P(j) = \mathsf{Coef}\ \mathsf{f}_j^3 \tag{4.7}$$

We also assume that static power is equal to a constant values Const for active cores and 0 for cores in deep sleep state.

4.5.2 Energy Model

To simplify the calculation of the integral of power consumption, we assume that the energy consumption of core j is computed as the sum of the product of the dynamic of the power dissipation of the core j by the execution time of all threads allocated and the product of the constant amount of static energy by the state of the core by the system life time.

$$E(j) = (\sum_{i=0}^{n} \mathsf{Coef} \cdot \mathsf{f}_j^3 \cdot \frac{\mathsf{C}_{ij}}{\mathsf{T}_i}) + \mathsf{Const} \cdot h \cdot \mathsf{state}(j) \tag{4.8}$$

Where:

- Coef: is the power consumption coefficient.
- Const: is the constant static power dissipated when the core is in active state.
- h: the system life time, here we consider it as the hyper period, computed as the least common multiple of all periods.
- state(j): is the core state, this parameter is equal to 0 if the core is in deep sleep state and 1 in the case of active state.

4.6 Allocation and Scheduling

After having defined task, architecture and energy models, we proceed to task allocation in this section. Firstly, we formulate the problem of allocating a set of tasks to m uniform cores as Linear Programming (LP) problem. Further, we propose a heuristic to solve efficiently the allocation problem in a very short time.

We assume a partitioned Earliest Deadline First (EDF) scheduling. Each core has its own single-processor EDF scheduler and a separate ready-queue. Therefore, the scheduling analysis can be performed with well- know techniques for single-processor scheduling. The analysis here are based on the demand function analysis introduced in Chapter 2.

4.6.1 Exact Scheduler

We formulate the FTC allocation problem as Linear Program (LP). We denote by $\mathcal{T}_j$ the threads allocated on core j and by $\mathsf{dbf}(\mathcal{T}_j, t)$ the demand bound function of the set of threads of $\mathcal{T}_j$ for a time of length$t$.

Let $\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_n\}$ be a task set of n sporadic tasks to allocate to m uniform cores.

First, we define variable x_{ij} as the percentage of the execution time C_i that will be allocated to core j. Thus, thread $\mathsf{Th}_{i,j}$ has an execution time that equals:

$$\mathsf{C}_{ij} = x_{ij} \cdot \mathsf{C}_i \tag{4.9}$$

Hence, x_{ij} is defined : $[0, 1]$.

$$\begin{cases} x \in [0,1] \\ i \in 1..n \\ j \in 1..m \end{cases} \tag{4.10}$$

The objective function (Equation (4.11) is set to minimize the energy consumption on all cores, thus it combines the energy consumption (Equation (4.8)) for all threads on the same core.

$$\text{minimize } E = \sum_{j=0}^{m}((\sum_{i=0}^{n} \mathsf{Coef} \cdot \mathsf{f}_j^3 \cdot x_{ij} \cdot \frac{\mathsf{C}_i}{\mathsf{T}_i}) + \mathsf{Const} \cdot h \cdot \mathsf{state}(j)) \tag{4.11}$$

The best solution that minimizes the energy consumption must respect the following constraints:

$$\sum_{j=1}^{m} x_{ij} = 1, \qquad \forall i \in \{0, \cdots, n\} \tag{4.12}$$

$$\mathsf{dbf}(\mathcal{T}_j, t) \leq t \qquad \forall t \in [0-h], \forall j \in \{0, \cdots, m\} \tag{4.13}$$

where :

$$\mathsf{dbf}(\mathcal{T}_j, t) = \sum_{i=0}^{|\mathcal{T}_j|} \left\lfloor \frac{t + \mathsf{T}_i - \mathsf{D}_i}{\mathsf{T}_i} \right\rfloor \mathsf{C}(\mathsf{s}_j) \cdot x_{ij}$$

Equation (4.12) insures that all the parts of the task are allocated to a core. Equation (4.13) grants that all allocated threads are schedulable according to the demand bound function based analysis for uniprocessor.

There are many NLP solvers, both freely available as open source software, and commercial ones. We selected LP_SOLVE solver, an open source solver designed to solve efficiently linear programs.

4.6.2 FTC Heuristic

In this section, we describe our allocation heuristic. Our approach uses an ordered task queue, a compound set of tasks that are not yet allocated. The heuristic selects at each iteration a task and a core and aims to allocate the selected task to the selected core if it is possible. If it is not possible than it tries to allocate a part of the task on the current core. If this second step fails, then the heuristic seek to allocate the task on the next core. If all cores are investigated and a task could not be allocated, the schedule is aborted. Thus, the heuristic goes two steps:

- **1st Step**: The queue is ordered according to some criteria (for example, decreasing utilization, or rate monotonic, etc.). As we will show further, the results do not depend too much on how the queue is ordered. Because the dynamic power depends mainly on the operating frequency of a core, it is more convenient to allocate the maximum load to the cores with the minimal speed first because they consume less energy. Hence, cores are also sorted in a non-increasing order by speed.

- **2nd Step**: The second step consists in performing the schedulability analysis. We select the task at the top of the task queue let it be τ_i and the lowest indexed core let it be j. Let $\mathcal{T}_j$ denote the set of threads already allocated on core j, we assume that $\mathcal{T}_j$ is feasible on core j. The goal of this step is to define the maximum portion of the execution time of task τ_i that can be allocated on core j, such that the system remains feasible. The maximum execution time of task τ_i corresponds to the single thread version of the task (since all execution times of the parallel threads are less than the sequential version of the task).

	D	T	$\mathcal{C}$
Th_1	7	10	2
Th_2	9	15	3
τ_3	5	6	4

TABLE 4.1: Example of excess-time evaluation

According to the computed portion of execution time, the task is *cut* into two threads: the 1^{st} is allocated on core j and the 2^{nd} is put back to the task queue to be allocated in the next iterations. We call this portion of execution time excess-time

Excess-time computation

Here, we describe how excess-time is computed. We start from the assumption that $\mathcal{T}_j$ is feasible on core j, thus:

$$\forall t \in [0 - t^*], \quad \mathsf{dbf}(\mathcal{T}_j, t) \leq t \tag{4.14}$$

To be feasible on core j, the set $\mathcal{T}_j \cup \{\tau_i\}$ must verify the following condition:

$$\forall t \in [0 - t^*], \quad \mathsf{dbf}(\mathcal{T}_j \cup \{\tau_i\}, t) \leq t \tag{4.15}$$

Lemma 1. *Let $\mathcal{T}_j$ be a set of threads feasible under EDF on core j, and τ_i be a sporadic thread not belonging to $\mathcal{T}_j$. Then, $\exists \mathsf{C}', 0 \leq \mathsf{C}' \leq \mathsf{C}_i(\mathsf{s}_j)$ such that the following condition is always verified.*

$$\forall t \in [0 - t^*], \mathsf{dbf}(\mathcal{T}_j, t) + \left\lfloor \frac{t + \mathsf{T}_i - \mathsf{D}_i}{\mathsf{T}_i} \right\rfloor (\mathsf{C}_i(\mathsf{s}_j - \mathsf{C}') \leq t \tag{4.16}$$

Proof. If we select $\mathsf{C}' = \mathsf{C}_i$ the second term is zero, and since $\mathcal{T}_j$ is schedulable by hypothesis, the lemma is verified. □

Let us suppose that $\mathcal{T}_j \cup \{\tau_i\}$ is not feasible on core j, for at least for one value of t Condition (4.15) is not verified for t. Hence, We estimate the largest portion of execution time of task τ_i that can be allocated on core j, by computing C' as the minimum value such that Condition (4.16) is verified.

Theorem 1. *Let $\mathcal{T}_j$ be a set of threads feasible under EDF on core j, τ_i be a task not belonging to $\mathcal{T}_j$, and let us assume that $\mathcal{T}_j \cup \{\tau_i\}$ is not feasible on core j.*

$\mathcal{T}_j \cup \mathsf{Th}$ is feasible if Th has the same period and deadline as τ_i and an execution time equals to $\mathsf{C}_i - \mathsf{C}'$ where:

$$\mathsf{C}' = \max_{t \in \{0 \cdots t^*\}} \left(\frac{t - \mathsf{dbf}(\mathcal{T}_j \cup \{\tau_i\}, t)}{\left\lfloor \frac{t + \mathsf{T}_i - \mathsf{D}_i}{\mathsf{T}_i} \right\rfloor} \right) \tag{4.17}$$

Proof. It is sufficient to invert Equation (4.16) but considering the largest portion possible, thus $\mathsf{dbf} = t$. □

In the rest of this work C' is called excess-time and computed as Equation (4.17).

Example Here we present an example of excess-time evaluation. Threads Th_1 and Th_2 are feasible on core p and we try to allocate task τ_3 on the same core as $\mathsf{Th}_1, \mathsf{Th}_2$ (thread details are shown in the table below).

The dbf of threads $\mathsf{Th}_1, \mathsf{Th}_2$ is shown in blue in Figure 4.1, whereas for threads $\mathsf{Th}_1, \mathsf{Th}_2, \mathsf{Th}_3$ it is presented in red. In interval $[0, 30]$, we notice 5 points where $\mathsf{dbf}(\{th_1, th_2, th_3\}, t)$ is greater than t. These points and the corresponding excess-time are reported in the table below (Table 4.2):

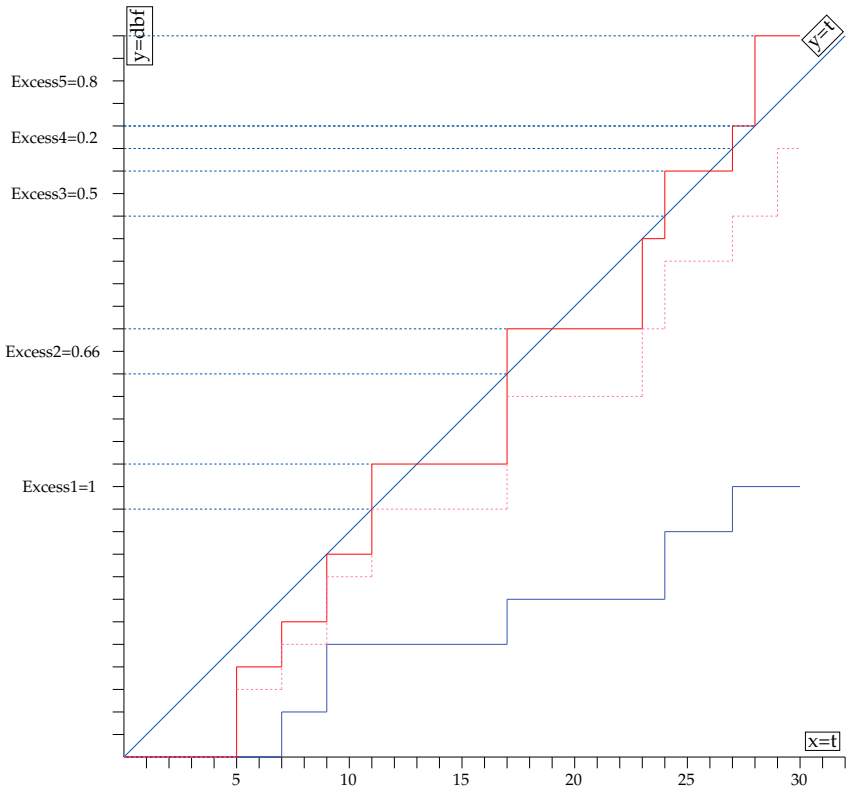


FIGURE 4.1: An example of excess − time evaluation

t	excess-time
11	1
17	0.66
24	0.5
27	0.2
28	0.8

TABLE 4.2: Excess-time values

Thus, only the first value is kept because it corresponds to the greatest value of excess-time. Notice that the system is schedulable if the execution time of the new incoming thread is reduced by excess-time (dashed red line).

Excess-time evaluation is described in Algorithm 1. The theoretical complexity of this algorithm is $\theta(h \cdot n)$

The excess-time is used in to choose the portion of the task to allocate to current core.

Allocation

Now that we have introduced the concept of excess-time, we use a simple heuristic to select cut the task and assign one thread. According to the value of the excess-time and execution time of the task only 3 cases need to be checked:

1. excess-time $= 0$, it means that the single thread version of the task and the thread-set already allocated on core j is feasible. Thus, single thread version of the task is allocated on the current core (the task is run in sequential way). Our approach favours the allocation of the single thread version of a task first because it does not suffer from any parallelization costs. This choice has been made because the parallel version in the next chapters will suffer from parallelization costs while sequential one does not.

Algorithm 1 excess-time Evaluation

Input: Taskset:$\mathcal{T}_j$, Task :τ_i , s_j
Output: excess-time = 0
$t = 0$
while $(t \leq h)$ **do**
  $\mathsf{dbf}^* = \mathsf{dbf}(\mathcal{T}_j, t) + \left\lfloor \frac{t+(\mathsf{T}_i-\mathsf{D}_i)}{\mathsf{T}_i} \right\rfloor \mathcal{C}(\mathsf{s}_j)$
  if $(\mathsf{dbf}^* > t)$ **then**
    $\text{temp-excess-time} = \left(\frac{t-\mathsf{dbf}(\mathcal{T}_j \cup \{\tau_i\}, t)}{\left\lfloor \frac{t+\mathsf{T}_i-\mathsf{D}_i}{\mathsf{T}_i} \right\rfloor} \right)$
    if (Temp-excess-time $>$ excess-time) **then**
      excess-time = Temp-excess-time
    end if
  end if
  t=t+1
end while

2. excess-time is equal to execution time of the single thread version of the task (excess-time $= \mathsf{C}i(\mathsf{s}_j)$). This means that the single thread version of the task is fully in excess on current core j. In other words, any thread having the same period and deadline with any execution time can not be allocated to core j. Thus, we seek to allocate threads on the next core $j+1$.

3. excess-time is greater than zero, and less than the execution time of single thread version of the task ($0 <$ excess-time $< \mathsf{C}i(\mathsf{s}_j)$). The single thread version of the task is not schedulable on current core j, but a thread that has the same period and deadline as task τ_i and an execution time equals to $(\mathsf{C}i(\mathsf{s}_j) -$ excess-time$)$ can be allocated on the current core. Thus, the task is split into two threads that have the same period and deadline as task τ_i: the first have an execution time that equals to $\mathsf{C}i(\mathsf{s}_j) -$ excess-time and is allocated to the current core. The second thread has an execution time that equals to excess-time and is put back to the task queue to be allocated in the next iterations.

The algorithm 2 combines different steps of our heuristic. The theoretical complexity of this algorithm is $\theta(n \log n \cdot h)$

Example: In this example, we try to allocate task set $\mathcal{T} = \{\tau_1, \tau_2\}$ to 3 cores of speed 1. The task characteristics are reported in the following table:

Task	C(1)	D	T
τ_1	9	6	7
τ_2	10	8	9

TABLE 4.3: Example of FTC scheduling: task set details

Here the different iterations to allocate τ_1, τ_2 to the cores $1, 2$ and 3:

1. The first selected task is τ_1 and the first selected core is core 1, then the excess-time is evaluated for this task, on this core. The evaluation result is that excess-time $= 3$, this value is greater than 0, and less than the task execution time 9. Thus, task is decomposed, according to the 3^{rd} case, into two threads: the first thread Th_{11} is allocated on core 1 with the characteristics $\mathsf{Th}_{11}(\mathsf{C}(1) = 6, \mathsf{D} = 6, \mathsf{T} = 7)$; the second thread $\mathsf{Th}_{12}(\mathsf{C}(1) = 3, \mathsf{D} = 6, \mathsf{T} = 7)$ is put back into the task queue to be allocated in the next iteration.

Algorithm 2 Free To cut model partitioning

```
OrderTasks()
sortCoresBySpeed()
for τ_i ∈ AllTasks do
    for ∀j&τ_i notAllocated do                                   ▷ loop2
        excess-time = evaluate_excess-time();
        if ( excess-time = 0 ) then
            Allocate τ_i to j
            break loop2
        else
            if ( C_i(s_j) < excess-time ) then
                Th_1=( C_i(s_j) − excess-time, D_i, T_i);
                Th_2=( excess-time × s_j, D_i, T_i);
                Allocate Th_1 to j
                put_back Th_2 to task_queue
                break loop2
            else
                SeekOnTheNextCore
            end if
        end if
    end for
end for
```

2. The selected task now is Th_{12} and the selected core is core 2. The excess-time is evaluated for this couple and it is equal to 0, thus the thread is allocated to core 2, according to the 1^{st} case.

3. The selected task in this iteration is task τ_2 and the selected core is core 1. The excess-time is evaluated for this task and core and it is equal to 9. As in the first iteration, the excess-time is greater than 0 and less than the task sequential execution time. Thus, the task is decomposed into two threads: the first thread Th_{21} is allocated to core 1 with the characteristics $\mathsf{Th}_{21}(\mathsf{C}(1) = 1, \mathsf{D} = 8, \mathsf{T} = 9)$; the second thread $\mathsf{Th}_{22}(\mathsf{C}(1) = 9, \mathsf{D} = 8, \mathsf{T} = 9)$ is put back into the task queue to be allocated in the next iteration.

4. We have only one task in the task queue, Th_{22} and the selected core is 2. In this case, the excess-time is equal to 4. As in the previous iteration, thread Th_{22} is decomposed into two threads: the first thread Th_{221} is allocated on core 1 with the characteristics $\mathsf{Th}_{221}(\mathsf{C}(1) = 5, \mathsf{D} = 8, \mathsf{T} = 9)$; the second thread $\mathsf{Th}_{222}(\mathsf{C}(1) = 4, \mathsf{D} = 8, \mathsf{T} = 9)$ is put back into the task queue to be allocated in the next iteration.

5. We have again only one task in the task queue, Th_{222} and the selected core is core 3. The excess-time is evaluated and it is equal to 0. Thus, thread Th_{222} is allocated to this core.

The task queue is now empty thus, and all tasks were allocated. Thus the allocation has been successful. The results of allocation are described in Table 4.4 :

Core	Threads
1	$\mathsf{Th}_{11}(\mathsf{C}(1) = 6, \mathsf{D} = 6, \mathsf{T} = 7)$ & $\mathsf{Th}_{21}(\mathsf{C}(1) = 1, \mathsf{D} = 8, \mathsf{T} = 9)$
2	$\mathsf{Th}_{12}(\mathsf{C}(1) = 3, \mathsf{D} = 6, \mathsf{T} = 7)$ & $\mathsf{Th}_{221}(\mathsf{C}(1) = 5, \mathsf{D} = 8, \mathsf{T} = 9)$
3	$\mathsf{Th}_{222}(\mathsf{C}(1) = 4, \mathsf{D} = 8, \mathsf{T} = 9)$

TABLE 4.4: The results of task allocation

Algorithm 3 UUniFast-Discard

```
Input: TotalU : Double,nbTask : Integer
Output: U⃗ : Double
nextSumU : Double;
discard : boolean;
while discard or utilization_is_valid do
    sumU = TotalU
    discard = false;
    for (i = 0; i < nbTask − 1; i++) do
        nextSumU = sumU * Math.pow(random(0,1), 1/ (nbTask - (i + 1)));
        U[i] = sumU - nextSumU;
        sumU = nextSumU;
        if (U[i] > 1) then
            discard = true;
        end if
    end for
    util[nbTask - 1] = sumU;
    if (util[nbTask - 1] > 1) then
        discard = true;
    end if
end while
```

4.7 Experimentation

In this section, we apply our heuristic on a large number of randomly generated synthetic task sets to evaluate the performances of our heuristic. For the experiments, we allocate the generated task sets on a set of 4 cores with speed $0.5, 0.5, 1$ and 1.

4.7.1 Task Generation

The UUniFast method, proposed by Bini and Buttazzo, 2005, can be used to generate a number of utilization factors, each one bounded by 1 and whose sum is a given number bounded by 1. The method has been extended by Davis and Burns, 2009 to multicore systems with the UUniFast-discard algorithm. The latter allows setting the bound to a number $m \geq 1$, and it adds the constraint that each utilization must not exceed 1. Algorithm 3 describes the UUnifast-Discard. Utilization_is_valid. is a simple method that checks if the sum of the generated utilization is less than the number of cores.

Goossens and Macq, 2001 have shown that the *hyperperiod* grows exponentially with the greatest value of period. They also proposed a method to bound the *hyperperiod* and generate periods T_i. We use their method for generating task periods.

Therefore, we compute the computation time of the single thread for little and for big cores $\mathsf{C}_i(1) = u(i) \times \mathsf{T}_i$.

Finally, the deadline is generated in the interval $[P_r \times \mathsf{T}_i, \mathsf{T}_i]$ where P_r is randomly generated between $[0.75, 1]$.

4.7.2 Simulations

Our algorithm has no direct competitors, in the sense that classic partitioning heuristics do not take into account intra-task parallelism or task decomposition. Therefore, we have no choice but to compare our heuristics to classic bin-packing heuristics, such as Best Fit (*BF*), Worst Fit (*WF*) and First Fit (*FF*) and the exact scheduler solved by LP_SOLVE.

The BF, FF, WF were implemented as they are stated in the literature without any parallelization features and without any modification to be adapted to uniform platforms. Total utilization is varied from 0.5 to 3.0. For each utilization, we generate 100 different task sets.

4.7.3 Results & discussions

Figure 4.2 shows schedulability rate as function of total utilization for the 100 task sets using the bin-packing allocation heuristics: BF, WF, FF, the FTC heuristic and the exact solution. Notice, in the figure, that our heuristic "FTC" outperforms hugely all the bin-packing heuristics, and keeps the schedulability rate equal to the exact solution until a total utilization of 2.5. In contrast to the results reported in the literature of partitioned scheduling for identical core platforms, BF is worse than WF and FF because it tries to load the most free processor first and then the most loaded processor first and in this case it loads processors with high speed first (those with speed is equal to 1). Thus, the schedulability chances are reduced because the task set is sorted by task utilization and the core with speed equals to 0.5 has less chance to accept the allocation of *heavy* tasks (heavy tasks are tasks with high utilization). Even at the maximum possible total utilization, our heuristic FTC is able to schedule more than 35% of task sets while the bin-packing heuristic can not schedule any task set, but FTC is still less than the exact scheduler which is able to find feasible allocation for more than 55% of task sets.

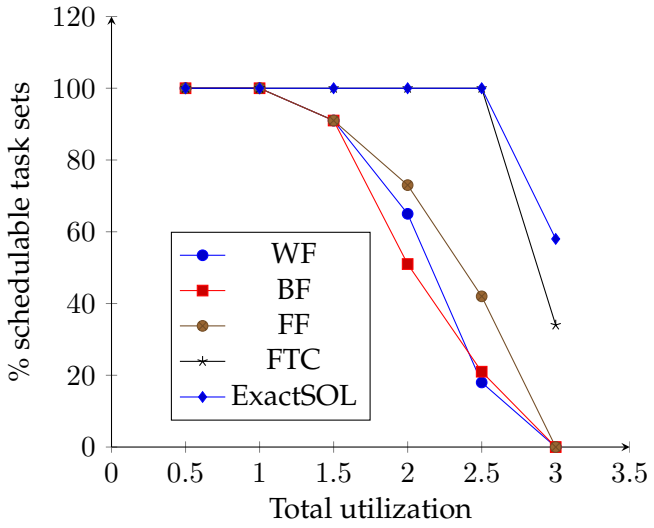


FIGURE 4.2: Schedulability rate for BF, WF, FF, FTC and Exact Scheduler

Our heuristic outperforms the bin-packing heuristics also in term of processor usage. As you can notice in Figure 4.3, it reports the average utilization per every core for all heuristics and exact scheduler. Our heuristic FTC and exact solution loads the processors as full as it is possible (core utilization is close to 1). Hence, FTC and exact scheduler allow to use less cores than bin-packing heuristic. But, for low total utilization (less than 2), you can notice that BF uses less cores than both exact scheduler and FTC because it uses high speed cores first. Notice that there is only a slight difference between FTC and the exact solution in terms of processor usage.

Figure 4.4 reports the energy consumption as a function of total utilization for all heuristics and the exact scheduler. FTC heuristic outperforms all bin-packing heuristics, and it consumes a slightly more energy than the exact scheduler. Notice here that FF bin packing heuristic outperforms BF and FF because it load more the first cores which are the core operating at a low speed, hence they consume less energy.

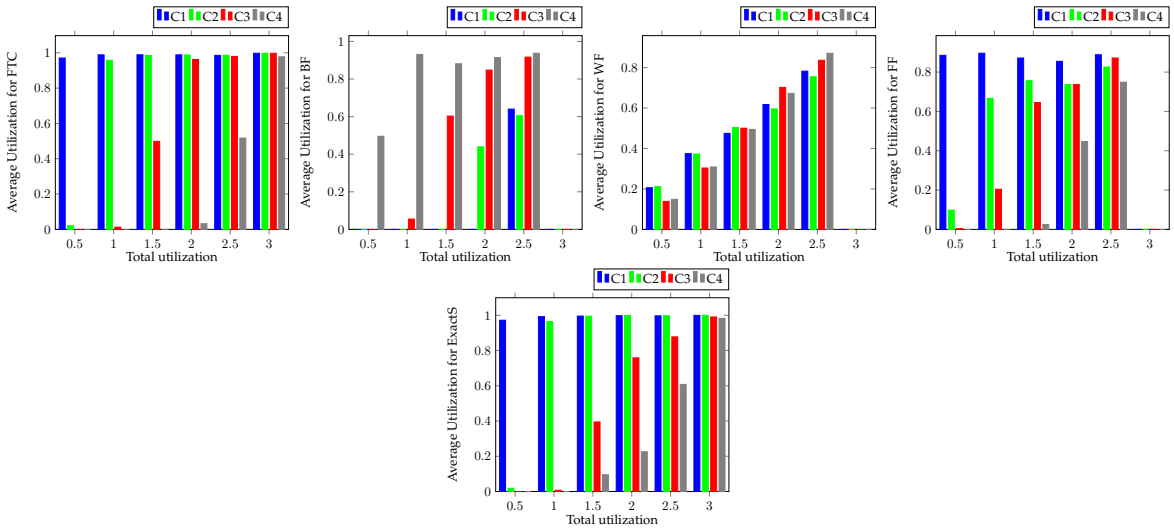


FIGURE 4.3: Average Utilization per cores for BF, WF, FF, FTC and the exact scheduler

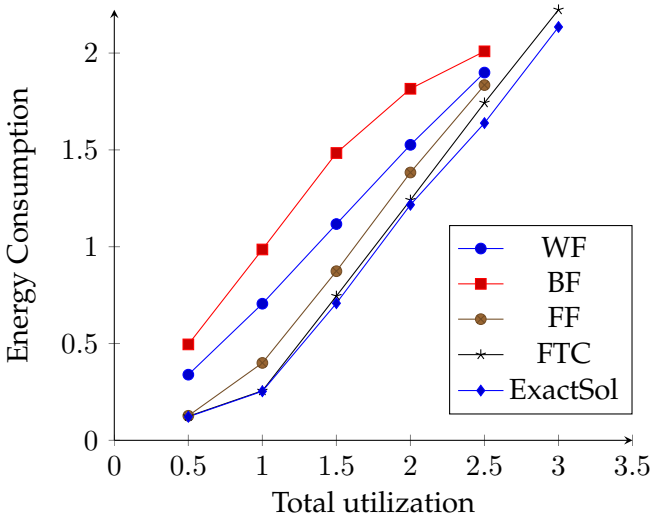


FIGURE 4.4: Energy consumption for different heuristics

4.8 Conclusion

In this chapter, we presented a parallel task model that does not take into account the parallelization costs and we proposed two methods to solve the problem of decomposing and allocating a set of n tasks of this model to a set of m uniform cores where each core has its own operating frequency point and it can not be changed. In the first method, we modeled the problem as a linear program and it was solved using LP_Solve, an open source solver. In the second method, we proposed a heuristic that allows to decompose the execution of a task into several parallel thread based on the dbf computation. We conducted several experiment for classic non-parallel bin packing heuristics and the proposed heuristic "FTC" and the exact scheduler. The results had shown that our heuristic outperforms all bin-packing heuristics and that the obtained decomposition and allocation is very close to the optimal one. In the next chapter, we propose a more realistic task model on a more realistic hardware architecture.

Chapter 5

Minimal frequencies for allocating Cut-Point Tasks model to Uniform & Heterogeneous Multicore Architectures

“ Le temps est du mouvement sur de l’espace. ”

Joseph Joubert

Contents

5.1 Introduction

In this chapter, we address the problem of executing task and data parallel (soft) real-time applications on heterogeneous computing platforms with the goal of reducing the energy consumption. We extend the model of task, architecture and energy proposed in the previous chapter to more realistic models. Here we focus on Dynamic Voltage and Frequency Scaling (DVFS), parallelization, real-time scheduling and resource allocation techniques. In the first part of the chapter, we present a model of the performance and energy consumption of a parallel real-time task executed on an ARM bigLITTLE architecture. We use this model in the second part of the chapter where we first define the optimization problem as an Integer Non-linear Programming (INLP) problem, and then propose heuristics for efficiently solving it.

In order to achieve an optimal decomposition with respect to the energy consumption for a set of tasks on heterogeneous multicore platform, we need to (i) set the operating frequency of cores; (ii) decompose each task into a set of parallel threads (if possible/desirable); (iii) perform a schedulability analysis and allocate the threads onto the cores to guarantee that each thread completes before its deadline.

5.2 System Model

We focus on scheduling moldable real-time tasks on heterogeneous multicore platforms. In such platforms, cores may have different characteristics (e.g. architecture, pipelines, memories, etc). These differences have an impact on the behaviour of a task (its execution time and energy consumption) when it is allocated on different cores. Thus, it is important to construct a realistic model of the task execution times, of the architecture and of the energy consumption.

5.2.1 Experimental platform

We use the *ODROID XU3* [1] Board as experimentation platform. The *ODROID XU3* board is compound of a *Samsung Exynos 5422*, a *Mali GPU*, a RAM memory and I/O peripherals.

The *Samsung Exynos 5422* is an ARM big.LITTLE multicore architecture. It consists of 8 cores: 4 big cores (*ARM Cortex A15*) and 4 little cores (*ARM Cortex A7*). The *ODROID XU3* board embeds 4 *INA231 current-shunt and power sensors*[2] that allow measuring the instant current and power dissipation of big cores, little cores, the GPU and the RAM memory. The *INA231* sensors have a high accuracy with a maximum error that reaches 0.5% (as per constructor specification). Besides the embedded sensors, an external energy sensor (*ODROID Smart-Power*[3]) is plugged to the power supply of the board to measure the overall power dissipation.

Each core of the Exynos 5422 has 2×32 Ko of L1-cache (data and instruction cache). Little cores share 512 Ko of L2-cache. Big cores share 2 Mo of L2-cache. Both big and little cores share 2 Go of RAM memory. Unlike the *Samsung Exynos 5420*, big and little cores of *Samsung Exynos 5422* can be *active* at the same time.

The frequency of little cores can be calibrated homogeneously for all little cores at the same time from 200Mhz to 1400Mhz by discrete steps of 100Mhz (13 modes). Similarly, the frequency of big cores can be calibrated from 200Mhz to 2000Mhz by discrete steps of 100Mhz (19 modes). Any core (big or little) can be set in low power state (*Standby power state*) [4]. The core enters the

[1] HardKernel board, link: http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127&tab_idx=2

[2] INA231 sensors are manufactured by Texas Instruments, Link: http://www.ti.com/lit/ds/symlink/ina231.pdf

[3] ODROID Smart Power of Hard Kernel, Link: http://odroid.com/dokuwiki/doku.php?id=en:odroidsmartpower

[4] ARM documentation: http://infocenter.arm.com/help/topic/com.arm.doc.den0022c/DEN0022C_Power_State_Coordination_Interface.pdf

standby power state by executing a *WFI*[5] or *WFE*[6] instructions and exits on a corresponding wake up event.

5.2.2 Architecture Model

According to the characteristics of many heterogeneous multicore architectures, we introduce the concept of *core group*. We model an architecture of m cores as a set of G groups. Each group $\mathcal{G}_g$ is compound of a set of cores that have the same frequency characteristics (minimal (f^g_{min}), maximal (f^g_{max}), operating frequency (f^g_{op})). Each group has a set of discrete frequencies (modes) and its own energy calculation characteristics (discussed in Section 5.2.6). Cores are indexed alternatively, so Group g of Core j is denoted with index $(j \mod G)$. This representation (Equation (5.1)) allows us to address homogeneous architectures like SMPs by setting the number of groups g to 1 and the ARM big.LITTLE architecture by setting the number of groups to 2. For example, the *Exynos 5422* of the *ODROID XU3* board is modelled with $g = 2$, where little cores are indexed 0,2,4,6 and big cores are indexed 1,3,5,7.

$$\begin{aligned}\mathcal{A} =&\{\mathcal{G}_g,\ g \in \{1 \dots G\}\}\\ \mathcal{G}_g =&(\{\mathsf{P}_j, (j \mod G) = g, j < m\},\\ &\mathsf{f}^g_{min}, \mathsf{f}^g_{max}, \mathsf{f}^g_{op}), g \in \{1 \dots G\}\end{aligned} \tag{5.1}$$

We denote m_g as the number of cores of group g.

In order to build task and energy models, we benchmark the *Exynos 5422 Samsung* processor with 6 different profiling tasks: matrix multiplication (MATMUL), Fast Fourier Transform (FFT), Quick sort (QS), Shortest graph path (Dijkistra), Basic math operation (BM), and susan-c. The last 4 benchmarks have been adapted from the corresponding MiBench benchmark suite (Guthaus et al., 2001). The tasks' code is written in C using the POSIX real-time thread library. The implemented threads are periodic and have real-time priorities. We measure the execution time by the mean of the POSIX **clock_gettime** system call and the power dissipation by the mean of the embedded sensors on big cores, little cores and RAM memory.

5.2.3 Model of the execution time

Seth et al., 2006 reported that a part of the task execution time does not depend on the core frequency (e.g. due to the central memory access). Bini, Buttazzo, and Lipari, 2005 proposed a similar approach for the task timing model where timing model has been modeled as the sum of semi-linear function of frequency and a constant that represents the memory access. Hence, we assume that the execution time of a thread depends on:

- the operating frequency of the group (f^g_{op}) where it is allocated (*assumption 1*),
- the microarchitecture (g) of this group (*assumption 2*),
- the thread itself. A part of the thread execution time depends on the core operating frequency, we denote it as $\mathsf{ct}^g_{\mathsf{Th}}(\mathsf{f})$. The other component, that we denote as ($\mathsf{mt}^g_{\mathsf{Th}}$), does not depend on the operating frequency of the core (e.g. due to the central memory access)(*assumption 3*).

To verify these assumptions, we conducted several experiments. At each experiment, we allocated the 6 benchmarking thread(s) onto the core(s) operating at a fixed frequency. In each scenario we vary the number of threads, the core(s) on which the thread(s) is(are) allocated, the operating frequency of this(these) core(s).

[5] Wait For Interrupt
[6] Wait For Event

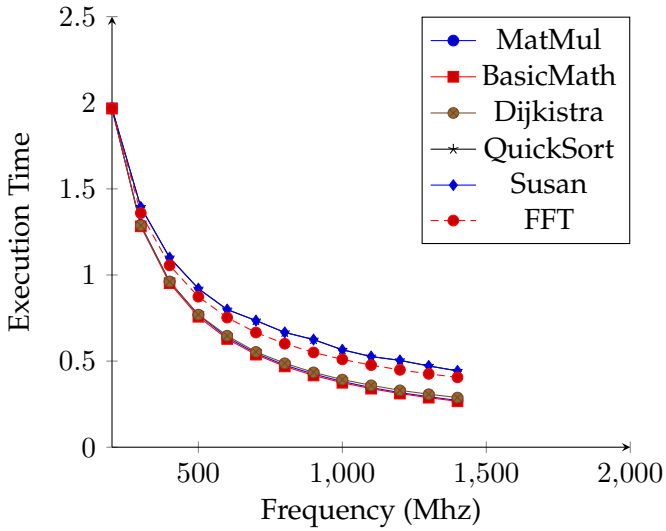


FIGURE 5.1: The execution time of the 6 benchmarks when allocated on one little/big Core

In Figure 5.1, we present the execution of one thread of each benchmark when it is executed on one little core as a function of frequency. The execution time of each thread has been calibrated so that their execution time at 200Mhz is approximately the same.

We observe that the profile of execution time as a function of frequency is very similar for the 6 benchmarks. However, as you can notice, there is a *invariant difference* between the execution time of the benchmarks. It can be explained by the different profile of the memory access pattern of each benchmark. To investigate this assumption, we conducted several experiments that are presented in the rest of this section.

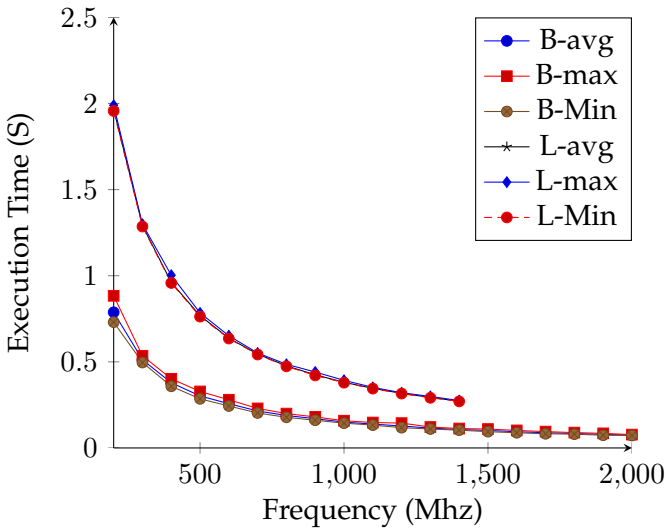


FIGURE 5.2: The execution time of square matrix multiplication (200x200) thread allocated on one little/big Core

Figure 5.2 shows the average execution time of 500 executions of one MATMUL thread allocated on one big core (B-avg) and on one little core (L-avg) as a function of frequency. The figure shows also the maximum and the minimum execution time on big cores (B-max, B-min) and on little cores (L-max, L-min). The average value changes in an interval of 2% on little cores and 3 % on big cores. Obviously, the execution time varies with the frequency in an inversely proportional manner (assumption 1 checked). We observe that the execution time of the thread allocated on a big core is approximately ($\times 3$) times shorter than the execution time of same

thread allocated on a little core operating on the same frequency. This is due to the larger and more complex pipeline of big cores compared to little cores. The ARM cortex A15 pipeline is an Out-Of-Order pipeline and contains complex branch predictors and powerful computation elements; whereas the ARM cortex A7 pipeline is a smaller and an In-order-pipeline. Thus, ARM Cortex A15 executes in average more instructions by one cycle than ARM Cortex A7 (assumption 2 is checked).

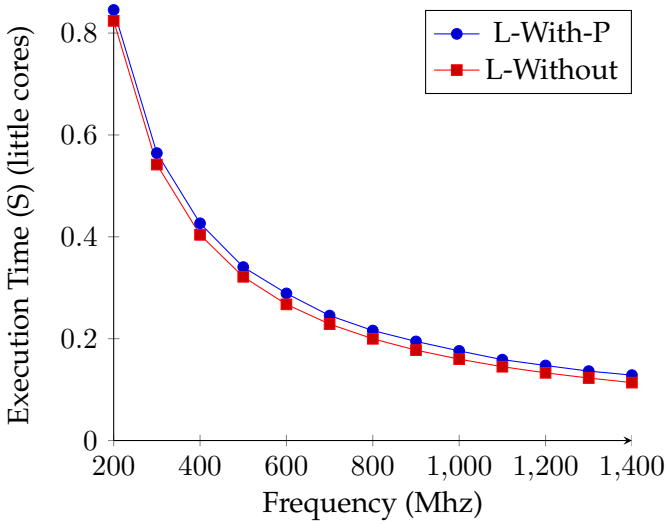


FIGURE 5.3: Execution time of the MATMUL (150x150) thread with and without interfering thread

Figure (5.3) shows the results of two different experiments on little cores. In the first, we allocate one MATMUL thread on one core (L-without is the average execution time of this thread as a function of frequency). In the second, we allocate the same thread as in the first experiment on a core and an interfering thread on another little core (L-with is the average execution time of the same thread as experiment 1 as function of frequency). The thread of the second experiment takes more time to execute than the same thread in the first experiment, even if it has no competition on the core where it is allocated. Thus, we assume that the difference in the execution time between both experiments is due to the interference on cache and memory access in the second experiment. To estimate this difference, we apply a non-linear regression on the execution time of both experiments. The difference is *almost* constant and is estimated to 0.015 sec. The same experiments were conducted on big cores, and similar observations can be done, but the difference is estimated to 0.007 sec. We assume that the difference is smaller on big cores because the L2-cache of big cores is larger than the L2-cache of little cores, so less cache-misses occur. Thus, the variation of the execution time of a thread as a function of frequency can be estimated as the sum of a linear function of the speed (inverse of frequency) and a constant value that represents the memory access time (Const_2 in Equation (5.2)).

$$\mathcal{C}_{\mathsf{Th}}(\mathsf{f}^g_{op}) = \frac{\text{Const}_1}{\mathsf{f}^g_{op}} + \text{Const}_2 \tag{5.2}$$

In order to confirm this assumption about Const_2, we changed the size of the matrix (100x100, 150x150, 200x200, 250x250, 300x300) in the MATMUL and we computed Const_2 and the resident set size (RSS[7]) for each case. Results are presented in Figure (5.4).

Figure (5.4a) shows the RSS as a function of the matrix size. Figure (5.4b) shows the values of Const_2 for the big core (Const_2-B) and for the little core (Const_2-L) as a function of the RSS: as you can see, RSS and the Const_2 are directly proportional. Thus, Const_2 depends mainly on the memory demands. In the rest of this chapter Const_2 is renamed as mt.

[7]RSS: is the portion of memory occupied by a process that is held in main memory (RAM)

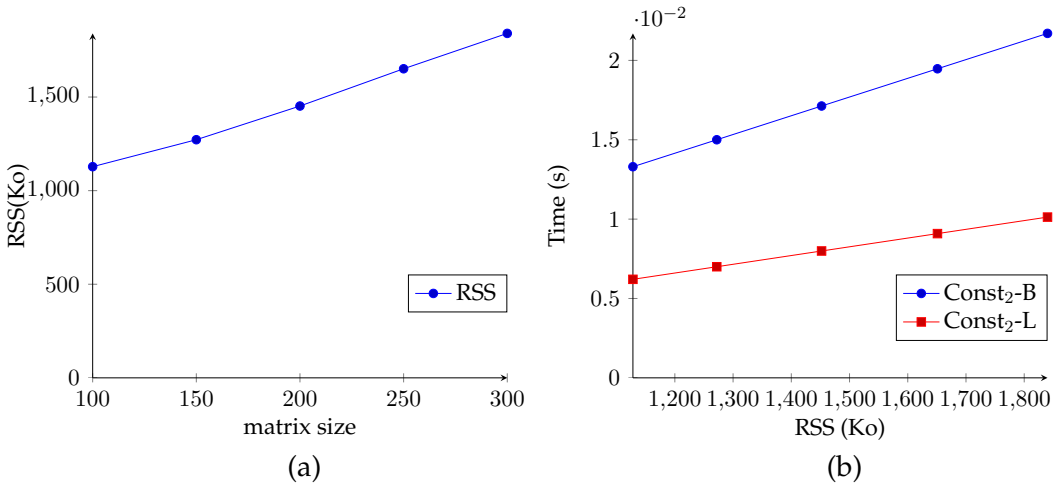


FIGURE 5.4: The computed mt as a function of RSS

The execution time of any thread $\mathcal{C}_{th}$ on group g operating at frequency f^g_{op} is the sum of a linear function of speed and the memory access time (mt) (Equation (5.3)).

$$\mathcal{C}^g_{\mathsf{Th}}(\mathsf{f}^g_{op}) = \frac{\mathsf{ct}^g_{\mathsf{Th}}(\mathsf{f}^g_{max})\mathsf{f}_{max}}{\mathsf{f}^g_{op}} + \mathsf{mt}^g_{\mathsf{Th}}. \tag{5.3}$$

Similar results has been obtained on an Intel i3 processor. The *i3-3217U* processor is compound of 2 physical cores hosting two logical cores by hyper-threading. The processor has 3 cache levels: L1, L2 and L3. The L1 cache has 2 x 32 KB of cache instruction and 2 x 32 KB for data cache. L2 cache has a size of 2 x 256 KB and L3 cache has a size of 3 MB. The frequency of each physical core can be set from 800 MHz to 1800 MHz by discrete steps of 100Mhz.

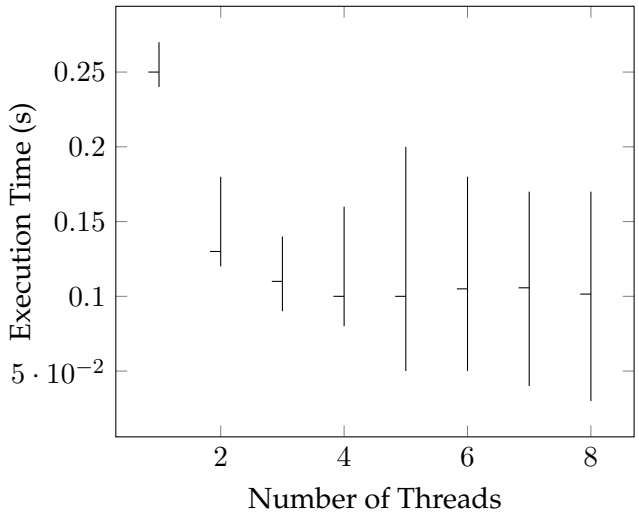


FIGURE 5.5: The execution time of matrix multiplication under several decompositions

In the experiment of Figure 5.5, the matrix multiplication task was decomposed into 2,3,4,5,6,7 and 8 threads with the same load (all threads do a similar size of data) and the operating frequency is set to 1800Mhz. The figure reports the maximum, the minimum and the average execution time for all decompositions. As You can notice, increasing the number of threads leads to a shorter execution time. But as expected, when the number of threads is greater than the number of available cores (4), the execution time can not be improved (shorter), even more the configuration with 8 threads has a bit longer execution time than the configuration with 6 threads. Moreover, we can notice that the execution time when the task is decomposed into

two threads is a little bit more than the execution time of the single thread version of the task divided by 2. The slight difference is due to the parallelization costs.

To estimate the parallelization costs, we can refer to the metrics stated in Chapter 1.

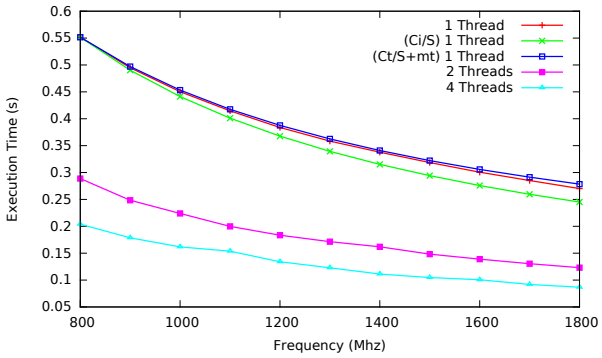


FIGURE 5.6: The execution time of different task decompositions at different frequencies

Figure 5.6 reports the results of three experiments: in the first we allocate the matrix multiplication (size of matrix is 200 x 200) thread on one core, in the second we decompose the matrix multiplication task into two threads with the same load and we allocate each one on a different core, in the last experiment we decompose the task into 4 threads each one allocated on a different core. Figure 5.6 reports the average execution time of the three experiments as a function of frequency. The figure shows also two semi-linear functions of frequency: the first one is

$$\mathsf{C}(\mathsf{f}) = \frac{\mathsf{C}(800Mhz) \cdot 800}{\mathsf{f}} \tag{5.4}$$

This timing model in the most used one in literature for uniform architectures and it is plotted in green. The second one is

$$\mathsf{C}(\mathsf{f}) = \frac{\mathsf{C}(800Mhz) \cdot 800}{\mathsf{f}} + 0.4 \tag{5.5}$$

and it is plotted in blue. Notice that these two functions are plotted only for the single thread version of the task. The first function underestimates the execution time of the task as a function of frequency. Hence the results of Equation (5.3) is quite general and representible for different hardware architectures.

5.2.4 Parallel moldable tasks

In this chapter, we consider sporadic moldable real-time tasks. Let $\mathcal{T}_{=}\{\tau_1, \tau_2, \ldots, \tau_n\}$ be a set of sporadic moldable independent synchronous tasks. Every task τ_i is characterized by the tuple $\tau_i = (\mathsf{D}_i, \mathsf{T}_i, \vec{\xi_{ig}}, \Gamma_i)$, where:

- D_i is the task relative deadline. We consider constrained deadline tasks ($\mathsf{D}_i \leq \mathsf{T}_i$).
- T_i: is the task period, it is the time between the releases of two consecutive instances of task τ_i.
- $\vec{\xi_{ig}}$ is the vector of power dissipation coefficients. It will be discussed in Section 5.2.6.
- Γ_i is a set of *cut-points*, $\Gamma_i = \{\gamma_{i,1}, \ldots, \gamma_{i,k_i}\}$. Each cut-point $\gamma_{i,k}$ represents one possible parallel decomposition of the task τ_i into parallel threads ($\mathsf{Th}_{i,k,1}, \mathsf{Th}_{i,k,2}, \ldots$). We denote as $|\gamma_{i,k}|$ the number of threads in cut-point $|\gamma_{i,k}|$ and we denote $\mathsf{Th}_{i,k,z}$ as the thread z of cut-point $\gamma_{i,k}$ from the task τ_i. Each thread $\mathsf{Th}_{i,k,z}$ has the same period and the same relative deadline of the task τ_i to which it belongs. We assume that the first cut-point $\gamma_{i,1}$

cut-point	Threads	ct^b	mt^b	ct^l	mt^l
$\gamma_{1,1}$	$Th_{1,1,1}$	64	9	250	37
$\gamma_{1,2}$	$Th_{1,2,1}$	35	7	139	29
	$Th_{1,2,2}$	35	7	139	29
$\gamma_{1,3}$	$Th_{1,3,1}$	50	8	207	33
	$Th_{1,3,3}$	17	5	49	15

TABLE 5.1: Cut-points list example

contains one thread representing sequential execution of task τ_i (single thread version), thus $\gamma_{i,1} = \{Th_{i,1,1}\}$.

For example, let τ_1 be a sporadic moldable task that implements MATMUL with size 200x200. Let us suppose that the task is modelled with 3 different cut-points as follows:

$$\tau_1 = (D_1 = 150, T_1 = 200, \Gamma_1 = \{\gamma_{1,1}, \gamma_{1,2}, \gamma_{1,3}\})$$

Γ_1 details are described in the following table (Table 5.1):

Task τ_1 can be run in a sequential way (single thread version) using the first cut-point $\gamma_{1,1}$. The execution time of τ_1 on a big core operating at the maximum frequency is equal to $\mathcal{C}^b_{1,1,1}(f_{max}) = [ct^b_{1,1,1}(f^b_{max}) + mt^b_{1,1,1}] = 64 + 9 = 73$ ms. If τ_1 is run on a little core operating at the maximum frequency and using the same cut-point, the execution time is equal to $\mathcal{C}^l_{1,1,1}(f_{max}) = [ct^l_{1,1,1}(f^l_{max}) + mt^l_{1,1,1}] = 250 + 37 = 287$ms.

The same task can be decomposed into 2 threads for a parallel execution using cut-points $\gamma_{1,2}, \gamma_{1,3}$. Since running a task in parallel brings an overhead for creating and synchronizing threads, the parallel execution *suffers* from an overhead, computed as the difference between the sum of execution time of parallel threads and the execution time of the single thread version of the task. For example using $\gamma_{1,2}$, if threads are allocated on big cores, the overhead is $(35 + 7 + 35 + 7) - (64 + 9) = 11$ ms. The overhead is hard to evaluate when threads of the same task are allocated on different groups, but it still exists.

Notice that we allow different values of ct and mt among parallel threads belonging to the same cut-point and we allow many cut-points with the same number of threads. Thus, this model is quite general and can be used to represent alternative decompositions for the same level of parallelism.

5.2.5 Power model

In this chapter, we calculate off-line an operating frequency for each group and the power state for every core, and we fix both for the whole system lifetime. Therefore, in this chapter we are not interested in measuring the needed time and power to change the frequency mode or the core power state.

In Figure (5.7), we measured the memory power dissipation of MATMUL as a function of frequency. The L2-cache of little cores is smaller than the L2-cache of big cores. Thus, the same thread demands more memory accesses when allocated on a little core than when it is on a big core, and this leads to a higher power dissipation. The power dissipation varies very little with core frequency, and again this small variation can be explained by the different number of memory accesses per second which depends on core frequency and on the data access pattern of the task. If we restrict to little cores, memory power dissipation represents [10-17] % of total power dissipation in the case of MATMUL, and [16-25] % in the case the FFT (not shown in the figure).

Figure (5.8) shows the power dissipation of one core where one MATMUL thread is allocated as a function of frequency. B-avg represents the average power dissipation for big cores

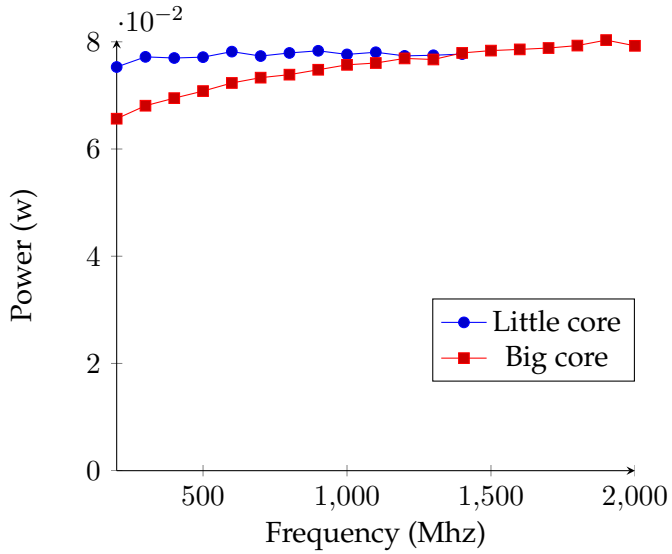


FIGURE 5.7: The memory power dissipation by one little and big core

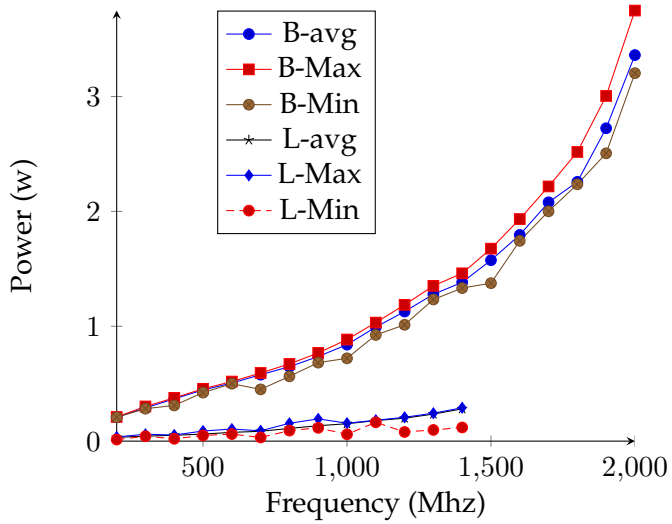


FIGURE 5.8: Power dissipation of matrix multiplication on big and little cores

(L-avg for little cores), B-min represents the minimal sensed power for big cores (L-min for little cores), B-max presents the maximal one for big cores (L-max for little cores). The timing acceleration obtained by using big core instead of little cores has an impact, as big cores consume more power than little cores. Even when operating at the highest frequency, little cores dissipate less power than big cores operating at the lowest frequency.

Figure 5.9 reports the average power dissipation of one core where one thread of the 6 benchmarks is allocated as a function of frequency. Each benchmark has its own energy consumption profile. Also, the variation of the frequency is small for lower frequencies (between 200, 600) and bigger for higher frequencies. This is due to the fact that power is the product of current and voltage, and that the voltage is constant for a subset of the lower frequencies.

Figure (5.10) shows the results of 6 experiments, 3 on little cores and the same 3 experiments on big cores. In the first, we allocate one thread on one core, in the second two threads on two different cores, and three threads on three different cores in the last one. The figure presents the average power dissipation as a function of frequency. Using more cores implies consuming more power. However, power dissipation is not multiplied by two and three because cores share the same *common* power, and this power is not replicated when using more than one core.

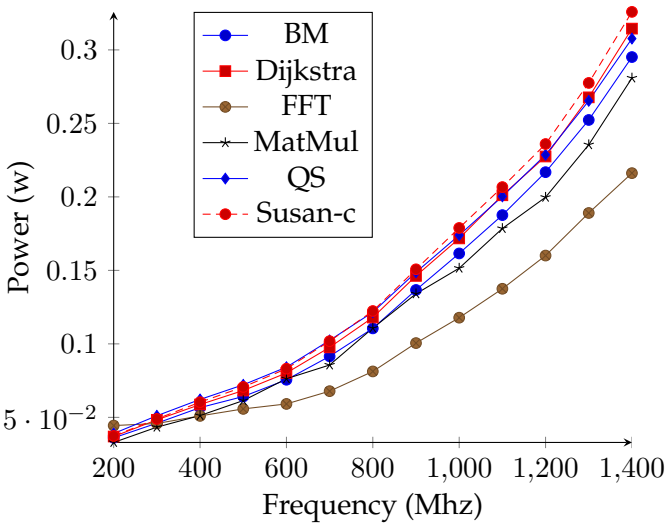


FIGURE 5.9: Power dissipation for different processing

Figure (5.10) reports also the results of a polynomial regression of the third degree on the real values for MATMUL and FFT. The maximum error on the regression is 2%.

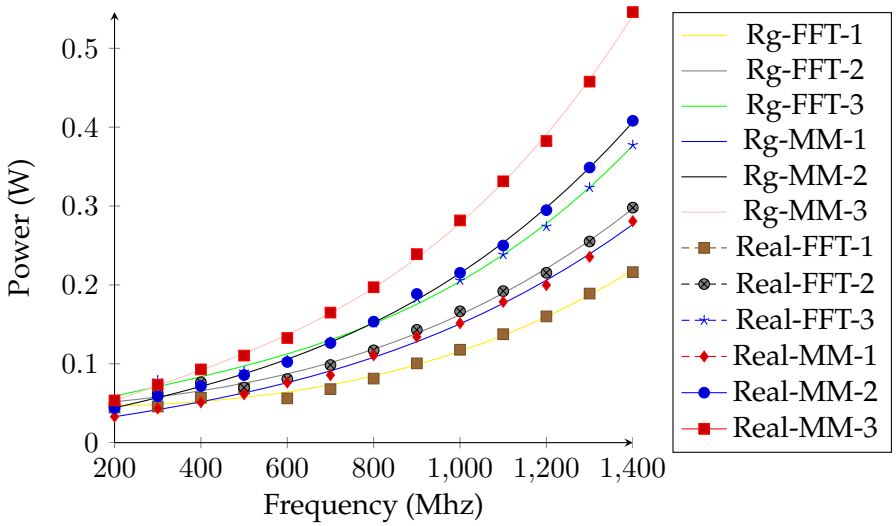


FIGURE 5.10: Real-values and regressions of power dissipation of little cores of matrix multiplication and Fourier transformations

The 6 benchmarks dissipate different amounts of power because they activate different components of the processor cores and those different components may consume different amounts of power. The polynomials resulting from the regressions on one little core for all benchmarks are reported in the table below (Table 5.2).

From all these experiments, we conclude that the power dissipation can be modelled as a third degree polynomial of frequency (Equation (5.6)), whose coefficients are different for each application. These coefficients are dependent on the task and on the core group. The vector of coefficients is denoted by $\vec{\xi_{ig}}$:

$$P_{\mathsf{Th},g}(\mathsf{f}) = \vec{\xi_{ig}} \cdot \vec{F^3}, \tag{5.6}$$

where $\vec{F^3}$ denotes the vector of powers of the frequency, $\{\mathsf{f}^3, \mathsf{f}^2, \mathsf{f}^1, \mathsf{f}^0\}$. In this chapter, we assume that threads of the same task dissipate the same amount of power, thus they have the same coefficients.

task	Coef f^3	Coef f^2	Coef f^1	Coef f^0
FFT	$4.609 \cdot 10^{-11}$	$2.193 \cdot 10^{-8}$	$3.410 \cdot 10^{-8}$	$4.433 \cdot 10^{-2}$
MATMUL	$5.220 \cdot 10^{-11}$	$4.053 \cdot 10^{-9}$	$7.763 \cdot 10^{-5}$	$1.675 \cdot 10^{-2}$
BasicMath	$3.475 \cdot 10^{-11}$	$6.500 \cdot 10^{-8}$	$3.016 \cdot 10^{-5}$	$2.883 \cdot 10^{-2}$
QuickSort	$3.427 \cdot 10^{-11}$	$5.659 \cdot 10^{-8}$	$5.261 \cdot 10^{-5}$	$2.755 \cdot 10^{-2}$
Susan-c	$4.833 \cdot 10^{-11}$	$4.245 \cdot 10^{-8}$	$5.934 \cdot 10^{-5}$	$2.460 \cdot 10^{-2}$
Dijkstra	$4.136 \cdot 10^{-11}$	$5.519 \cdot 10^{-8}$	$4.542 \cdot 10^{-5}$	$2.682 \cdot 10^{-2}$

TABLE 5.2: The power dissipation coefficients for the 6 benchmarks

Even if threads use different hardware components, there is a basic amount of power that is dissipated as long as the core is turned on and the frequency is fixed, even when the idle process runs in the OS. We call this *absolute static power* (P^g_{stat}).

Finally, the power consumed by a set of threads allocated on core group g is expressed as the sum of every thread power dissipation and the absolute static power. Equation (5.7) will be used in the rest of this work to compute the power dissipation:

$$P^g(\mathsf{f}) = \sum P_{\mathsf{Th},g}(\mathsf{f}^g_{op}) + P^g_{stat} \tag{5.7}$$

5.2.6 Energy model

The energy consumption is the integral of the power dissipated by a thread over the time when it executes. It can be approximated as the product of the average power dissipation of thread Th and its execution time $\mathcal{C}_{\mathsf{Th}}$:

$$E^g(\mathsf{Th}, \mathsf{f}) = \mathcal{C}_{\mathsf{Th}} \cdot P_{\mathsf{Th},g}(\mathsf{f}) \tag{5.8}$$

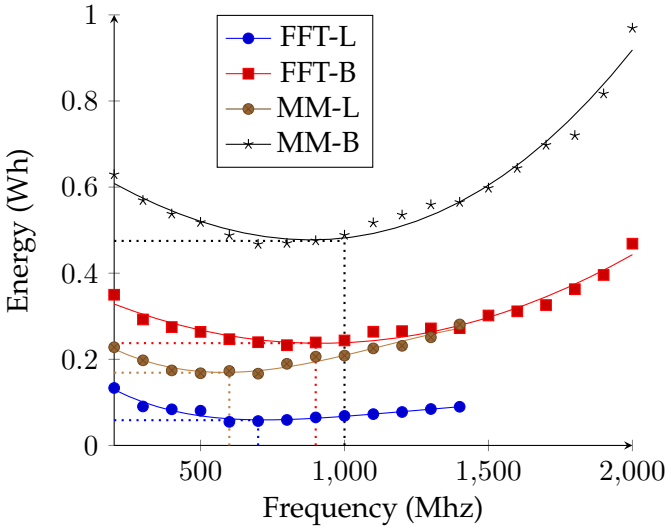


FIGURE 5.11: Energy consumption matrix multiplication and Fourier transformations threads allocated on little and big cores

Figure 5.11 shows the energy consumption of MATMUL and FFT on little and big cores. MATMUL and FFT consume different amounts of energy, however they still vary in the same way as a function of frequency. Starting from 200Mhz, both are lower when the frequency is higher until a certain frequency (400 For Susan, 500Mhz for MATMUL and QuickSort and Dijkstra, 600Mhz for FFT and Basic Math). After this minimum, increasing the frequency involves

more energy consumption. In the figure we do not take into account the timing constraints: obviously, not all frequencies allow to complete the task within its deadline.

We call the frequency that corresponds to the least amount of energy per each thread (regardless its timing constraints) *effective frequency* and it is denoted by $\mathsf{f_{eff}}$. The effective frequency is the one which cancels the derivative of energy, and it can be easily found, for example by using a dichotomy search in a few iterations.

After having described architecture, task and energy models, we address the problem of scheduling.

5.3 Allocation & Scheduling

In the case of heterogeneous platforms, it is hard to handle the migration of a thread from a group to another because for each group we may have different timing analysis. Thus, in this work, we consider partitioned Earliest Deadline First (EDF) scheduling. Each core has its own separate ready-queue and single-processor EDF scheduler known to be optimal for this purpose. Therefore, the scheduling analysis can be performed with well-known techniques for single-processor scheduling. We assume that all tasks are independent of each other, whereas threads belonging to the same tasks and running on different cores need to synchronize their activation times and deadlines. The scheduling analysis is based on demand-analysis proposed by Baruah, Rosier, and Howell, 1990.

Thread $\mathsf{Th}_{i,k,z}$ allocated on core j is denoted by ($\mathsf{aTH}_{i,j,k,z}$), and let $\mathcal{T}_j$ denote the set of threads allocated on core j: $\mathcal{T}_j = \{\mathsf{aTH}_{i,j,k,z}, i \in \{1 \dots n\}, k \in \{1 \dots K_i\}, z \in \{1 \dots |\gamma_{i,k}|\}\}$. Let t be a non-negative integer: the demand bound function $\mathsf{dbf}(\mathcal{T}_j, t)$ denotes the maximum cumulative execution requirement that can be generated by jobs of $\mathcal{T}_j$ that have both ready times and deadlines within any time interval of length t:

$$\mathsf{dbf}(\mathcal{T}_j, t) = \sum_{\mathsf{aTH} \in \mathcal{T}_j} \left\lfloor \frac{t + \mathsf{T}_i - \mathsf{D}_i}{\mathsf{T}_i} \right\rfloor \mathcal{C}^g_{i,k,z}(\mathsf{f}^g_{op}) \tag{5.9}$$

Theorem 2 (Baruah 1990 Baruah, Rosier, and Howell, 1990)**.** *Task set $\mathcal{T}_j$ is feasible if and only if the following condition is verified for all values of t:*

$$\forall t \leq t^*, \mathsf{dbf}(\mathcal{T}_j, t) \leq t$$

where t^* is a constant that depends on the utilization of the task set (see Baruah, Rosier, and Howell, 1990 for more details on the analysis algorithm). In this work, we consider t^* equal to the *hyperperiod* which is computed as the least common multiple of task periods.

First, we will start by presenting the problem formulation as an Integer Non-Linear Programming problem (INLP), then, we will present our heuristics for frequency selection and thread allocation.

5.3.1 Optimal schedulers

We formulate the problem of scheduling a task set $\mathcal{T}_=\{\tau_1, \tau_2, \dots, \tau_n\}$ on multicore architecture $\mathcal{A}$ as an Integer Non-Linear Programming (INLP) problem. The problem of task allocation without decomposition is already a NP complete problem, by adding the decomposition and the frequency selection, the problem becomes harder and non-linear.
We define the binary variable $x_{i,j,k,z}$ as:

$$\begin{cases} 1 & \text{if thread } \mathsf{Th}_{i,k,z} \text{ is allocated on core } j \\ 0 & \text{otherwise} \end{cases}$$

We define frequency as a discrete variable:

$$\mathsf{f}^g_{op} \in \{\mathsf{f}^g_{min}, \cdots, \mathsf{f}^g_{max}\}$$

Given these definitions, we now present one formulation of the problem as INLP problem. The objective function can be expressed as follows:

$$\min E = \sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{K}\sum_{z=1}^{m} h\ \vec{\xi_{ig}} \cdot \vec{F^3} \left(\frac{\mathcal{C}^g_{i,k,z}(\mathsf{f}^g_{op})}{\mathsf{T}_i} \right) x_{i,j,k,z} + \tag{5.10}$$
$$\sum_{g}^{G} h\ P^{stat}_g \left\lceil \frac{(\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{K}\sum_{z=1}^{m} x_{i,j,k,z})}{\frac{m}{G}} \right\rceil$$
$$(where : j \in \{1 \cdots m\}, j \mod G = g)$$

The objective function is the sum of two terms. The first term expresses the energy that depends on thread execution. The second term expresses the common energy consumption that is generated by at least one thread when allocated on group g (otherwise it is turned off). The ceiling in the second term is equal to 1 if at least one thread is allocated on core group g, and it is equal to 0 otherwise.

The constraints of the problem:

$$\sum_{j=1}^{m}\sum_{k=1}^{K}\sum_{z=1}^{m} x_{i,j,k,z} = m, \quad \forall i \tag{5.11}$$

$$\sum_{i=1}^{n} \mathsf{dbf}(\mathcal{T}_j, t)\ x_{i,j,k,z} \leq t, \quad \forall j, t \in [0 - h] \tag{5.12}$$

$$x_{i,j,k,z} + x_{i,j',k',z'} \leq 1, \quad \forall (i,k,z,j,z',j',k' > k) \tag{5.13}$$

$$\sum_{j=1}^{m} x_{i,j,k,z} \leq 1, \quad \forall i,k,z \tag{5.14}$$

In order to formulate our problem as INLP, the maximum number of threads per every cut-point must be defined. Observe that it is not necessary to have a number of threads per cut-point greater than the number of cores, otherwise some threads will be forced to be allocated on the same core and it would increase pessimism of our analysis because each thread utilization is inflated by a cost of parallelization. Thus z is upper bounded by m. As you will notice in the next section, to simplify the heuristic, our allocation approach may indeed allocate threads of the same cut-point on the same core.

To grant the respect of the schedulability of every task set $\mathcal{T}_j$ at each time $t, t \in [0, h]$, the demand bound function must verify Condition (5.12). Constraint (5.11) imposes that all threads of the same cut-point must be allocated. A correct solution of the problem must verify that all threads of the same task belong to the same *cut-point*: we express this as a conflict constraint (Constraint (5.13)). Constraint (5.13) is an optimized version of the following constraint:

$$x_{i,j,k,z} + x_{i',j',k',z'} \leq 1, \forall i,j,k,z,i',j',k',z', k \neq k' \tag{5.15}$$

if any $x_{i,j,k,z}$ is equal to 1, it excludes all other cut points, because any $x_{i',j',k',z'}$ where k is different of k' is equal to 0 (because the sum is equal to 1 and the first one is already less or equal to 1). This constraint is not generated for thread of the same cut point, because k is different from k', thus only one cut-point can be selected. As addition operation "+" is commutative,

and we want to reduce the number of constraints of this kind (which is very huge), we do not to generate duplicated constraints, like $x_{1,2,1,1} + x_{1,1,2,1} \leq 1$ and generate $x_{1,1,2,1} + x_{1,2,1,1} \leq 1$, by imposing that k to be greater than k'. Thus, the selected cut-point is unique.

Finally, we must verify that each thread is allocated on one core only (Constraint (5.14)).

There are many NLP solvers, both freely available as open source software, and commercial ones. We used Knitro Solver[8] Byrd, Nocedal, and Waltz, 2006, a commercial solver expressly designed for solving efficiently this kind of problems. Unfortunately the problem at hand is a very complicated combinatorial problem. We run the solver on a small example with 3 tasks with 5 cut-points per each task, to be executed on a ARM big/Little with 8 cores, having 13 frequency modes for little cores and 19 for big cores. The optimization took 12 hours before resolution. So we had been limited to very small sizes of the problem, which are not presented. The problem complexity can be reduced by using dbf approximations as in **baruah2016** but the produced results are not optimal. We believe that it is impractical to use a NLP solver for this kind of problems. Thus, in this chapter we propose heuristics to obtain quasi-optimal solutions in a reasonable time.

5.3.2 Scheduling heuristics

We propose a greedy algorithm for selecting the frequencies and the cut-points, and allocating the threads on the different available cores, so to minimize total energy consumption while guaranteeing that all deadlines are respected.

Let $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ be a set of n tasks. We start by defining the *baseline utilization* for each task on a group as $u_i^g(\mathsf{f}_{op}^g) = \frac{\mathcal{C}_{i,1,1}^g(\mathsf{f}_{op}^g)}{\mathsf{T}_i}$, where $\mathcal{C}_{i,1,1}^g(\mathsf{f}_{op}^g)$ is the execution time of the single thread cut-point, considering group g operating on frequency f_{op}^g. Then $U^g(\mathsf{f}_{op}^g) = \sum_i u_i^g(\mathsf{f}_{op}^g)$ is the total baseline utilization of the task set on group g.

Lemma 2. *A necessary condition for task set $\mathcal{T}$ to be feasible on m_g identical cores operating on frequency f_{op}^g with m_g^a active cores is the following:*

$$U^g(\mathsf{f}_{op}^g) \leq \frac{\mathsf{f}_{op}^g}{\mathsf{f}_{max}^g} m_g^a \tag{5.16}$$

Proof. The proof is based on the necessary condition for a task set on uni-processor architecture Baruah and Fisher, 2005: the total utilization of a task set must not exceed 1. However, in the original theorem, there is no notion of frequency scaling: $\sum u_i \leq 1$. This version can be adapted to uni-processor with variable frequencies by adding the frequency as parameter to compute the u_i such as:

$$\sum u_i(\mathsf{f}_{op}) \leq \frac{\mathsf{f}_{op}}{\mathsf{f}_{max}}$$

[8] Knitro Solver web page: http://www.artelys.com/fr/optimization-tools/knitro

This equation must be verified for all active cores:

$$\sum_{\tau_i \in \mathcal{T}_1} u_i(\mathsf{f}_{op}) \leq \frac{\mathsf{f}_{op}}{\mathsf{f}_{max}}$$
$$\cdots \qquad \cdots$$
$$\sum_{\tau_i \in \mathcal{T}_{m_g^a}} u_i(\mathsf{f}_{op}) \leq \frac{\mathsf{f}_{op}}{\mathsf{f}_{max}}$$

By doing the sum of left and right sides

$$\sum_{\tau_i \in \mathcal{T}_1} u_i(\mathsf{f}_{op}) + \cdots + \sum_{\tau_i \in \mathcal{T}_{m_g^a}} u_i(\mathsf{f}_{op}) \leq \frac{\mathsf{f}_{op}}{\mathsf{f}_{max}} + \cdots + \frac{\mathsf{f}_{op}}{\mathsf{f}_{max}}$$

Where

$$\sum_{\tau_i \in \mathcal{T}_1} u_i(\mathsf{f}_{op}) + \cdots + \sum_{\tau_i \in \mathcal{T}_{m_g^a}} u_i(\mathsf{f}_{op}) = U^g(\mathsf{f}^g_{op})$$
$$\frac{\mathsf{f}_{op}}{\mathsf{f}_{max}} + \cdots + \frac{\mathsf{f}_{op}}{\mathsf{f}_{max}} = \frac{\mathsf{f}^g_{op}}{\mathsf{f}^g_{max}} m_g^a$$

□

We call $(\frac{\mathsf{f}^g_{op}}{\mathsf{f}^g_{max}} m_g^a)$ the *current strength* ($\mathcal{S}^g$) of group g:

$$\mathcal{S}^g = \frac{\mathsf{f}^g_{op}}{\mathsf{f}^g_{max}} m_g^a. \tag{5.17}$$

Given a set of threads that have been allocated on group g: we define as *needed strength* Ω^g of these threads in group g as the minimum necessary *strength* of the group for these threads to be feasible.

Theorem 3. *The needed strength per each group is bounded by the total baseline utilization of the thread set ($U^g(\mathsf{f}^g_{max})$) and by the maximum current strength that group g can provide (m_g):*

$$\Omega^g \in [\min(U^g(\mathsf{f}^g_{max}), m_g), \max(U^g(\mathsf{f}^g_{max}), m_g)] \tag{5.18}$$

Proof. From Lemma 2, we have:

$$U^g(\mathsf{f}^g_{op}) \leq \frac{\mathsf{f}^g_{op}}{\mathsf{f}^g_{max}} m_g^a$$

The operating frequency is less than the maximum frequency $\mathsf{f}_{op} \leq \mathsf{f}_{max}$ and the number of active cores is $1 \leq m_g^a \leq m_g$. Obviously, $U^g(\mathsf{f}^g_{max}) \leq U^g(\mathsf{f}^g_{op})$, and the equation of Lemma (2) becomes:

$$U^g(\mathsf{f}^g_{max}) \leq U^g(\mathsf{f}^g_{op}) \leq \frac{\mathsf{f}^g_{op}}{\mathsf{f}^g_{max}} m_g^a \leq m_g$$

Thus,

$$U^g(\mathsf{f}^g_{max}) \leq \Omega^g \leq m_g$$

□

Our approach is "greedy", because it increments the needed strength Ω^g of groups at each iteration until a feasible schedule is found, or the needed strength of all groups reaches m_g. Our approach, described in Algorithm (11), allocates the maximum possible number of threads to one group at each iteration in three steps:

- Select the frequency for the current group by invoking Algorithm 5.

Algorithm 4 Full algorithm

Input: $\mathcal{T}$: *TaskSet*
for $(g \in \mathcal{G}_g)$ **do**
  compute U^g
  $\mathcal{S}^g = min\{U^g, |\mathcal{G}_g|\}$
  while (not feasible) & $(\mathcal{S}^g <= \mathcal{S}^g_{max})$ **do**
    $\mathcal{S}^g = \Omega^g$
    selectFrequencies($\mathcal{S}^g$)
    feasible = allocate($\mathcal{T}$, $\mathcal{A}$);
    $\Omega^g = \Omega^g +$ strengthIncValue
  end while
  if (task-queue = $\emptyset$) **then**
    return true;
  end if
end for
return false;

- Try to allocate the maximum number of threads to the current group by invoking Algorithm 12. We check the schedulability of each core group using a parallel partitioning heuristic using a schedulability test relying on dbf computation (see Section 5.3.4).
- If the needed strength Ω^g is less then the maximum strength of the current group g and the schedule of step 2 is not feasible, the current strength will be increased by *strengthIncValue*.

We now describe each step in more details.

5.3.3 Frequency selection

The goal of this step is to select the operating frequency of the current group and the number of cores that can be turned off (i.e. set in deep power state). Our frequency selection algorithm is simple: at first it assumes that all cores are active ($m^a_g = m_g$), thus the needed strength is distributed *equally* on all cores (because all cores have the same operating frequency). Hence, frequency is computed by inverting Equation (5.17). The resulting value may be less than the least effective frequency of all threads. Thus, the computed frequency is set to the maximum value between the minimal effective frequency and f^g_{op}. However such frequency may not be available, so we approximate it with the next available mode (ceil-to-next function in Algorithm 5).

Since the frequency may be higher than expected due to the ceiling, the current strength may had be increased. As a consequence, it is maybe possible to set some cores in deep power state. The number of cores to be set in deep power state is computed as the difference between the number of cores of the current group and the ratio of the increased strength and the first input strength. Algorithm 5 describes the frequency selection, its complexity is $O(1)$.

5.3.4 CP partitioning

We proceed now to task decomposition and thread allocation. The goal of this step is to select one cut-point according to which the task is decomposed into a set of parallel threads and then, allocate these threads. Algorithm 12 performs decomposition and allocation and it consists of three steps:

1. selecting a task and a core,
2. computing excess-time,
3. selecting cut-points and assigning threads.

Algorithm 5 Frequency Selection

Input: strength$\mathcal{S}^g$
$\mathsf{leastFreq}^g = \mathsf{minof}(\mathsf{f}^g_{\mathsf{eff}})$
$\mathsf{f}^g_{op} = \frac{\mathcal{S}^g}{m_g}\mathsf{f}_{max}$
if ($\mathsf{f}^g_{op} < \mathsf{leastFreq}^g$) **then**
  $\mathsf{f}^g_{op} = \mathsf{leastFreq}^g$
end if
$\mathsf{f}^g_{op} = \text{ceil-to-next}(\mathsf{f}^g_{op})$
set-deep-power-state for ($m_g - \left\lfloor \frac{\frac{\mathsf{f}^g_{op}}{m_g}\mathsf{f}^g_{max}}{\text{input}\mathcal{S}^g} \right\rfloor$) cores

Algorithm 6 allocate

```
for (τ ∈ 𝒯) do
  for (P ∈ 𝒢_g) do
    evaluate excess-time
    (Th_1, Th_2) = LookForCP(τ, excess-time)        ▷ cut-point selection
    if Th_1 != null then
      allocate(Th_1, P)
      𝒯 = 𝒯 + Th_2
    else
      if last core group then
        return false;
      else
        𝒯_next = 𝒯_next + τ
      end if
    end if
  end for
end for
return false;
```

First Step The algorithm uses a queue of tasks that contains the tasks that have not yet been allocated. The queue is ordered according to some criteria (for example, decreasing utilization, or rate monotonic, etc.). As we will show in Section 5.4, the results do not depend too much on how the queue is ordered.

In the first step, we select the task at the head of the task queue, let it be τ_i, and the first core of the current core group, let it be j. $\mathcal{T}_j$ defines the set of threads already allocated on core j.

Second step The second step consists in performing the schedulability analysis. This step is very similar to the same step described in the previous chapter. However, a slight differences exist and that is why we report the lemmas and theorems with the slight differences. The goal of this step is to define the maximum portion of the execution time of task τ_i that can be allocated on core j, such that the system remains feasible. The maximum execution time of task τ_i corresponds to the single thread version of the task (since all execution times of the parallel threads are less than the sequential version of the task). We start from the assumption that $\mathcal{T}_j$ is feasible on core j, thus:

$$\forall t \in [0 - t^*], \quad \mathsf{dbf}(\mathcal{T}_j, t) \leq t \tag{5.19}$$

To be feasible on core j, the thread set $\mathcal{T}_j \cup \{\mathsf{Th}_{i,1,1}\}$ must verify the following condition, where $\mathsf{Th}_{i,1,1}$ corresponds to the single thread version of task τ_i:

$$\forall t \in [0 - t^*], \quad \mathsf{dbf}(\mathcal{T}_j \cup \{\mathsf{Th}_{i,1,1}\}, t) \leq t \tag{5.20}$$

Lemma 3. *Let $\mathcal{T}_j$ be a set of threads feasible under EDF on core j, and Th be a periodic thread not belonging to $\mathcal{T}_j$. Then, $\exists \mathcal{C}', 0 \leq \mathcal{C}' \leq \mathcal{C}_{\mathsf{Th}}$ such that the following condition is always verified.*

$$\forall t \in [0 - t^*], \mathsf{dbf}(\mathcal{T}_j, t) + \left\lfloor \frac{t + \mathsf{T}_i - \mathsf{D}_i}{\mathsf{T}_i} \right\rfloor (\mathcal{C}_{\mathsf{Th}} - \mathcal{C}') \leq t \tag{5.21}$$

Proof. If we select $\mathcal{C}' = \mathcal{C}_{\mathsf{Th}}$ the second term is zero, and since $\mathcal{T}_j$ is schedulable by hypothesis, the lemma is verified. □

Let us suppose that $\mathcal{T}_j \cup \{\mathsf{Th}_{i,1,1}\}$ is not feasible on core j, for at least for one value of t Condition (5.20) is not verified for t. Hence, we estimate the largest portion of execution time of task τ_i that can be allocated on core j, by computing $\mathcal{C}'$ as the minimum value such that Condition (5.21) is verified.

Theorem 4. *Let $\mathcal{T}_j$ be a set of threads feasible under EDF on core j, Th_1 be a periodic thread not belonging to $\mathcal{T}_j$, and let us assume that $\mathcal{T}_j \cup \{\mathsf{Th}_1\}$ is not feasible on core j.*

$\mathcal{T}_j \cup \mathsf{Th}_2$ is feasible if Th_2 has the same period and deadline as Th_1 and an execution time less than $\mathcal{C}_{\mathsf{Th}_1} - \mathcal{C}'$ where:

$$\mathcal{C}' = \max_{t \in \{0 \cdots t^*\}} \left(\frac{t - \mathsf{dbf}(\mathcal{T}_j \cup \{\mathsf{Th}_1\}, t)}{\left\lfloor \frac{t + \mathsf{T}_i - \mathsf{D}_i}{\mathsf{T}_i} \right\rfloor} \right) \tag{5.22}$$

Proof. It is sufficient to invert Equation (5.21). □

In the rest of this work $\mathcal{C}'$ is called excess-time and computed as Equation (5.22).

Third step Now that we have introduced the concept of excess-time, we use a simple heuristic to select cut-points and assign threads. According to the value of the excess-time and the single thread version of the task, only 3 cases need to be checked as in the previous chapter:

1. excess-time $= 0$, it means that the single thread version of the task and the thread-set already allocated on core j is feasible. Thus, single thread version of the task is allocated on the current core (the task is run in sequential way).

2. excess-time is equal to execution time of the single thread version of the task (excess-time $= \mathcal{C}^g_{\tau,1,1}(\mathsf{f}^g_{op})$), it means that the single thread version of the task is fully in excess on current core j as in the previous chapter.Xe seek to allocate threads on the next core $j + 1$.

3. excess-time is greater than zero, and less than the execution time of single thread version of the task ($0 <$ excess-time $< \mathcal{C}^g_{\tau,1,1}(\mathsf{f}^g_{op})$). The single thread version of the task is not schedulable on current core j, but any thread(s) of the task that have the execution time less than ($\mathcal{C}^g_{\tau,1,1}(\mathsf{f}^g_{op}) -$ excess-time) can be allocated on the current core.

In the last step the allocation is different from the one in the previous chapter. We need to decompose the task according to one possible $\gamma_{i,k} \in \Gamma_i$. We use a search function that returns two lists, $\sigma'_{i,k}$ and $\sigma''_{i,k}$: the first one consists of threads that fit on the current core, whereas the second one consists of threads that do not fit and need to be allocated elsewhere.

The threads in $\sigma''_{i,k}$ are used to create a new sub-task. This task contains two cut-points: the first cut-point contains a single thread whose execution time is the sum of the execution times of all threads in $\sigma''_{i,k}$: the ct (respectively mt) of the all these threads are summed to form the execution time of the single thread. The second cut-point of this new sub-task is simply $\sigma''_{i,k}$.

Theorem 5. *Let τ_i be a task that we want to allocate, and let $\gamma_{i,k}$ be one of its cut-points. Let* excess-time *be the excess time generated by the single thread version of τ_i, and let $\sigma'_{i,k}$ and $\sigma''_{i,k}$ be the two thread list obtained by our decomposition.*

The decomposition is feasible if and only if Equation (5.23) is verified:

$$\begin{aligned} &\exists \sigma''_{i,k} \subset \gamma_{i,k}, \quad \sigma''_{i,k} \neq \emptyset, \gamma_{i,k} \neq \sigma''_{i,k} \\ &\sum_{\mathsf{Th}_{i,k,z} \in \sigma''_{i,k}} \mathcal{C}^g_{i,k,z}(\mathsf{f}^g_{op}) \geq \text{excess-time}. \end{aligned} \tag{5.23}$$

Proof. $\sigma'_{i,k}$ can not be empty, otherwise no thread is allocated on the current core. Furthermore, $\sigma'_{i,k}$ can not contain all threads in the cut-point, otherwise the single thread version of the task is feasible on the current core. Therefore, the $\sigma'_{i,k}$ must fit on the current core. □

For one task, numerous $\sigma'_{i,k}$ can verify Condition (5.23). We select the solution according to two criteria:

1. first, we take the solution with the minimum overheard;
2. in case of tie, we select the solution with the maximum number of threads in $\sigma''_{i,k}$: in fact, a cut-point that contains more threads has more flexibility to be decomposed again (if needed).

Example Assume that allocation is done on a big core operating of the maximum frequency. Assume the task cut-point list (Γ) described in the table below (Table 5.3). This task can be

cut-points	Thread	ct^b	mt^b	ct^l	mt^l
$\gamma_{1,1}$	$\mathsf{Th}_{1,1,1}$	8	3	27	7
$\gamma_{1,2}$	$\mathsf{Th}_{1,2,1}$	5	2	16	5
	$\mathsf{Th}_{1,2,2}$	5	2	16	5
$\gamma_{1,3}$	$\mathsf{Th}_{1,3,1}$	2	1	8	3
	$\mathsf{Th}_{1,3,2}$	2	1	8	3
	$\mathsf{Th}_{1,3,3}$	5	1	17	3

TABLE 5.3: Example of cut-point selection : cut-point details

decomposed into several ways. In fact, the only cut-point that verifies Condition (5.23) is $\gamma_{1,3}$, and it can be decomposed in 4 different ways, according to the following table (Table 5.4): We

N	$\gamma_{1,k}$	σ'	σ''
1	$\gamma_{1,3}$	$\mathsf{Th}_{1,3,2}$, $\mathsf{Th}_{1,3,3}$	$\mathsf{Th}_{1,3,1}$
2	$\gamma_{1,3}$	$\mathsf{Th}_{1,3,1}$, $\mathsf{Th}_{1,3,3}$	$\mathsf{Th}_{1,3,2}$
3	$\gamma_{1,3}$	$\mathsf{Th}_{1,3,1}$, $\mathsf{Th}_{1,3,2}$	$\mathsf{Th}_{1,3,3}$
4	$\gamma_{1,3}$	$\mathsf{Th}_{1,3,3}$	$\mathsf{Th}_{1,3,1}$,$\mathsf{Th}_{1,3,2}$

TABLE 5.4: Example of cut-point selection: the selected results

choose the last one because all solutions have the same overhead, but the last one contains more threads in σ''.

The complexity of Algorithm 7 of cut-point selection is $\Theta(\max(|\Gamma_i|) \times \max(|\gamma_{i,k}|))$. However, please notice that after the first decomposition the number of different cut points is reduced to only two because threads must belong the same cut-point and Condition (5.23) must be verified. Thus, the average complexity is reduced considerably after the first allocation.

Algorithm 7 lookForCp

Input: excess-time,f_{op}: Frequency,τ_i:Task, g: core group
Output:(List[ThreadS], List[ThreadS])
MoreThanTH: List[ThreadS] = Nil;
lm: List[ThreadS] = Nil;
Lout: List[ThreadCP] = Nil;
for $(all\ \gamma_{i,k} \in \Gamma_i)$ **do**
  cum= 0
  l = Nil
  for $(all\ \mathsf{Th}_{,i,k,z,\in\gamma_{i,k}})$ **do**
    if $(\mathcal{C}^g_{i,k,z}(\mathsf{f}_{op}) \leq (\mathsf{ct}^g_{\tau,1,1}(\mathsf{f}_{op}) - \mathsf{excess\text{-}time}))$ **then**
      lm = lm.add($\mathsf{Th}_{g,i,k}z$)
      lout = $\gamma_{i,k}$ / lm
      $cum+ = \mathcal{C}^g_{i,k,z}(\mathsf{f}_{op})$
      if (cum $\geq$ Kept) **then**
        lm = lm.add(MoreThanTH);
        lout = $\gamma_{i,k}$ / lm;
      else
        MoreThanTH += $\mathsf{Th}_{g,i,k}z$
      end if
    end if
  end for
end for
return (lm, lout)

Algorithm 11 combines all the heuristics described in this section. The complexity of this algorithm is $\theta(\mathsf{nbrIt} \times n \times m \times h \times \sum_{i=1}^{n} max(|\ \gamma_{i,k}\ |))$ where nbrIt is the number of iterations from the lowest Ω^g to the highest Ω^g for each group g. This algorithm is referred as CPM in the rest of this work.

5.4 Results and discussions

In order to evaluate our approach, we apply our heuristics on a large number of randomly generated synthetic task sets. For the experiments, we modelled the big/LITTLE architecture described in Section 5.2.1: our platform consists of 8 cores, divided into 2 groups of 4 cores each (little and big).

5.4.1 Task Generation

In this chapter, we tested two scenarios. In the first, we use the UUniFast-discard algorithm. In the second, we removed the latest constraint of UUniFast-discard ($u_i \leq 1$), because we allow the single thread version of tasks to have an utilization greater than 1. The modified UUNIfast-Discard is described in Algorithm 8.

The modified UUNIFAST-discard is used to generate n baseline utilization factors of the single thread version on little cores operating at f^g_{max}. To obtain realistic utilization, we generate the baseline utilization of each task on the big core by multiplying the baseline utilization for little cores by a random factor between $[0.3, 0.4]$.

Goossens et al. in Goossens and Macq, 2001 have shown that the *hyperperiod* grows exponentially with the greatest value of period. They also proposed a method to bound the *hyperperiod* and generate periods T_i. We use their method for generating task periods.

Therefore, we compute the computation time of the single thread for little and for big cores:

Algorithm 8 Utilization generation

Input: N,U: integer
Output: u: Array[N]
nextSumU: Float = 0;
numberOftask :Integer = N;
while $(numberOfTasks > 0)$ **do**
  var sumU = U;
  for $(i = 0\ to\ N-1)$ **do**
    nextSumU = (sumU $\times$ (random()$^{1/(N-(i+1))}$)
    u[i]=sumU-nextSumU
    sumU=nextSumU
  end for
  u[nbTask-1]=sumU
  numberOfTasks= numberOfTasks -1
end while
return u

- $\mathcal{C}^l_{i,1,1}(\mathsf{f}^l_{max}) = u^l_i \times T_i$
- $\mathcal{C}^b_{i,1,1}(\mathsf{f}^b_{max}) = u^b_i \times T_i$

We define as P^g_i the average rate of central memory access of task τ_i on group g. Then, we generate mt and ct for the single thread version as:

$$\begin{aligned}
\mathsf{mt}^b_{i,1,1} &= \mathcal{C}^b_{i,1,1}(\mathsf{f}^b_{max}) \times P^b_i \\
\mathsf{ct}^b_{i,1,1}(\mathsf{f}^b_{max}) &= \mathcal{C}^b_{i,1,1}(\mathsf{f}^b_{max}) - \mathsf{mt}^b_{i,1,1} \\
\mathsf{mt}^l_{i,1,1} &= \mathcal{C}^l_{i,1,1}(\mathsf{f}^l_{max}) \times P^l_i \\
\mathsf{ct}^l_{i,1,1}(\mathsf{f}^l_{max}) &= \mathcal{C}^l_{i,1,1}(\mathsf{f}^l_{max}) - \mathsf{mt}^l_{i,1,1}
\end{aligned}$$

where P^b_i is generated between $[P_{min}, P_{max}]$ and P^l_i is generated between $[P^b_i, P_{max}]$. This parameter does not change during run-time, however in reality it depends on the interference from other active jobs.

Task τ_i has K_i cut-points, where K_i is generated randomly between 2 and 5. For each cut-point $\gamma_{i,k}$, We generate between 2 and 8 threads.

We use again UUniFast to generate $|\gamma_{i,k}|$ baseline utilization factors for the threads belonging to cut-point $\gamma_{i,k}$ on little cores.

Then we generate the baseline utilization for each thread on big cores by multiplying little's utilization by a random factor between 0.3 and 0.4. To take into account the overhead of decomposition (due to synchronization barriers, scheduling, etc.), we inflate the utilization of the thread to $U'_i = U_i \times (1 + \mathsf{cost}\ \ | \gamma_{i,k} |)$, where cost is a constant that represent the overhead in percentage. We fixed this overhead to $\mathsf{cost} = 0.05$ per thread.

Finally, the deadline is generated in the interval $[P_r \times T_i, T_i]$ where P_r is randomly generated between $[0.75, 1]$. The task set generation algorithm is described in Algorithm 10.

5.4.2 Simulations

Our algorithm has no direct competitors, in the sense that classic partitioning heuristics do not take into account intra-task parallelism or task decomposition. Therefore, we have no choice but to compare our heuristics to classic bin-packing heuristics, such as Best Fit (*BF*), Worst Fit (*WF*) and First Fit (*FF*).

The BF, FF, WF were implemented as they are stated in the literature without any parallelization features, and without suffering from any parallelization costs. The same frequency

Algorithm 9 Heuristics

Input: $\mathcal{T}$: *TaskSet*
for $(g \in \mathcal{G}_g)$ **do**
  compute U^g
  $\mathcal{S}^g = min\{U^g, |\mathcal{G}_g|\}$
  while (not feasible) & $(\mathcal{S}^g <= \mathcal{S}^g_{max})$ **do**
    $\mathcal{S}^g = n\mathcal{S}^g$
    selectFrequencies($\mathcal{S}^g$)
    feasible = **BF** or **WF** or **FF**($\mathcal{T}$, $\mathcal{A}$);
    $n\mathcal{S}^g = n\mathcal{S}^g +$ strengthIncValue
  end while
  if (task-queue = $\emptyset$) **then**
    return true;
  end if
end for
return false;

selection algorithm as the one for CPM is combined with BF, FF and WF such that they have the ability to select cores frequencies, and power states as described in algorithm 9.

To be fair, we compare BF, WF, FF against our heuristic (CPM) on two different scenarios. In the first scenario, we impose that the single thread version of each task must have a baseline utilization on little cores inferior to 1. This scenario is advantageous for BF, WF and FF because they allocate only the single thread version of each tasks, whereas CPM may need to decompose the task into several threads, thus suffering from parallelization costs. In the second scenario, we relax the constraint on utilization by allowing the single thread version of the tasks to have a baseline utilization on little cores greater than 1. We expect that this scenario will be advantageous to CPM compared to BF, WF and FF because the sequential execution is not possible on little cores for tasks with density greater than 1.

For both scenario 1 and 2, we generated 1000 task set per each total baseline utilization and total baseline utilization is varied from 0.5 to 16.

Algorithm 10 TaskSetGeneration

Output: taskSet
u^l:Array[Float]=UUnifastDiscard()/UUnifastDiscardModified()
u^b:Array[Float]=u^b $\times$ random$(0.3, 0.4)$
for $(i = 0\ to\ n-1)$ **do**
  $\mathcal{C}^b_{i,1,1}(\mathsf{f}^b_{max}) = \mathsf{T}_i \times u^b_i$
  $\mathcal{C}^l_{i,1,1}(\mathsf{f}^l_{max}) = \mathsf{T}_i \times u^l_i$
  $P^b = \mathsf{random}(P_{min}, P_{max})$
  $\mathsf{mt}^b_{i,1,1} = \mathcal{C}^b_{i,1,1}(\mathsf{f}^b_{max}) \times P^b$
  $\mathsf{ct}^b_{i,1,1}(\mathsf{f}^b_{max}) = \mathcal{C}^b_{i,1,1}(\mathsf{f}^b_{max}) - \mathsf{mt}^b_{i,1,1}$
  $P^l = \mathsf{random}(P^b, P_{max})$
  $\mathsf{mt}^l_{i,1,1} = \mathcal{C}^l_{i,1,1}(\mathsf{f}^l_{max}) \times P^l$
  $\mathsf{ct}^l_{i,1,1}(\mathsf{f}^l_{max}) = \mathcal{C}^l_{i,1,1}(\mathsf{f}^l_{max}) - \mathsf{mt}^l_{i,1,1}$
  $\mathsf{D}_i = \mathsf{random}(0.75, 1) \times \mathsf{T}_i$
  add the sequential thread to Γ_i
  k=random$(2, 5)$;
  for $(k = 1\ to\ K)$ **do**
    Z=random$(2, m)$;
    UZ^l = UUniFast(Z,1+ Cost $\times$ Z);
    UZ^b = UZ^b $\times$ random$(0.3, 0.4)$;
    for $(z = 0\ to\ Z-1)$ **do**
      $\mathsf{mt}^b_{i,k,z} = \mathsf{mt}^b_{i,1,1} \times \mathrm{UZ}_z$;
      $\mathsf{ct}^b_{i,k,z}(\mathsf{f}^l_{max}) = \mathsf{ct}^b_{i,1,1}(\mathsf{f}^b_{max}) \times \mathrm{UZ}_z$;
      $\mathsf{mt}^l_{i,k,z} = \mathsf{mt}^l_{i,1,1} \times \mathrm{UZ}_z$;
      $\mathsf{ct}^l_{i,k,z}(\mathsf{f}^l_{max}) = \mathsf{ct}^l_{i,1,1}(\mathsf{f}^l_{max}) \times \mathrm{UZ}_z$;
      add Thread to CP
    end for
    add CP to Γ_i
  end for
  add Task to taskset
end for
return taskset;
random(a, b): is a function that generate a random number between a and b

5.4.3 Scenario 1

In Figure 5.12, we plot the number of schedulable task sets as a function of the total baseline utilization. For small total utilization (< 10), all BF, WF, FF, and our heuristic (CPM) can schedule 100 % of the task sets. However, when the total utilization is greater than 10, CPM outperforms the classic heuristics. In fact, our heuristic always tries to allocate the maximum number of threads per each core, by performing task decomposition.

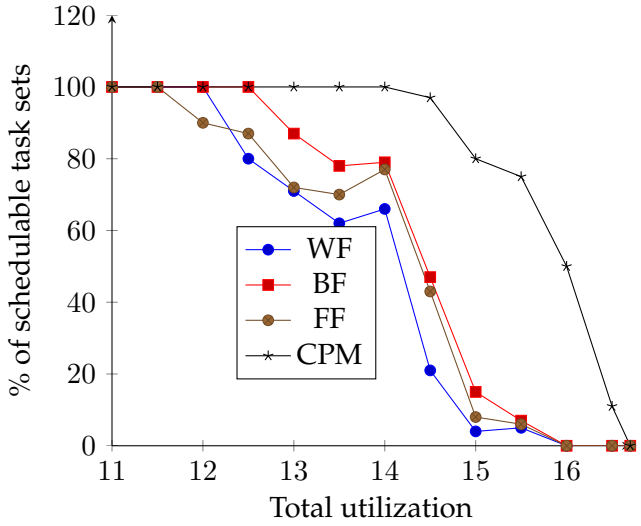


FIGURE 5.12: The number of schedulable task sets

A similar effect can also be seen in Figure 5.13, where we show the average utilization per little and big cores as a function of the total utilization. For little cores, our approach reaches an average utilization very close to 1 which is greater than all other classical heuristics. Again, this can be explained by the fact that the task decomposition feature of CPM that can split a large task into several smaller parallel threads that can better fit on an already loaded core.

Even for big cores, the average utilization is close to 1 and higher than the one obtained with BF, WF, FF, which may seem to be contradictory with what we just observed for little cores. In fact, CPM selects smaller frequencies than all other classic heuristics as shown in Figures 5.14a, 5.14b. Even when BF, WF and FF allocate the same threads as CPM on the big cores, the latter is able to set lower frequencies (so the average utilization is higher).

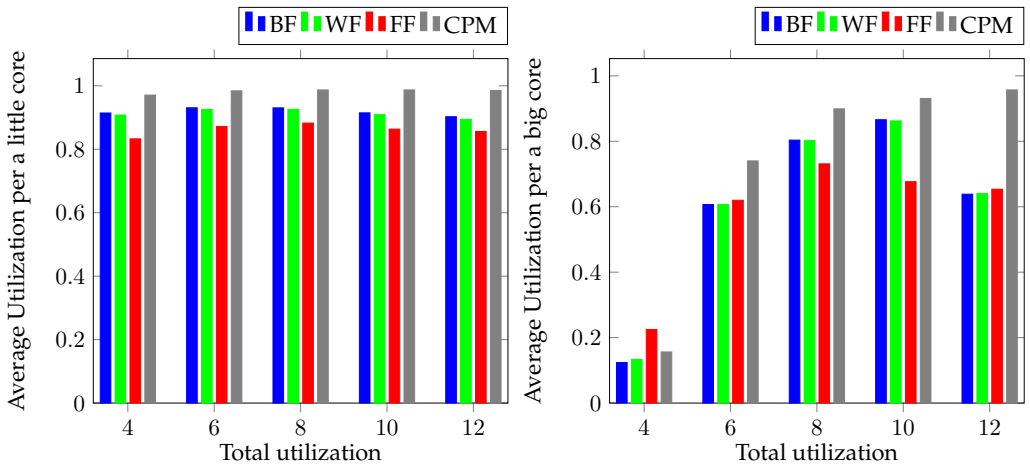
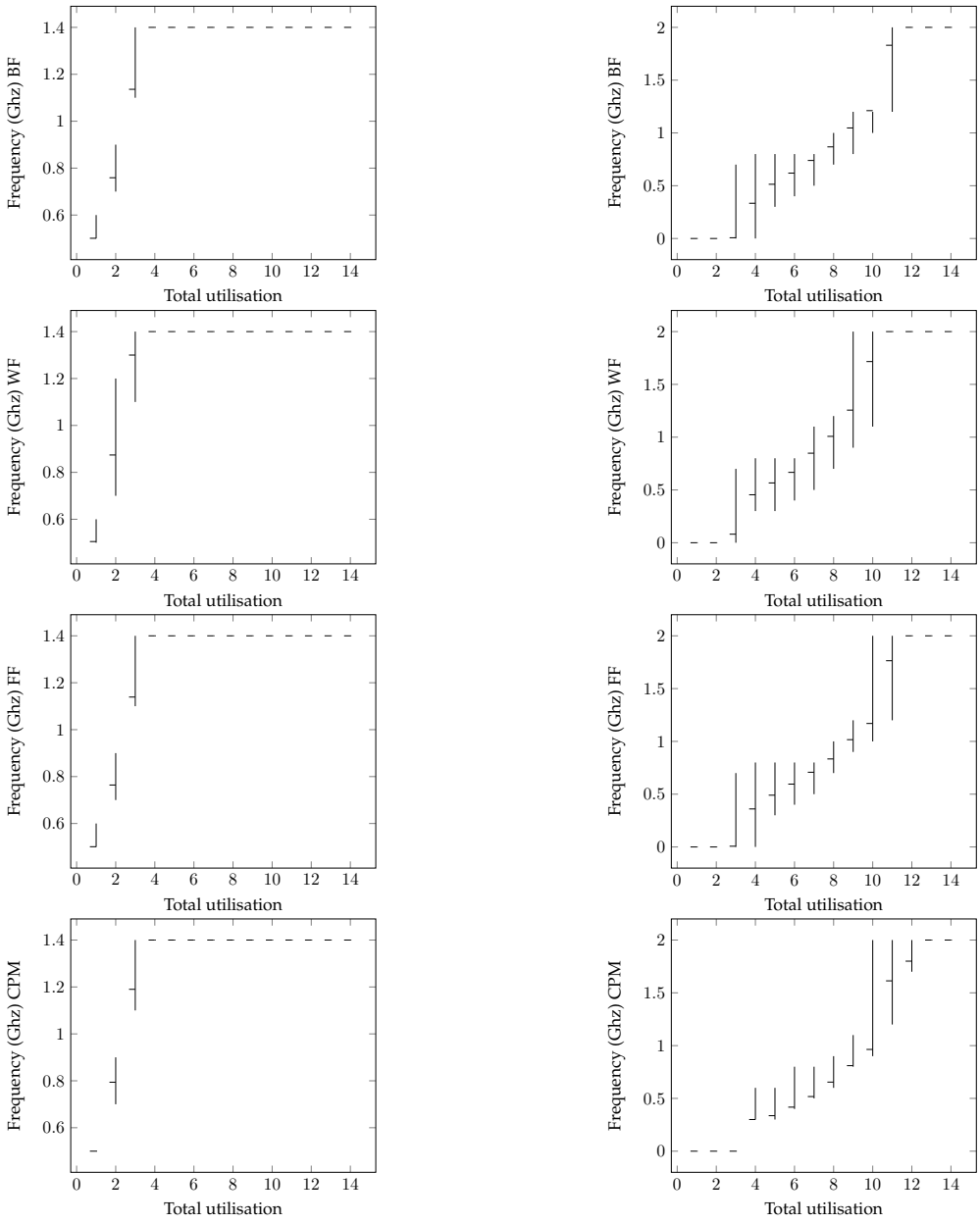


FIGURE 5.13: Average utilization per each core group

Figure 5.14a reports the maximum, the minimum and the average selected frequency for little cores as a function of total utilization using the classic heuristics and CPM. We notice that

for little cores, CPM sets the frequency of little cores slightly higher than the other heuristics, because CPM favours little cores over big cores.



(A) Max, Min, Avg Selected Frequency on Little cores

(B) Max, Min, Avg Selected Frequency on Big cores

FIGURE 5.14: Scenario 1: Selected Frequency for big and little cores

For big cores, CPM sets the average frequency lower than any other heuristics, as shown Figure 5.14b. In general, this allows us to reduce energy consumption on big cores (as can be seen in Figure 5.15) thanks to the fact that CPM allocates a smaller number of threads on big cores than all other heuristics. The distance between the maximum and the minimum frequency found during the experiments is the smallest between all heuristics. This means that: 1) our algorithm is quite effective and gives more stable results; 2) the practical average complexity of CPM is lower because function *allocate* in Algorithm 11 is invoked less times.

In Figure 5.15, we plot the energy consumption of little and big cores as a function of the total utilization. For little cores, CPM consumes more energy than bin-packing heuristics. However, this energy is recovered on big cores, where CPM consumes less power than the best

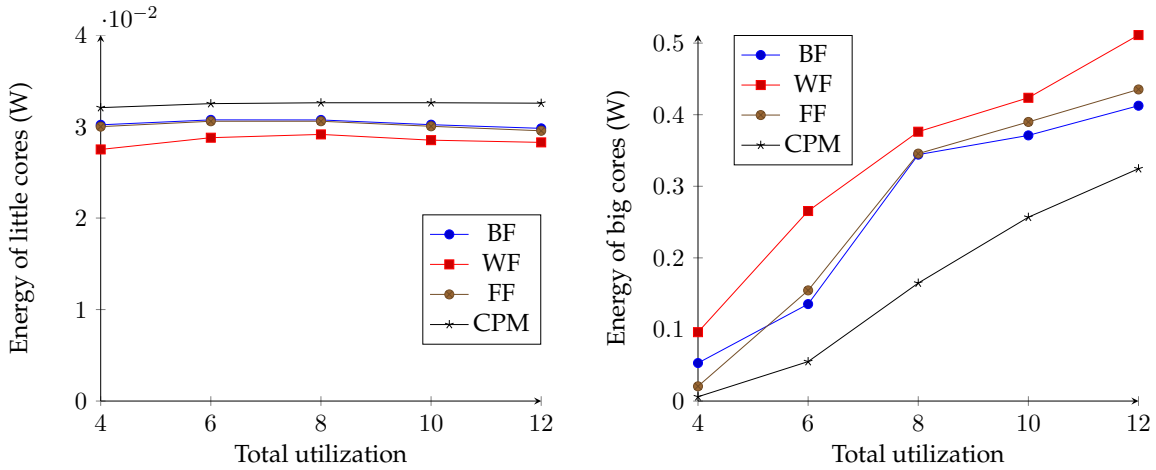


FIGURE 5.15: Energy consumption for big and little cores

bin-packing heuristics BF.

5.4.4 Scenario 2

In this scenario, the baseline utilization of the single thread version of a task executing on the little cores may be greater than 1.

Figure 5.16 reports the number of schedulable task sets as a function of total utilization. Since the single thread version of a task may have baseline utilization greater than 1, BF, WF and FF are outperformed by CPM, which keeps the schedulability rate higher.

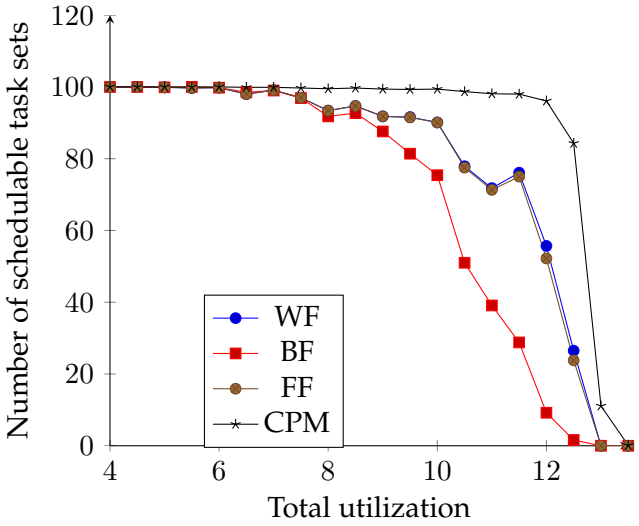


FIGURE 5.16: The number of schedulable task sets

We do not discuss the rest of the results for Scenario 2: the statistical comparison is meaningless because CPM outperform all other heuristics.

5.5 Conclusion

In this chapter, we first presented a general methodology for modelling the energy consumption of periodic tasks on heterogeneous multicore architectures such as the ARM big/LITTLE. We discovered that different tasks have different power dissipation profiles and we proposed an execution time and power consumption model which take task characteristics into account.

Then we proposed an heuristic for parallelizing and allocating threads on heterogeneous multicore platforms, and setting the frequency and the power state of the cores so to reduce the total energy consumption without missing deadlines. Our methodology is effective when compared to classical bin-packing heuristic algorithms. With our work we show the advantages of parallelization from the point of view of the energy consumption. Our model is realistic, as it considers moldable tasks that easily allow specifying parallel decomposition in the style of OpenMP, it does not require specific scheduling synchronization among threads, and can be applied to existing operating systems as Linux. To deal with the dynamic task behavior, in the next chapter we extend the di-graph task model proposed Stigge et al., 2011.

Chapter 6

Modeling parallel tasks with di-graphs

“ Si tu ne profites pas du temps que tu as de libre,
tu n’en profiterais pas davantage quand ce temps serait
dix fois plus considrable. ”

Alexandra David-Neel

Contents

6.1 Introduction

Many real-time task models have been proposed in the past to deal with the every increasing complexity of real-time software. For example, the classical Liu and Layland model can not express the dynamic behavior of tasks whose processing depends on the input data. The multiframe model has been proposed by Mok and Chen, 1996 as a first simple extension to deal with multimedia tasks, where the behavior of each instance of a periodic task depends on input and follows itself a periodic cycle. Later, Baruah, 2010 proposed the GMF (Generalized MultiFrame model) to express alternative processing described by a Directed Acyclic Graph (DAG). Lately, Stigge et al., 2011 proposed the di-graph model that further extends the GMF model by allowing to express the alternative behaviors with a finite automaton (we summarize the di-graph model in the next section).

However, the di-graph model does not express the potential parallelism that could exist in such tasks. In this chapter, we present a generalization of the di-graph model to effectively deal with parallel tasks. In particular, we allow a task to express a dynamic level of parallelism by using a special form of di-graph.

Such extension is useful in many parallel application that exhibit dynamic behavior and some form of real-time constraints. For example, in the MTI (*Moving Target Indication*) radar application, airborne radars scan ground surfaces and air looking for targets. Since precise detection is a time consuming task, initially, the area is quickly scanned for potential target objects. When the system detects a potential target, the processing focuses on the concerned area, and more effective detection algorithms are used only on a subset of data. This kind of incremental processing is also found in many other situations, e.g. communication systems, satellite navigation systems, or active sonar detection systems, in general in any STAP (Space-Time Adaptive Processing stated by Ward, 1994; Guerci, 2014) processing.

MPEG Video encoding is another example of tasks that could be modeled by our model. A MPEG video is a compound set of frames of different types (**I-Frame**, **P-Frame**, **B-Frame**). The **I-Frame** (Intra-coded picture) is a fully specified picture compressed with JPEG. **P-Frame** (Predicted picture) and **B-Frame** (Bi-predictive picture) hold only part of the image information, so they occupy less space to be stored than an I-frame. The encoder produces a (mostly) periodic sequence of frames: for example, a typical sequence consists of 12 frames of type **IBBPBBPBBPBBI**. However, the sequence may change dynamically. Notice that for video decoding, different processing is applied according to the frame type.

In this chapter we first propose the *Parallel Di-Graph* model, that extends the di-graph proposed by Stigge et al., 2011 to parallel tasks. We show also how Array-OL, proposed by Glitia, Dumont, and Boulet, 2010, an existing specification language dedicated to STAP applications, can be used to specify a parallel di-graph.

Then, we address the problem of allocating a set of parallel tasks, each one modeled by a parallel digraph, onto a multicore architecture consisting of a set of identical cores, in a way that respects all timing constraints and select the minimum possible frequency. We propose a sufficient feasibility test for partitioned scheduling of a set of di-graph tasks on an identical core platform. Based on this test, we also propose a set of heuristics for parallelization and partitioning of a set of di-graph tasks. Our heuristics allow to select the operating frequency of all cores to study the effectiveness of task models against each other at variable frequencies. Our frequency selection algorithm can be used to reduce the energy consumption. A set of synthetic experiments are presented that emphasize the effectiveness of our model against other less expressive models proposed in the literature.

6.2 Some related work

In this section, we report some related works to non-parallel real time task models. The parallel models proposed in the literature of real-time systems extends the well-known Liu and Layland

model. Thus, these models have basically a fixed period and execution time. However, tasks like MPEG encoding or MTI can not be effectively expressed by these models. They can only express the worst case of different instances as the worst case execution, however these tasks are compound of different type of processings at different instances. Researchers in real-time systems (not for parallel tasks) have proposed more expressive models like the multiframe model (Mok and Chen, 1996). In this model, a task is expressed by a repetitive sequence of instances with different execution time for each. Baruah, 2010 generalized this model to express the dependency between instances like a condition on an external event. In the model of Baruah a task is expressed by a graph where connected nodes express the appearances order of instances of a task. Each graph has an entry and exits. Stigge et al., 2011 extended the model in Baruah, 2010 and expressed a task by a digraph. We extended the model of Stigge et al., 2011 to parallel tasks. We refer to the model proposed by Stigge et al., 2011 by Stigge model, and Liu and Layland model as the periodic model.

6.3 System Model

6.3.1 Architecture model

We consider a set of m identical cores. Cores have the same micro-architecture and share the same frequency characteristics (operating frequency f_{op}, maximum frequency f_{max} and minimum frequency f_{min}). *ODROID C2*[1] is an example of a such computing platform. It is a compound set of 4 ARM Cortex-A53 operating on the same frequency. Each core has 2 $\times$ 32Kb of Instructions/Data L1-cache and all cores share 512Kb of L2-cache and an external SDRAM-DDR3 of size of 2Gb. In this work, we allow cores to change the frequency in order to compare the schedulability rates at frequency variation, and the effectiveness of scheduling algorithms against each other at variable frequencies. This work can be easily extended to reduce the energy consumption.

6.3.2 Task Model

Each task τ_i in the system is characterized by a tuple $\tau_i = (\mathsf{D}_i, \mathsf{T}_i, \mathsf{G}(\mathsf{V}_i, \mathsf{E}_i))$ where D_i is the task relative deadline; T_i is the task period, i.e. the time between the releases of two consecutive instances of τ_i, as we consider constrained deadline tasks, $\mathsf{D}_i \leq \mathsf{T}_i$; $\mathsf{G}(\mathsf{V}_i, \mathsf{E}_i)$ is a *di-graph* where V_i is the set of vertices, and E_i is the set of edges. Each vertex ($v_{i,j} \in \mathsf{V}_i$) is one possible parallel decomposition of task τ_i into a set of parallel threads ($v_{i,j} = \{\mathsf{Th}_{i,j,1}, \cdots, \mathsf{Th}_{i,j,z}\}$). We denote by $\mathsf{Th}_{i,j,z}$ thread z of vertex $v_{i,j}$ of task τ_i, and by $|v_{i,j}|$ the number of threads of vertex $v_{i,j}$.

The execution time of thread $\mathsf{Th}_{i,j,z}$ is noted $\mathcal{C}^{\mathsf{th}}_{i,k,z}(\mathsf{f})$. The execution time of vertex $v_{i,j}$ is defined as the sum of the execution times of all of its threads.

$$\mathcal{C}^{\mathsf{v}}_{i,j}(\mathsf{f}_{op}) = \sum_{fz=0}^{v_{i,j}} \mathcal{C}^{\mathsf{th}}_{i,j,z}(\mathsf{f}_{op})$$

The execution time of a vertex corresponds to the execution time of the sequential version of the vertex when all threads are allocated on the same core.

The set of edges expresses the precedence order between instances of task τ_i: $e(v_{i,s}, v_{i,d}) \in \mathsf{E}_i$ expresses the constraint that instance $v_{i,d}$ may be released only after T_i time units from the release of instance $v_{i,s}$ (see Figure 6.1). A vertex with two or more outgoing edges means that one of the destination vertex will be selected according some internal condition in the code (for example the value of the input data). Since we do not model directly the code behavior, in our model the choice of outgoing edge is *non-deterministic*. An edge that goes from one vertex to the same vertex is called a *self-edge*.

[1]ODROID CU2: http://www.hardkernel.com/main/products/prdt_info.php?g_code=G145457216438&tab_idx=2

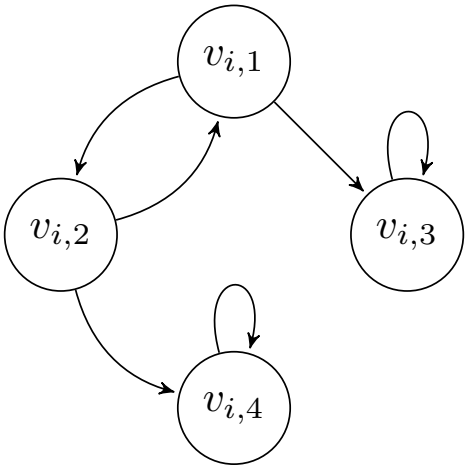


FIGURE 6.1: Example of parallel di-graph task.

We consider that the execution time of a thread scales in a semi-linear way with the frequency as defined by Bini, Buttazzo, and Lipari, 2005.

$$\mathcal{C}^{th} = \frac{\mathsf{ct} \cdot \mathsf{f}_{max}}{\mathsf{f}_{op}} + \mathsf{mt} \tag{6.1}$$

where ct is the part of the execution time of the thread that depends on the operating frequency and mt is the part that does not depend on the operating frequency (e.g. central memory access). We define the execution time parameters $(\mathsf{ct}, \mathsf{mt})$ in the same way as proposed by Zahaf et al., 2016a and Zahaf et al., 2016b

Example Let τ_i be a task $\tau_i = (\mathsf{D}_i = 8, \mathsf{T}_i = 10, \mathsf{G}(\mathsf{V}_i, \mathsf{E}_i))$. $\mathsf{G}(\mathsf{V}_i, \mathsf{E}_i)$ is shown in Figure 6.1 and the details of the execution times of threads of every vertex are reported in the table below (Table 6.1): In this example, we assume that the given execution times correspond to

Vertex	Thread List
$v_{i,1}$	2,2,5
$v_{i,2}$	4,3
$v_{i,3}$	6
$v_{i,4}$	2,2,4

TABLE 6.1: An example of a task modeled by a parallel di-graph

the execution time at the maximum frequency. When the task is in $v_{i,2}$, it is decomposed into two parallel threads ($\mathsf{Th}_{i,2,1}$, $\mathsf{Th}_{i,2,2}$) with an execution time that equals to 4 and 3 units of time, respectively, and must finish the execution within 8 time units. After 10 time units (period), the task moves to one between $v_{i,1}$ or $v_{i,4}$.

If it goes to $v_{i,1}$, it will be decomposed into 3 parallel threads, otherwise it goes to $v_{i,4}$, and it will be decomposed into 3 parallel threads. In this second case, all future instances of the task τ_i will be run according to $v_{i,4}$ because the only outgoing edge from this vertex is a self-edge.

We highlight the fact that we allow two vertices to have the same level of parallelism with different combinations of execution times. We allow also two vertices to have the same level of parallelism and the same amounts of execution time to express different states of the task.

6.4 Parallel applications

In this section, we show how we can model two different applications, an MPEG decoder and a STAP application, using our parallel di-graph model.

6.4.1 MPEG encoding/decoding

MPEG is a standard for compression of video and audio. An example of MPEG encoding/decoding (with a frame rate of 25 FPS) can be expressed with our model as task $\tau_{\text{MPEG}} = (\mathsf{D}_{MPEG} = 40ms, \mathsf{T}_{MPEG} = 40ms, \mathsf{G}(\mathsf{V}_{\text{MPEG}}, \mathsf{E}_{\text{MPEG}},))$.

The corresponding graph $\mathsf{G}(\mathsf{V}_{\text{MPEG}}, \mathsf{E}_{\text{MPEG}},)$ is described in Figure 6.2. The vertex I expresses the I-Frame compression/decompression. This operation is basically JPEG-encoding which is based on DCT [2]. In the I-vertex, the picture is decomposed into several blocks of size 8×8 and then DCT transformations are applied. Here, we assume that encoding I-Frames is done with 4 threads. The P-vertex represents the P-Frame encoding/decoding. A P-Frame is encoded based on its difference on a previous frame. Thus, the frame is divided into macro-blocks and comparing each of the macro-blocks with a corresponding block. The search for correspondence on macro-blocks can be parallelized. The number of parallel threads can be defined according to the size of macro-block and the size of frames. We set the number of threads of this vertex to 2. B-Frame encoding uses the same algorithm as with the P-Frames but it considers the previous and the next frame. Again this can be parallelized, and we assume that it will be done with 3 threads. Notice here that the task behavior depends basically on input data and the level of parallelism may change at run time.

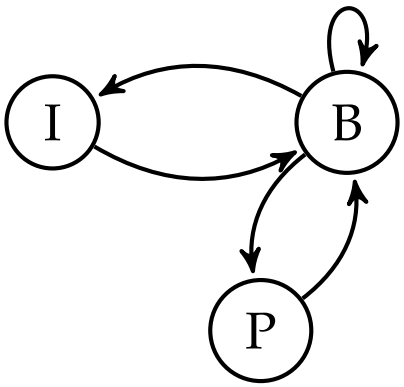


FIGURE 6.2: A di-graph modeling MPEG encoding

6.4.2 Array-OL

The Array-OL language proposed by Boulet, 2007 is a high-level specification language dedicated to multidimensional intensive signal processing applications. It allows to specify both the task parallelism and the data parallelism of these applications by focusing on their complex multidimensional data access patterns. Array-OL expresses all the potential parallelism in intensive signal processing application and does not express any implementation specification. Using some correct-by-construction refactorings of Array-OL specifications as those proposed by Glitia and Boulet, 2008; Glitia et al., 2011, can transform the potential parallelism into effective parallelism with a control on the grain of this effective parallelism. Applying different refactorings give different versions of the application with different parallel execution characteristics.

[2] Discrete Cosine Transformation

The Array-OL language deals with only one type of data structures: multidimensional arrays. The idea of Array-OL is to express data access by the way of regular tilings of the input and output arrays of data-parallel tasks. The number of repetitions of a data-parallel task implies the number of tiles of each input and output array. The tiles are defined as regularly spaced multidimensional sub-arrays, they all have the same shape (for a given array) and they are regularly spaced. Thus the tiling can be expressed as an affine relation. In addition to this affine relation, the indices of the elements of the tiles in the array are considered modulo the shape of the array. This construction (the so-called tilers) allows to express all common data access patterns of multidimensional signal processing applications including sub- or oversamplings, and cyclic accesses. The introduction of delays in Array-OL as proposed by Glitia, Dumont, and Boulet, 2010 completes the toolbox of this language.

Array-OL has been included in the MARTE UML profile for Modeling and Analysis of Real-Time Embedded Systems (the MARTE specifications can be found in Object Management Group, 2009) to express data parallel tasks and repetitive hardware. The refactorings are available in the Gaspard2 framework Gamati et al., 2011.

A data-parallel recurrent task is specified in ArrayOL by *repetitions*. Each repetition is the processing of a sub-array of the input array to produce a sub-output-array. The basic hypothesis is that all the repetitions of a recurrent task are independent and that input sub-arrays processed by different instances of the same recurrent task have the same shape (size), respectively the same hypothesis is assumed for the output data. In order to give all the information needed to create sub-arrays (input and output), a tiler is associated to both input and output arrays. A tiler is able to build the different sub-arrays from an input array, or to store them in an output array. It describes the coordinates of the elements of the tiles from the coordinates of the elements of the patterns. It contains the following information:

- F : a fitting matrix.
- o: the origin of the reference pattern (for the reference repetition).
- P : a paving matrix.

From the tiler, all sub-arrays can extracted by enumerating its other elements relatively to this reference element, the reference element for each sub array is computed by using the paving operating by the mean of paving matrix. The fitting matrix is used to compute the other elements of a sub-array. The coordinates of the elements of the pattern are built as the sum of the coordinates of the reference element and a linear combination of the fitting and an enumeration of the possible vectors. The set of possible vectors is bounded by the sub-array size.

Figure 6.3 is an example of a data parallel task modeled with Array OL. The horizontal filter converts a video stream from size (1920,1080) to (720,1080). As you can notice, the input tiler allows to create input sub-arrays and an output tiler that allows to create output sub-arrays. The shapes of the arrays and patterns are noted on the ports. The repetition space indicating the number of repetitions to be done on each instance of the recurrent task are defined as an multidimensional array. Each dimension of this repetition space can be seen as a parallel loop and the shape of the repetition space gives the bounds of the loop indices of the nested parallel loops. in Figure 6.3, the repetition space is specified on the ports for the input and output.

Array-OL implementations can be easily expressed by our model. To express the data parallelism, we can set the period and the deadline as the time between two arrivals of data (data refreshing). On each vertex of our model, we can map different grains of parallelism. Each vertex can be connected to all other vertices and to itself in order to allow passing from any parallelism grain to another without any restrictions. Hence, we express in such way *malleable parallel tasks* in the same logic as those proposed in Paolillo et al., 2014. We can also express the task parallelism by adding threads of different processings to the same vertex. Array-OL can express also a precedence order between different processings, we express that by adding an edge between two different processings.

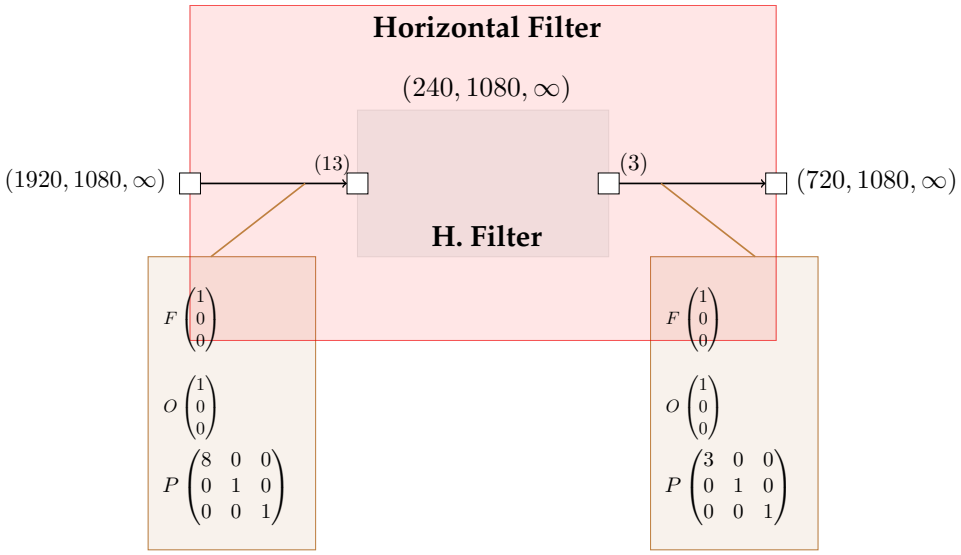


FIGURE 6.3: An Example of video filter with Array OL

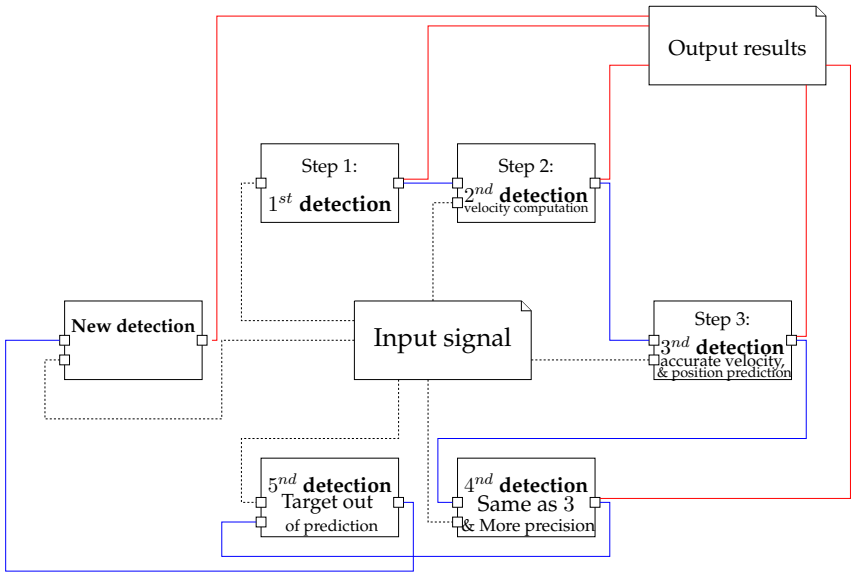


FIGURE 6.4: The radar tracking applications with Array-OL

A typical application that can be modeled by Array-OL implemented by our model is the automatic radar trackers which operates as follows:

1. Target is detected as the received echo exceeds a threshold. There is no information about its velocity.
2. Target is detected again but within the uncertainty boundary. A crude velocity estimate is made and the position where the target will appear next is predicted.
3. The target appears and tracking filters estimates of position and velocity improve, and the next sample prediction is made with a smaller position uncertainty
4. As with (3)
5. The actual target position falls outside the position uncertainty boundary because it has accelerated for example, and track is lost.
6. A new target is detected with unknown velocity.

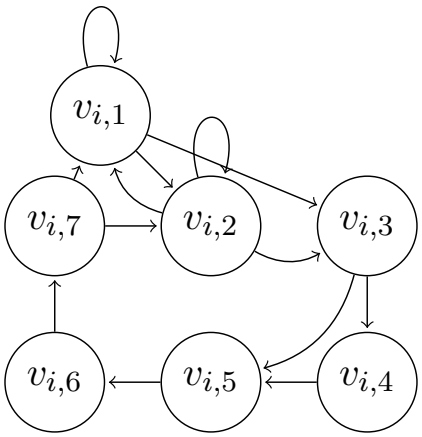


FIGURE 6.5: Radar tracking application with our model

The example of automatic radar trackers can be expressed by Array-OL as in Figure 6.4 (tilers are not specified). Each step of the processing is described by a block, and each block communicates its results with the next block by the edges between blocks and all blocks receive the data from the input stream and may require the results of another processing. Our model can easily express an implementation of the Array-OL model of Figure 6.4 by setting period and deadline to the data refreshing time. Each step of the automatic radar tracking algorithm is a different processing, and it is expressed by a vertex in our model. We may be able to add vertices to express different parallel configurations (different grains of parallelism) for each block of the Array-OL model. In Figure 6.5, we make the choice to allow only the first step to have two parallel configurations, thus $v_{\mathsf{MTI},1}, v_{\mathsf{MTI},2}$ expresses the first step, the two vertices are connected to each other and each one has a self-edge in order to have the ability to choose to stay on its configuration or change it to the other parallel configuration with a different parallelism grain. The j step after the first one is presented with vertices $v_{\mathsf{MTI},j+1}$. Only the finest parallelism grain is presented for these vertices in order to allow decomposing as much as it is possible any block of processing and in order to have a simple example of the task graph.

Our work can easily integrate the synchronization model proposed by Morteza Mohaqeqi and Yi, 2016 to add synchronization between threads.

We claim that parallel real-time task models proposed in the literature (as those proposed by Liu and Layland, 1973b, by Mok and Chen, 1996, by Baruah, 2010, and by Stigge et al., 2011) are not expressive enough to properly express this kind of applications.

6.5 Schedulability analysis

In this work, we consider partitioned Earliest Deadline First (EDF) scheduling. Each core has its own single-processor EDF scheduler and a separate ready-queue. We assume that all threads are independent of each other, whereas threads belonging to the same vertex and running on different cores need to synchronize their activation times and deadlines.

Definition 3. *We denote by $\pi_i^p(t)$ the p^{th} sequence of instances that can be generated by task τ_i in any interval of time of length t. We denote by $\Pi_i(t)$ the set of all paths that can be generated of any interval of time of length t.*

As an example, in the following table (Table 6.2) we report the set $\Pi_i(30)$ for the task of Figure 6.1:

p	$\Pi_i(30)$
$\pi_i^1(30)$	$(v_{i,1}, v_{i,2}, v_{i,4})$
$\pi_i^2(30)$	$(v_{i,1}, v_{i,2}, v_{i,1})$
$\pi_i^3(30)$	$(v_{i,1}, v_{i,3}, v_{i,3})$
$\pi_i^4(30)$	$(v_{i,2}, v_{i,4}, v_{i,4})$
$\pi_i^5(30)$	$(v_{i,2}, v_{i,1}, v_{i,3})$
$\pi_i^6(30)$	$(v_{i,2}, v_{i,1}, v_{i,2})$
$\pi_i^7(30)$	$(v_{i,3}, v_{i,3}, v_{i,3})$
$\pi_i^8(30)$	$(v_{i,4}, v_{i,4}, v_{i,4})$

TABLE 6.2: An example of a path set

6.5.1 Decomposition

Definition 4. *Let τ_i be a task. τ_i', τ_i'' are two tasks with the same period and deadline as τ_i such $\tau_i' = (\mathsf{D}_i, \mathsf{T}_i, \mathsf{G}(\mathsf{V}_i', \mathsf{E}_i'))$,$\tau_i'' = (\mathsf{D}_i, \mathsf{T}_i, \mathsf{G}(\mathsf{V}_i'', \mathsf{E}_i''))$. τ_i', τ_i'' are decomposition of τ_i if and only if :*

$$\forall (i,j) \in \mathbb{N}^2, \quad \begin{aligned} v_{i,j}' \cup v_{i,j}'' &= v_{i,j} \\ v_{i,j}' \cap v_{i,j}'' &= \emptyset \\ \mathsf{E}_i' = \mathsf{E}_i'' &= \mathsf{E}_i \end{aligned}$$

		Decomp (1)		Decomp (2)	
vertex	content	subtask1	subtask 2	subtask1	subtask 2
$v_{i,1}$	2,2,5	2	2,5	2,2	5
$v_{i,2}$	4,3	4,3	$\emptyset$	3	4
$v_{i,3}$	6	$\emptyset$	6	$\emptyset$	6
$v_{i,4}$	2,2,4	2,4	2	2	2,4

TABLE 6.3: The decomposition according to $[val] = 6$ of task of Figure 6.1

Example Let τ_i be a task, task graph of τ_i is described in Figure 6.1. Table 6.3 shows also two possible decompositions of this task *decomp(1)* and *decomp(2)*. *decomp(1)* is a decomposition that verifies the Definition 4. Notice that each thread belong to only one sub task, and both sub-tasks have the same edge's set.
decomp(2) shows another decomposition. In this decomposition, we split a task according to a certain positive value, let it be $[val]$, in such way the execution time of every vertex in one of the two sub-task is equal or less than $[val]$. The second sub-task contains the threads that are

not in the first sub task (the threads that if they are put in the first sub task, they will procreate the break of the constraint that every vertex must have an execution time is equal or less than $[val]$). Notice that every vertices in the sub task in the right on *decomp(2)* have an execution time less or equal to $[val] = 6$. We will show further the importance of this kind of decompositions.

6.5.2 Analysis

Path Demand Function

The path demand function $\mathsf{pdf}(\pi_i^p(t), t)$ denotes the cumulative execution requirement that can be generated by path $\pi_i^p(t)$ over a time interval of length t. Let ω be an integer that denotes the ω^{th} vertex in path $\pi_i^p(t)$ and $\mathcal{C}^{\omega}_{\pi_i^p(t)}$ denotes the execution time of this vertex.

$\mathsf{pdf}(\pi_i^p(t), t)$ can be computed as follows:

$$\mathsf{pdf}(\pi_i^p(t), t) = \sum_{\omega=0}^{1+\left\lfloor \frac{t-\mathsf{D}_i}{\mathsf{T}_i} \right\rfloor} \mathcal{C}^{\omega}_{\pi_i^p(t)} \tag{6.2}$$

Task Demand bound Function

The demand bound function of task τ_i is the maximum path demand function of any time interval of length t, it is denoted by $\mathsf{tdbf}(\tau_i, t)$, and it is computed as:

$$\mathsf{tdbf}(\tau_i, t) = \max\{\mathsf{pdf}(\pi_i^p(t), t), \forall \pi_i^p(t) \in \Pi_i(t)\} \tag{6.3}$$

The demand bound function of a set of tasks $\mathcal{T}_k$ is the sum of the dbf of all its tasks:

$$\mathsf{dbf}(\mathcal{T}_k, t) = \sum_{\tau_i \in \mathcal{T}_k} \mathsf{tdbf}(\tau_i, t) \tag{6.4}$$

Theorem 6. *Task set $\mathcal{T}_k$ is feasible on a uniprocessor if and only if the following condition is verified for all values of t:*

$$\forall t \leq t^*, \mathsf{dbf}(\mathcal{T}_k, t) \leq t, \forall t \in [0, h] \tag{6.5}$$

Proof. Let assume that all the threads of the same vertice are allocated on the same core, thus they will have the same priority, and may run as one single task. Thus, the proof of feasibility can be done in a similar way as in the previous work e.g as in the one proposed of Baruah, 2010 where the proof was done for uniprocessor architectures. As the tasks have only one period, it is sufficient to check the schedulability only on the interval of length from 0 to h, where h is the least common multiple of all periods. □

The feasibility test of Theorem 6 concerns the uniprocessor architectures. In this work, we address the problem of allocation of a set of tasks to a set of identical cores by partitioning. Thus, the problem is simplified to a set of m uniprocessor allocation problem. Hence, the schedulability is checked at the moment of allocation of a task to a core for the task with the tasks already allocated to that core. In contrast with the schedulability analysis done for no parallel systems, if the system is not feasible on a core, we do not seek to allocate the whole task to another core, but we check if it is possible to allocate only a part of the task on the current core.

Our allocation algorithm uses an ordered task queue and an architecture compound of m identical cores. At each iteration, it selects the task in the head of the task queue let it be τ_i, and the highest loaded processor first in the logic of Best Fit bin-packing heuristic, let it be k. Our approach checks the feasibility for the task set already allocated on core k let it be $\mathcal{T}_k$, with task τ_i by checking:

$$\forall t \leq h, \mathsf{dbf}(\mathcal{T}_k \cup \{\tau_i\}, t) \leq t \tag{6.6}$$

If condition (6.6) is verified, task set $\mathcal{T}_k \cup \{\tau_i\}$ is feasible on core k. if Condition (6.6) is not verified, task set $\mathcal{T}_k \cup \{\tau_i\}$ is not feasible on core k but we may be able to allocate a part of task τ_i on the current core by decomposing the execution of the task into two sub-tasks.

Checking the system feasibility comes to check the feasibility of all possible decompositions of all tasks on all cores for all possible operating frequencies. The complexity of an exact feasibility test is high and time consuming. In this section, we will show how to reduce the number of decompositions to be checked and propose a heuristic for threads allocation.

Definition 5. *Let τ_i, τ_i' be two tasks. $\tau_i = \tau_i'$ if and only if they have the same edges, sames vertices, and the same threads.*

Lemma 4. *Let $\mathcal{T}_k$ be a feasible task set and τ_i a task that does not belong to $\mathcal{T}_k$. and $\mathcal{T}_k \cup \{\tau_i\}$ is not feasible. $\exists \tau_i'$ and τ_i'' as a decomposition of τ_i:*
$\tau_i' = (\mathsf{D}_i, \mathsf{T}_i, \mathsf{G}(\mathsf{V}_i', \mathsf{E}_i))$,$\tau_i'' = (\mathsf{D}_i, \mathsf{T}_i, \mathsf{G}(\mathsf{V}_i'', \mathsf{E}_i))$ *and :*
$\mathcal{T}_k \cup \{\tau_i'\}$ *is feasible under EDF.*

Proof. We need just to prove that at least one possible decomposition exists.
Let assume $\tau_i'' = \tau_i$, this imply :

$$\forall i, j, z, \quad \mathcal{C}_{i,j,z}^{\mathsf{th}'} = 0$$

because all vertices of τ_i' are empty thus:

$$\begin{aligned} \forall t, \quad & \mathsf{tdbf}(\tau_i', t) = 0 \\ & \mathsf{dbf}(\mathcal{T}_k \cup \{\tau_i'\}, t) = \mathsf{dbf}(\mathcal{T}_k, t) + \mathsf{tdbf}(\tau_i, t) = \mathsf{dbf}(\mathcal{T}_k, t) \end{aligned}$$

as $\mathcal{T}_k$ is feasible, we have

$$\mathsf{dbf}(\mathcal{T}_k \cup \{\tau_i'\}, t) \leq t \tag{6.7}$$

Thus Lemma 4 is verified. □

Let $\mathcal{T}_k$ be a feasible task set and τ_i a task that does not belong to $\mathcal{T}_k$. If $\mathcal{T}_k \cup \{\tau_i\}$ is not feasible, task τ_i can be decomposed in several decompositions that may verify Lemma 4. Let $|S_i|$ denote the maximum number of thread in any vertex of V_i, and $|\mathsf{V}_i|$ the number of vertices of task τ_i. The number of possible decompositions is

$$\#decompositions = |\mathsf{V}_i| \cdot ({}^1C_{|S_i|} + {}^2C_{|S_i|} + \cdots + {}^{|S_i|-1}C_{|S_i|} + 1)$$

Where ${}^pC_{|S_i|}$ denote the combination of p threads among the maximal number of threads per vertex $|S_i|$. Notice that the number of decomposition is huge and checking all these possible decomposition for all task on all cores can easily cause a combinatorial explosion. Thus, we propose a method that allows extracting only feasible decompositions (See Theorem 7).

Theorem 7. *Let $\mathcal{T}_k$ be a feasible task set, τ_i is a task that does not belong to $\mathcal{T}_k$ and $\mathcal{T}_k \cup \{\tau_i\}$ is not feasible.*
$\exists \alpha \in \mathbb{R}$ *and* $\exists \tau_i', \tau_i''$ *as a decomposition ($\tau_i' = (\mathsf{D}_i, \mathsf{T}_i, \mathsf{G}(\mathsf{V}_i', \mathsf{E}_i'))$) of task τ_i where :*

$$\forall v_{i,j}' \in \mathsf{V}_i', \mathcal{C}_{i,j}^{v_{,}'}(\mathsf{f}_{op}) \leq \alpha \tag{6.8}$$

such $\mathcal{T}_k \cup \{\tau_i''\}$ is feasible and

$$\alpha = \max_{t \in [0,h]} \left\{ \frac{t - \mathsf{dbf}(\mathcal{T}_k, t) - \left\lfloor \frac{t+\mathsf{T}_i-\mathsf{D}_i}{\mathsf{T}_i} \right\rfloor \cdot \max\{\mathcal{C}_{i,j}^{\mathsf{v}}(\mathsf{f}_{op})\}}{\left\lfloor \frac{t+\mathsf{T}_i-\mathsf{D}_i}{\mathsf{T}_i} \right\rfloor} \right\} \tag{6.9}$$

Proof. Let $\mathcal{T}_k$ be a feasible task set, already allocated on core k, and $\tau_i = (\mathsf{D}_i, \mathsf{T}_i, \mathsf{G}(\mathsf{V}_i, \mathsf{E}_i))$ a task that does not belong to $\mathcal{T}_k$. $\{\tau_i\} \cup \mathcal{T}_k$ is not feasible.

Let $\tau_{tmp} = (\mathsf{D}_{tmp}, \mathsf{T}_{\text{tmp}}, \mathsf{G}(\mathsf{V}_{\text{tmp}}, \mathsf{E}_{\text{tmp}}))$ be a task such as $\mathsf{D}_{tmp} = \mathsf{D}_i$ and $\mathsf{T}_{tmp} = \mathsf{T}_i$. Task τ_{tmp} contains only one vertex, with one threads which has an execution time that equals:

$$\mathcal{C}^{\mathsf{v}}_{tmp,1}(\mathsf{f}_{op}) = \max\{\mathcal{C}^{\mathsf{v}}_{i,j}(\mathsf{f}_{op}), \forall j\}$$

The only vertex have a self edges.

Notice that τ_{tmp} is the worst case version of the task τ_i. The dbf of task τ_{tmp} can be computed as in Baruah, Rosier, and Howell, 1990 by :

$$\mathsf{tdbf}(\tau_{tmp}, t) = \left\lfloor \frac{t + \mathsf{T}_{tmp} - \mathsf{D}_{tmp}}{\mathsf{T}_{tmp}} \right\rfloor \cdot \mathcal{C}^{\mathsf{v}}_{tmp,j}(\mathsf{f}_{op}) \tag{6.10}$$

Obviously, τ_{tmp} generate more processor demand than τ_i, hence :

$$\mathsf{tdbf}(\tau_i, t) \leq \mathsf{tdbf}(\tau_{tmp}, t),\ \forall t$$

Thus, the maximum portion of the execution time of task τ_{tmp} that can be allocated on core k, such that the system $\{\tau_{tmp}\} \cup \mathcal{T}_k$ remains feasible is equal or greater than the maximum portion of the execution time of task τ_i that can be allocated on core k such the system $\{\tau_i\} \cup \mathcal{T}_k$ (only the maintained portions) is feasible. We denote this portion as α.

Thus, if we are able to define α for τ_{tmp} we are surely safe for the same α for τ_i.

To abbreviate the equations, and since that $\mathsf{D}_{tmp} = \mathsf{D}_i$ and $\mathsf{T}_{tmp} = \mathsf{T}_i$, we use T_i and D_i instead of those of task τ_{tmp}.

$$\begin{aligned} \mathsf{dbf}(\{\tau_{tmp}\} \cup \mathcal{T}_k, t) &= \mathsf{dbf}(\mathcal{T}_k, t) + \mathsf{tdbf}(\tau_{tmp}, t) \\ &= \mathsf{dbf}(\mathcal{T}_k, t) + \left\lfloor \frac{t + \mathsf{T}_i - \mathsf{D}_i}{\mathsf{T}_i} \right\rfloor \cdot \mathcal{C}^{\mathsf{v}}_{tmp,j}(\mathsf{f}_{op}) \end{aligned} \tag{6.11}$$

Here, we assume that for some values of t, $\mathsf{dbf}(\{\tau_{tmp}\} \cup \mathcal{T}_k, t)$ is greater than t because we assumed that the system is not feasible. Let $\mathcal{C}^{\mathsf{v}}_{tmp,j}(\mathsf{f}_{op}) = \alpha + \beta$ where β refers to the execution time that completes the excess α to achieve all the task τ_{tmp} execution time. α is than the execution time that refers to the excess and β to the portion of time that could be allocated on core k. Thus, we combine β and α with Equation (6.11).

$$\mathsf{dbf}(\{\tau_{tmp}\} \cup \mathcal{T}_k, t) = \mathsf{dbf}(\mathcal{T}_k, t) + \left\lfloor \frac{t + \mathsf{T}_i - \mathsf{D}_i}{\mathsf{T}_i} \right\rfloor (\alpha + \beta) \tag{6.12}$$

To be schedulable, we need only to keep the β execution time of task τ_{tmp}, thus $\beta = \mathcal{C}^{\mathsf{v}}_{tmp,j}(\mathsf{f}_{op}) - \alpha$. In order to reduce the slack time, and increase the processor utilization, we consider the feasibility at extreme case, thus we consider : $\mathsf{dbf}(\{\tau_{tmp}\} \cup \mathcal{T}_k, t) = t$ thus, we have :

$$t = \mathsf{dbf}(\mathcal{T}_k, t) + \left\lfloor \frac{t + \mathsf{T}_i - \mathsf{D}_i}{\mathsf{T}_i} \right\rfloor \cdot (\mathcal{C}^{\mathsf{v}}_{tmp,j}(\mathsf{f}_{op}) - \alpha) \tag{6.13}$$

$$\alpha = \frac{t - \mathsf{dbf}(\mathcal{T}_k, t) - \left\lfloor \frac{t + \mathsf{T}_i - \mathsf{D}_i}{\mathsf{T}_i} \right\rfloor \cdot \mathcal{C}^{\mathsf{v}}_{tmp,j}(\mathsf{f}_{op})}{\left\lfloor \frac{t + \mathsf{T}_i - \mathsf{D}_i}{\mathsf{T}_i} \right\rfloor} \tag{6.14}$$

Since for several values of t we can have different values of α, we need to keep only the maximum excess $\max\{\alpha, \forall\, t\}$ to cover all other excess points, thus :

$$\epsilon = \max_{t\in[0,h]} \left\{ \frac{t - \mathsf{dbf}(\mathcal{T}_k, t) - \left\lfloor \frac{t+\mathsf{T}_i-\mathsf{D}_i}{\mathsf{T}_i} \right\rfloor \cdot \mathcal{C}^{\mathsf{v}}_{tmp,j}(\mathsf{f}_{op})}{\left\lfloor \frac{t+\mathsf{T}_i-\mathsf{D}_i}{\mathsf{T}_i} \right\rfloor} \right\} \tag{6.15}$$

□

Thus, it is necessary to compute α and choose one decomposition of those that verify Theorem 7.

6.6 Heuristics

We propose a greedy algorithm for selecting the frequencies, decompose the task , and allocating the threads on the different cores.

The frequency selection algorithm is simple. It starts from the lowest possible frequency and increase it mode by mode at each iteration till finding a feasible schedule or the maximum frequency is reached.

On every frequency mode, we try to find a feasible allocation. If the system is not feasible under the current frequency. The frequency mode is increased to the next mode. If no feasible schedule is found for all frequencies, the schedule is aborted. The allocation algorithm is described in Algorithm 11

Algorithm 11 Full algorithm

Input: $\mathcal{T}$: *TaskSet*
while ($\mathsf{f}_{op} <= \mathsf{f}_{max}$) **do**
  feasible = allocate($\mathcal{T}$,$multiprocessor$,f_{op});
  if (feasible) **then**
    return true;
  end if
  $\mathsf{f}_{op} = next\_mode()$
end while
return false;

Algorithm **allocate** (Algorithm 12) is described in the next section.

6.6.1 Task decomposition & thread allocation

We proceed now to task decomposition and thread allocation. The goal of this step is to allocate the current task on the current core if it is possible. If it is not possible, we select one possible decomposition according to which the task is decomposed into two sub-tasks. The first is allocated on the current core and the second is put back into the task list. Algorithm 12 performs decomposition and allocation and it consists of two steps:

1. selecting a task, a core and computing α (Theorem 7),
2. task decomposition and threads allocation.

First Step The algorithm uses a queue of tasks that contains the tasks that have not yet been allocated. In the first step, we select the task at the head of the task queue (tasks are in a random order), let it be τ_i, and the first available core in the logic of Best Fit, let it be k. $\mathcal{T}_k$ defines the set of threads already allocated on core k. Now, we proceed in performing the schedulability analysis. The goal of this step is to define the maximum portion of the execution time of task τ_i that can be allocated on core j, such that the system remains feasible (See Theorem 7).

The algorithm **decompose** is described as the third step.

Algorithm 12 allocate

```
for (τ_i ∈ 𝒯) do
    for all cores k in the core list do
        evaluate α
        if α == 0 then
            allocate τ_i on core k,
            proceed_to_the_next_task
        else
            (sub-task_1, sub-task_2) = Decompose(τ_i, α)
            if (∀v_{i,j} ∈ sub-task_1, v_{i,j}! = ∅) then
                allocate sub-task_1 on core k
                put back sub-task_2 in the task list
                proceed_to_the_next_task
            end if
        end if
    end for
    if (if no-allocation found for τ_i) then
        return false;
    end if
end for
return false;
```

Second step Now that α had been computed, we use a simple heuristic to decompose task graph. We denote by C^{max} the maximum execution time of a thread among all vertices of current task. According to the value of α and C^{max}, only 3 cases need to be checked:

1. $\alpha = 0$, it means that $\mathcal{T}_k \cup \{\tau_i\}$ is feasible on the core k. Thus, all threads of all vertices are allocated on the same core, and the every vertex is run in a sequential way.

2. α is equal to C^{max}. This means that there is no space for any part of the task to be allocated on the current core. In other words, not even the smallest thread in all the task set can be allocated to current core k. Thus, we seek to allocate threads on the next core $j + 1$.

3. α is greater than zero, and less than C^{max}. The system $\mathcal{T}_k \cup \{\tau_i\}$ is not feasible on core k but a part of a task may be feasible with $\mathcal{T}_k$ on the current core. In this case, the task is decomposed according to α.

According to the value of excess, we decompose a task into one decomposition as in Definition 4. Here, the decomposition is constrained by excess. The first sub-task is a task where execution time of every vertex is less than α, this task is put back to the task list, to be allocated in the next iterations. The other sub task which is constituted by the rest of threads on each vertex of the first task, and it is allocated on the current core. Algorithm 11 combines all the heuristics described in this section. The complexity of this algorithm is $\theta(\#\mathsf{Modes} \times m \times n \times \max\{|\mathsf{V}_i|\} \times h))$ where $\#\mathsf{Modes}$ is the number of modes available on our architecture.

6.7 Results and Discussions

To evaluate our approach, we apply our heuristics on a large number of randomly generated synthetic task sets. For the experiments, we modeled an architecture composed of 4 cores. Cores share the same frequency modes and can operate all at the same frequency in 13 modes starting from 200MHz, to 1.4 GHz by step of 100MHz. In this section, we refer to the ratio of the operating frequency by the maximal frequency as *speed*.

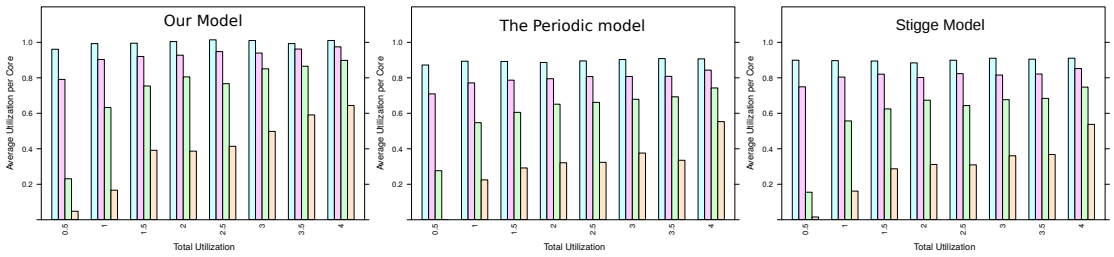


FIGURE 6.6: 1st Scenario: Schedulability Rate

6.7.1 Task Generation

The UUniFast-Discard Emberson, Stafford, and Davis, 2010 algorithm is used to generate n utilization factors, each one bounded by 1 whose sum is given number bounded by m the number of cores in our platform. We generate the number of vertices of every task randomly between 2 and 5. We define a probability p that expresses the chance to have an edge between two vertices. Of course, we ensure that all vertices of the same task are connected. If they are not, edges are added to link the isolated vertices to insure full connectivity of every task graph. The period of every task is generated using the algorithm proposed by Goossens and Macq, 2001. The deadline is generated as a random value between $0.75 \cdot \mathsf{T}_i$ and T_i.

For every vertex, we generate the vertex utilization by randomly varying the task utilization in an interval of -0.1 and +0.1 of the task utilization in the first scenario, and -0.3 and +0.3 in the second one. After, we choose a random number of threads between 1 and 4. We apply UUniFast again on the vertex utilization to generate the utilization of threads. We inflate the vertex utilization by a cost per each thread to reproduce the effect of thread creation, termination and synchronization.

Finally, to generate each thread execution time, we generate a random probability M_p in the interval $[0.05, 0.1]$ which represents the rate of memory access. This parameter is used to generate mt and ct as described in section 6.3.2. Thus, mt, ct are generated as follows:

$$\begin{aligned} \mathsf{mt} &= T \cdot u_{th} \cdot M_p \\ \mathsf{ct}(\mathsf{f}_{max}) &= T \cdot u_{th} \cdot (1 - M_p) \end{aligned}$$

Where u_{th} is the generated utilization per each thread.

Algorithm 13 describes the task set generation. The parameters are:

- maxvarU: this parameter describes the maximum variability that can occur on a task utilization ,
- connectibityG: this parameter represents the chance to have an arc between two vertices,
- nminvertices, nmaxvertices: these paremeters bound the number of vertices,
- nminthreads, nmaxthreads: as the two precedent parameters, these bound the number of threads,
- deadlinePercentage: this parameter describes the variation on the constrained deadline.

Algorithm 13 Task Set generation

input: NumberOfexprementsPerU, maxvarU, connectibityG, nminvertices, nmaxvertices,nminthreads, nmaxthreads, deadlinePercentage
u_{array} =generate_utilizations_by_UUNIFASTM(n, U, nbrproc);
for $(\tau_i \in \mathcal{T})$ **do**
  T = Generate_Period()
  D = RandomBitween((deadlinePercentage * T).toInt, T);
  nvertices = RandomBitween(nminvertices, nmaxvertices)
  for $v_{i,j} \in \mathsf{V}_i$ **do**
    var threads: List[Thread] = Nil
    val u_v = (math.random * (u_n(i) - u_n(i) * (1 - maxvarU))) + (u_n(i) * (1 - maxvarU))
    if $(u_v \geq 1)$ **then**
      nthreads = RandomBitween(2, nmaxthreads)
    else
      nthreads= RandomBitween(nminthreads, nmaxthreads)
    end if
    threadsUtilizations = generate_utilizations_by_UUNIFAST(nthreads, u_v, nbrproc)
    for $\forall threads$ **do**
      C = generate_thread_execution_time;
      addThreadToVertice
    end for
    addVertice (j, threads)
  end for
  for $v_{i,j} \in \mathsf{V}_i$ **do**
    for (**do**$\mathsf{V}_i j' \in \mathsf{V}_i$)
      p = (RandomBitween(1, 100) / (100.toDouble))
      if $p \geq connectibityG)$ **then**
        addTheArc($v_{i,j}, v_{i,j'}$)
      end if
    end for
  end for
  if not all_graph_connected **then**
    connect_the_graph
  end if
  tau = Task(i, vertices, arcs, T, D)
  taskset = taskset + temptau
end for

6.7.2 Simulations

Our scheduling algorithm and task model is unique in the sense that partitioning heuristics does not consider this kind of tasks or parallelization. Therefore, in this chapter we try to emphasize the benefit of using our model. Hence, we compare our model with a partitioned scheduling of Stigge and the periodic models. Firstly, we allow only a small variation of the every vertex utilization (± 0.1). Secondly, a larger variation is allowed (± 0.3). This two scenarios are designed to study in the first, the benefit of the parallelization and to emphasize the benefit of using such expressive models in the second scenario.

We vary total utilization from 0.5 to 4 per step of 0.5. For each total utilization, we generate 100 different task sets.

6.7.3 Scenario 1

In this scenario, the variation from a vertex utilization to another is limited to ± 0.1.

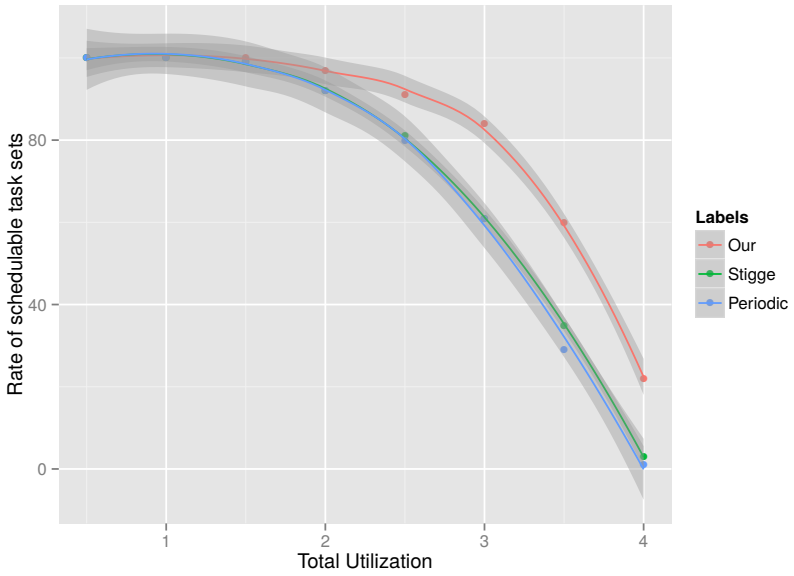


FIGURE 6.7: 1st Scenario: Schedulability Rate

Figure 6.7 reports the schedulability rate. As the variation from each vertex to another is small, Stigge task model overtake just a little bit the periodic model. However, our model still hugely outperforms Stigge model, and the periodic models due to the parallelization features of our algorithm.

Figure 6.6 shows the average utilization per each core as a function of total utilization. On the left, the figure shows the average utilization of our task model, in the middle Stigge model and on the right the periodic task model. Notice that almost all models load cores in the same way because the algorithm used for allocation has the same logic as Best-Fit heuristic (there are processors that are hugely more loaded than others) but our algorithm still has an average utilization per core higher than the two other models. This helps to reduce effectively the static power.

Figure 6.8 presents the maximum, minimum and the mean selected speed for the generated task sets to be schedulable as a function of total utilization for our model, Stigge model and the periodic model. Here we allow the speed to be greater than 1 in order to compare the necessary speed for all the 100 task sets to be schedulable. Notice that for total utilization between 0.5 and 1, all models select the same speed. Starting from total utilization equals to 1.5, our model selects the speed less than Stigge and periodic model, because it allows achieving more load on one core than the other models. This can be noticed also on Figure 6.6 where we can see that the average utilization in our model is greater than Stigge and periodic models.

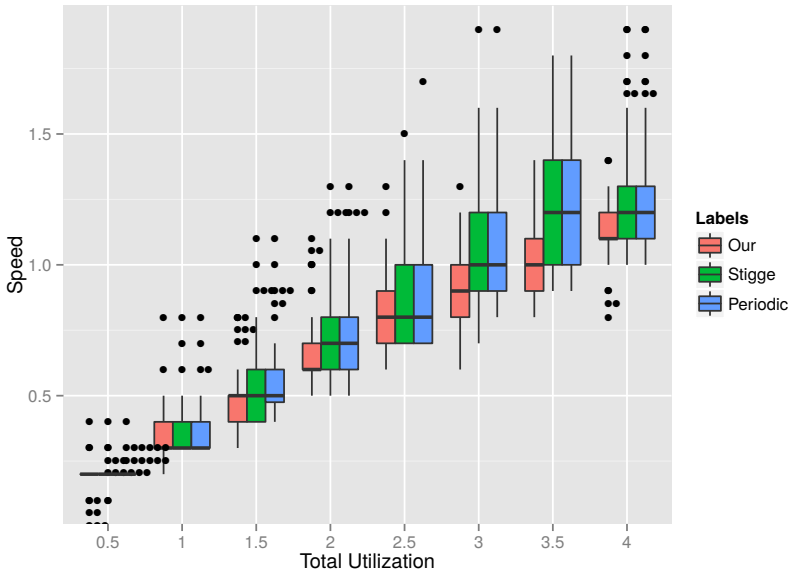


FIGURE 6.8: 1st Scenario: Average Speed (All) as function of total utilization

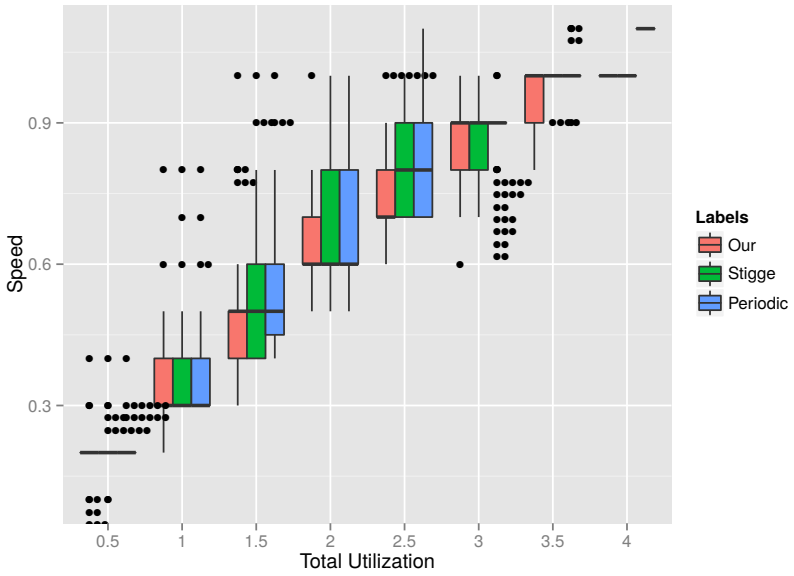


FIGURE 6.9: 1st Scenario: Average Speed (Only Schedulable) as function of total utilization

Figure 6.9 expresses the same results as Figure 6.8, but only for schedulable task sets, thus speed is bounded by 1. Notice here, that our model selects just a little bit lower speed than the other models. However, when total utilization is high, with our model, the scheduling algorithm is able to find feasible scheduling whereas other models does not.

6.7.4 Scenario 2

In this scenario, we vary the vertex utilization to more or less of 0.3 of the task utilization. Figure 6.10 presents the schedulability rate of the 100 task sets as a function of total utilization.

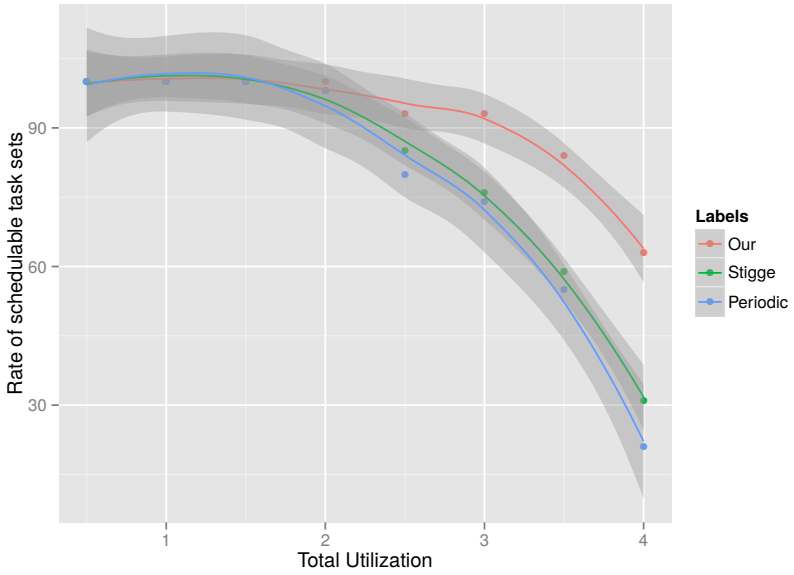


FIGURE 6.10: 2st Scenario: Schedulability Rate

We notice here that Stigge model outperforms periodic model, because it allows tasks to have different utilizations at each instance while the periodic model keep that utilization fixed. Our model still outperforms hugely Stigge model, and keeps the same gap as in the first scenario.

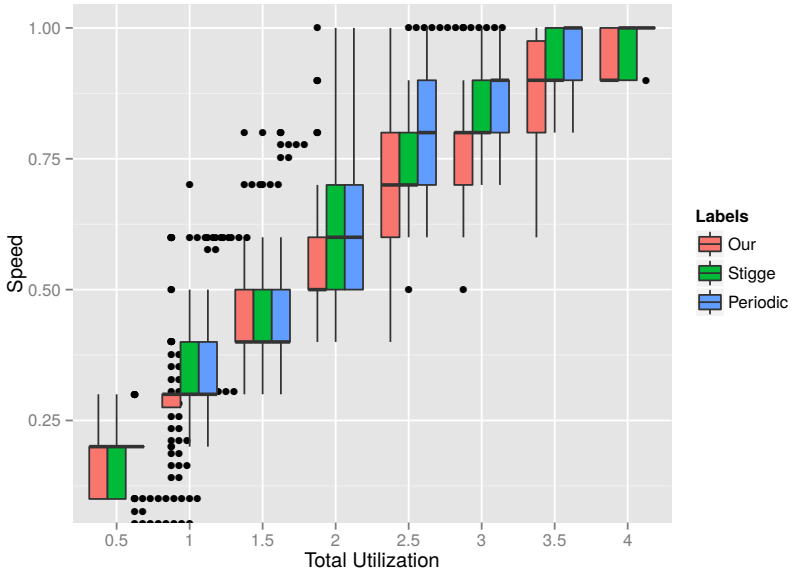


FIGURE 6.11: 2st Scenario: Average Speed (Only Schedulable) as function of total utilization

Figure 6.11 shows the selected speed as a function of the total utilization for the three models. Notice here, that the scheduling algorithm with our model allows to select the speed at worst as Stigge model, both our model and Stigge model outperform the periodic model but ours still outperforms Stigge model for high utilization factors. Figure 6.12 shows the average energy consumption of the three models for this scenario. We use the same energy models proposed in Zahaf et al., 2016a; Zahaf et al., 2016b. As we select a speed lower than all the other

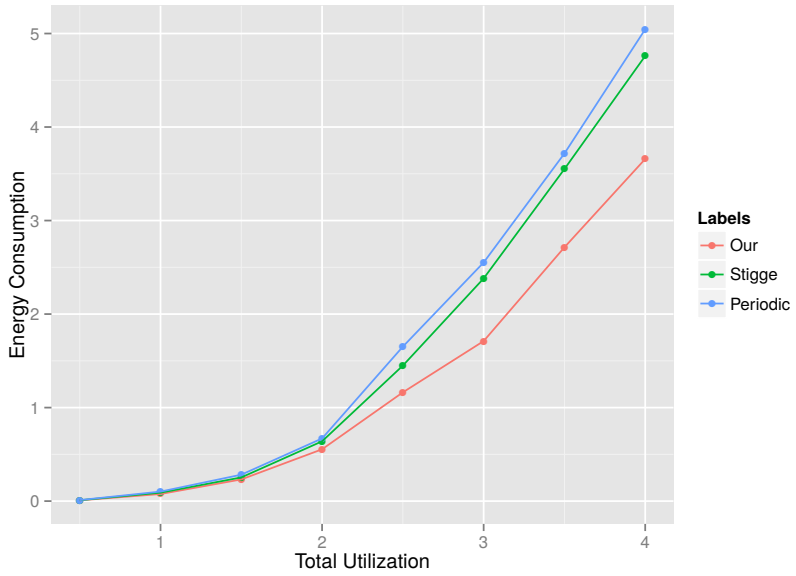


FIGURE 6.12: 2st Scenario: Energy Consumption as a function of total utilization

models, the energy consumption is lower. Therefore, the more the utilization is high, the more our model is efficient in terms of energy consumption.

6.8 Conclusion

In this chapter, we presented an extension of the task model proposed by Stigge et al., 2011 to express the potential intra-task parallelism. In our model, each vertex is a potential decomposition of instances of the task into a set of parallel threads. The task can be run according to a vertex according to the input data, and a set of edges. Thus, the choice of each vertex is not deterministic. We proposed a sufficient feasibility test for partitioned EDF on a set of identical cores. We propose also a heuristic to select the core frequency that can be used to reduce the energy consumption while all deadlines are met. We presented a wide set of experiments that showed that our model and scheduling heuristic are more effective than sequential task models proposed in the literature.

Conclusion & perspectives

Conclusions

In this thesis, we investigate the choices for parallelization and allocation of a set of real-time tasks to different types of multicore platforms. As parallelization consists of decomposing a task into several sub-task that cooperate to solve the same problem, an overhead is assigned to parallel execution. These overheads are mainly due to the sub-task creation, termination and synchronization.

The parallel real-time models proposed recently in the literature do not take into account several realistic parameters such as the parallelization costs, the static/dynamic parallel grain size definition and does not take into account heterogeneous cores in computing platforms such as ARM bigLITTLE. Moreover, only few works consider the parallelization and reducing the energy consumption. Especially because of the slow developement of battery technologies against the increasing energy demand in nowadays systems such as CPS. Hence, In this thesis we had chosen to focus on the limits of the models proposed in the litterature to cope with the nowadays computing needs.

Particularly, in chapter 4 we addressed the problem with a simple moldable task model that does not take into account parallelization costs, but that solves already the problem of the representation of difference parallel grain size for the same task. We proposed two solutions for the problem: an exact solution by modeling the problem as a linear problem and it was solved using lp_solve solver, and an heuristic that by experimentation shown that the obtained results were very close to optimal ones.

In the 5^{th} chapter we first build a realistic timing and energy models for a parallel execution on heterogeneous platforms. Based on different parallelization techniques such those proposed in Cilk or OpenMP. We proposed a task model that takes into account the parallelization costs. The problem consists in allocating a set of task modeled by a realistic parallel model to a set of heterogeneous cores with the ability to calibrate the core frequency and state to reduce the energy consumption. The problem in hand now is very hard and was solved by expressing it as a Mixed Integer Non Linear problem and only small-size problem could be solved using Knitro Solver. Hence, we proposed a heuristic that allow to have quasi-optimal solutions in a very short time.

As the two proposed models where for moldable tasks, and are not expressive enough to cope with the increasing complexity of CPS applications and their dynamic behavior, we proposed in Chapter 6 a methodology to present parallel tasks with di-graphs. The proposed model was very expressive against the models proposed in the literrature. We addressed also the problem of allocating such tasks to a set of identical cores with the ability of setting the core frequency and state. The problem here is very hard, and finding an optimal solution for a very small problems is very computational. Thus we proposed several techniques, methods, and mathematical proofs to reduce the complexity of the problem. The proposed methods where used to propose a heuristic that solves the problem in a reasonable time. The results obtained by testing a very large set of expirements have shown that our model and scheduling heuristics are effective against less expressive models and that it can be used to reduce the energy consumption.

Limitations & perspectives

Shared ressources

In this thesis we addressed the problem of parallelization under the assumption that parallel threads communication waiting time is bounded, and that it is included in the worst case execution time analysis. In the last chapter, the communication between tasks are expressed by the precedence order between different instances of a task. However, this parameter is bounded by the worst case which is very pessimistic. It may be possible to apply shared resources techniques proposed in the litterature of real-time scheduling to reproduce more realistic.

Global scheduling

All our contributions reported in this thesis concern only partitioned scheduling. The basic idea is that for heterogeneous computing platforms, job-level migration is not allowed especially at job migration because the timing analysis of the preempted thread at the preemption point can not be done. However, it may be possible to allow migration of the same thread for the cores of same group.

Dynamic voltage and frequency scaling

When a parallel thread ended its execution shorter than the worst case execution time, a slack time is engendred by these early end. Thus, it may be convenient to recalibrate dynamically the core frequency to allow to have less slack time, and may be save energy.

Personal publications

1. H.E. Zahaf, A.E. Benyamina, R. Olejnik, Giuseppe Lipari, Pierre Boulet “Modeling parallel tasks with di-graphs”, RTNS2016 H.E. Zahaf, 2016c

2. H.E. Zahaf, A.E. Benyamina, R. Olejnik, Giuseppe Lipari “Energy-efficient partitionning for periodic soft Real-Time Tasks on single-ISA heterogeneous Architectures”, under final revision to Journal Of System Architecture H.E. Zahaf, 2016a

3. H.E. Zahaf, R. Olejnik, G. Lipari, A.E Benyamina , ”Modelling the Energy Consumption of Soft Real-Time Tasks on Heterogeneous Computing Architectures”, EEHCO’2016: Energy Efficiency with Heterogenous Computing, Prague, January 15-16, 2016H.E. Zahaf, 2016b

4. H.E. Zahaf, R. Olejnik, G. Lipari, A.E Benyamina , ”Energy-aware parallel tasks scheduling on multicore architectures”, ACACES Workshop’2015: Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems, Fiuggi , July 12-18, 2015 H.E. Zahaf, 2015b

5. H.E. Zahaf, R. Olejnik, G. Lipari, A.E Benyamina , ”Energy-aware moldable real-time task scheduling on uniform architectures”, EDiS’2015: Embedded and Distributed Systems, Oran, November 15-16, 2015 H.E. Zahaf, 2015a

6. H.E. Zahaf, A.E. Benyamina, R. Olejnik ”Intensive Real-Time Task Scheduling on Uniform Multiprocessors”, Models, Optimization, and Mathematical analysis (MOMA), Special Issue IWMCS’2014,ISSN2253-0665(2014), Vol 2, Issue 1, Pages 3-13,H.E. Zahaf, 2014b

7. H.E. Zahaf, A.E. Benyamina, R. Olejnik ”Intensive Real-Time Task Scheduling on Uniform Multiprocessors”,The 2 nd international Workshop on Mathamatics and Computer Science IWMCS’2014, december 1-3, 2014, Tiaret, Algeria.H.E. Zahaf, 2014c

8. H.E. Zahaf, A.E. Benyamina, R. Olejnik ”Energy-Aware Work Load for Real-Time MapReduce Environment”,The 1st doctorial days of LAPECI, Oran, September 28-29, 2014, Oran, Algeria.H.E. Zahaf, 2014a

9. H.E. Zahaf, A.E. Benyamina, R. Olejnik ”Smart cities, Scenarios and Applications (Poster)”, National Exhibition of Valuation Research Result Programmes, April 8-9, 2014, Oran, Algeria.H.E. Zahaf, 2014d

Bibliography

Andersson, Björn, Konstantinos Bletsas, and Sanjoy Baruah (2008). "Scheduling arbitrary-deadline sporadic task systems on multiprocessors". In: *Real-Time Systems Symposium, 2008*. IEEE, pp. 385–394.

Andersson, Björn and Eduardo Tovar (2006). "Multiprocessor scheduling with few preemptions". In: *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)*. IEEE, pp. 322–334.

Aspray, William (2004). *Chasing Moore's Law: Information Technology Policy in the United States*. SciTech Publishing.

Audsley, Neil et al. (1993). "Applying new scheduling theory to static priority pre-emptive scheduling". In: *Software Engineering Journal* 8.5, pp. 284–292.

Audsley, Neil C et al. (1990). *Deadline monotonic scheduling*. Citeseer.

Baheti, Radhakisan and Helen Gill (2011). "Cyber-physical systems". In: *The impact of control technology* 12, pp. 161–166.

Barry, Richard et al. (2008). "FreeRTOS". In: *Internet, Oct*.

Baruah, S. (2010). "The Non-cyclic Recurring Real-Time Task Model". In: *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pp. 173–182. DOI: 10.1109/RTSS.2010.19.

Baruah, Sanjoy and Nathan Fisher (2005). "The partitioned multiprocessor scheduling of sporadic task systems". In: *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*. IEEE, 9–pp.

Baruah, Sanjoy K, Louis E Rosier, and Rodney R Howell (1990). "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor". In: *Real-Time Systems* 2.4, pp. 301–324.

Bini, Enrico, Giorgio Buttazzo, and Giuseppe Lipari (2005). "Speed modulation in energy-aware real-time systems". In: *Real-Time Systems, 2005.(ECRTS 2005). Proceedings. 17th Euromicro Conference on*. IEEE, pp. 3–10.

Bini, Enrico and Giorgio C. Buttazzo (2005). "Measuring the Performance of Schedulability Tests". In: *Real-Time Systems* 30.1, pp. 129–154. ISSN: 1573-1383. DOI: 10.1007/s11241-005-0507-9. URL: http://dx.doi.org/10.1007/s11241-005-0507-9.

Blumofe, Robert D et al. (1996). "Cilk: An efficient multithreaded runtime system". In: *Journal of parallel and distributed computing* 37.1, pp. 55–69.

Bonomi, Flavio et al. (2012). "Fog computing and its role in the internet of things". In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, pp. 13–16.

Boulet, Pierre (2007). *Array-OL Revisited, Multidimensional Intensive Signal Processing Specification*. Research Report RR-6113. INRIA, p. 24. URL: https://hal.inria.fr/inria-00128840 (visited on 07/22/2016).

Burns, Alan and Andrew J Wellings (2001). *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education.

Byrd, Richard H., Jorge Nocedal, and Richard A. Waltz (2006). "KNITRO: An integrated package for nonlinear optimization". In: *Large Scale Nonlinear Optimization, 3559, 2006*. Springer Verlag, pp. 35–59.

Chandra, Rohit (2001). *Parallel programming in OpenMP*. Morgan Kaufmann.

Cisco (2015). *Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are*. White paper. Cisco, p. 6. URL: https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf.

Colin, Alexei, Arvind Kandhalu, and Ragunathan Raj Rajkumar (2015). "Energy-Efficient Allocation of Real-Time Applications onto Single-ISA Heterogeneous Multi-Core Processors". In: *Journal of Signal Processing Systems*, pp. 1–20.

Colin, Antoine and Isabelle Puaut (2000). "Worst case execution time analysis for a processor with branch prediction". In: *Real-Time Systems* 18.2-3, pp. 249–274.

Collette, Sébastien, Liliana Cucu, and Joël Goossens (2008). "Integrating job parallelism in real-time scheduling theory". In: *Information Processing Letters* 106.5, pp. 180–187.

Courbin, Pierre, Irina Lupu, and Jol Goossens (2013). "Scheduling of hard real-time multi-phase multi-thread (MPMT) periodic tasks". en. In: *Real-Time Systems* 49.2, pp. 239–266. ISSN: 0922-6443, 1573-1383. DOI: 10.1007/s11241-012-9173-x. URL: http://link.springer.com/10.1007/s11241-012-9173-x (visited on 12/25/2014).

Davis, R. I. and A. Burns (2009). "Priority Assignment for Global Fixed Priority Pre-Emptive Scheduling in Multiprocessor Real-Time Systems". In: *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pp. 398–409. DOI: 10.1109/RTSS.2009.31.

Davis, Robert I. and Alan Burns (2011). "A Survey of Hard Real-time Scheduling for Multiprocessor Systems". In: *ACM Comput. Surv.* 43.4, 35:1–35:44. ISSN: 0360-0300. DOI: 10.1145/1978802.1978814. URL: http://doi.acm.org/10.1145/1978802.1978814.

Dertouzos, Michael L. (2002). "Control Robotics: The Procedural Control of Physical Processes." In: *IFIP Congress*, pp. 807–813. URL: http://dblp.uni-trier.de/db/conf/ifip/ifip74.html#Dertouzos74.

Drozdowski, M. (2004). *Scheduling Parallel Tasks Algorithms and Complexity, chapter 25. Handbook of SCHEDULING Algorithms, Models and Performance Analysis*. CHAPMAN and HALL/CRC.

Emberson, Paul, Roger Stafford, and Robert I Davis (2010). "Techniques for the synthesis of multiprocessor tasksets". In: *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pp. 6–11.

Feitelson, Dror G and Larry Rudolph (1992). "Gang scheduling performance benefits for fine-grain synchronization". In: *Journal of Parallel and Distributed Computing* 16.4, pp. 306–318.

Fennel, Helmut et al. (2006). "Achievements and exploitation of the AUTOSAR development partnership". In: *Convergence* 2006, p. 10.

Flynn, Michael J (1972). "Some computer organizations and their effectiveness". In: *IEEE transactions on computers* 100.9, pp. 948–960.

Gamati, Abdoulaye et al. (2011). "A Model Driven Design Framework for Massively Parallel Embedded Systems". In: *ACM Transactions on Embedded Computing Systems (TECS)* 10.4. URL: https://hal.inria.fr/inria-00637595 (visited on 07/22/2016).

Garg, Ankita (2009). "Real-time linux kernel scheduler". In: *Linux Journal* 2009.184, p. 2.

Gehringer, Edward F, Daniel P Siewiorek, and Zary Segall (1987). *Parallel processing: the Cm* experience*. Digital Press.

Glitia, Calin and Pierre Boulet (2008). "High Level Loop Transformations for Multidimensional Signal Processing Embedded Applications". In: *International Symposium on Systems, Architectures, MOdeling, and Simulation (SAMOS VIII)*. Samos, Grce. URL: http://hal.inria.fr/inria-00565154/en.

Glitia, Calin, Philippe Dumont, and Pierre Boulet (2010). "Array-OL with delays, a domain specific specification language for multidimensional intensive signal processing". In: *Multidimensional Systems and Signal Processing* 21.2, pp. 105–131. DOI: 10.1007/s11045-009-0085-4. URL: http://dx.doi.org/10.1007/s11045-009-0085-4 (visited on 05/28/2010).

Glitia, Calin et al. (2011). "Repetitive model refactoring strategy for the design space exploration of intensive signal processing applications". In: *Journal of Systems Architecture* 57.9, pp. 815–829. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2010.12.002. URL: http://www.sciencedirect.com/science/article/pii/S1383762110001645 (visited on 07/22/2016).

Goossens, Joël and Vandy Berten (2010). "Gang FTP scheduling of periodic and parallel rigid real-time tasks". In: *arXiv preprint arXiv:1006.2617*.

Goossens, Joël, Shelby Funk, and Sanjoy Baruah (2003). "Priority-driven scheduling of periodic task systems on multiprocessors". In: *Real-time systems* 25.2-3, pp. 187–205.

Goossens, Joel and Christophe Macq (2001). "Limitation of the hyper-period in real-time periodic task set generation". In: *In Proceedings of the RTS Embedded System (RTS01*. Citeseer.

Guan, Nan and Wang Yi (2014). "General and efficient response time analysis for EDF scheduling". In: *Proceedings of the conference on Design, Automation & Test in Europe*. European Design and Automation Association, p. 255.

Guerci, Joseph R (2014). *Space-time adaptive processing for radar*. Artech House.

Guide, VxWorks Programmers (1999). "Wind River Systems". In: *Alameda, Calif*.

Guthaus, Matthew R et al. (2001). "MiBench: A free, commercially representative embedded benchmark suite". In: *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE, pp. 3–14.

Han, C-C and K-J Lin (1989). "Scheduling parallelizable jobs on multiprocessors". In: *Real Time Systems Symposium, 1989., Proceedings.* IEEE, pp. 59–67.

HardKernel (2016). *Odroid C2 datasheet*. URL: http://www.hardkernel.com/main/products/prdt_info.php (visited on 08/14/2016).

H.E. Zahaf R. Olejnik, G. Lipari A.E Benyamina (2015a). "Energy-aware moldable real-time task scheduling on uniform architectures". In: *CEDiS'2015: 1st Algerian Conference Embedded and Distributed Systems, Oran*, pp. 3–13.

H.E. Zahaf A.E. Benyamina, R. Olejnik (2014a). "Energy-Aware Work Load definition for Real-Time MapReduce Environment". In: *The 1st doctorial days of LAPECI, Oran*.

— (2014b). "Intensive Real-Time Task Scheduling on Uniform Multiprocessors". In: *Models, Optimization, and Mathematical analysis (MOMA), Special Issue IWMCS,ISSN2253-0665(2014), Vol 2, Issue 1*, pp. 3–13.

— (2014c). "Intensive Real-Time Task Scheduling on Uniform Multiprocessors". In: *The 2 nd international Workshop on Mathamatics and Computer Science IWMCS'2014*.

— (2014d). "Smart cities, Scenarios and Applications (Poster)". In: *National Exhibition of Valuation Research Result Programmes*.

H.E. Zahaf A.E. Benyamina, R. Olejnik Giuseppe Lipari (2015b). "Energy-aware parallel tasks scheduling on multicore architectures". In: *CACES'2015: Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems*. Hipeac.

— (2016a). "Energy-Efficient Scheduling for Moldable Real-Time Tasks on Heterogeneous Computing Platforms". In: *under revision to Journal Of System Architecture*. Elesevier.

— (2016b). "Modelling the Energy Consumption of Soft Real-Time Tasks on Heterogeneous Computing Architectures". In: *EEHCO'2016: Energy Efficiency with Heterogenous Computing*. Prague.

H.E. Zahaf A.E. Benyamina, R. Olejnik Giuseppe Lipari Pierre Boulet (2016c). "Modeling parallel tasks with di-graphs". In: *submitted to RTNS2016*.

Intel (2016). *Intel i5 datasheet*. URL: http://ark.intel.com/fr/products/69114/Intel-Core-i5-3350P-Processor-6M-Cache-up-to-3_30-GHz (visited on 08/14/2016).

Karnouskos, Stamatis (2011). "Cyber-physical systems in the smartgrid". In: *2011 9th IEEE International Conference on Industrial Informatics*. IEEE, pp. 20–23.

Kato, S. and Y. Ishikawa (2009). "Gang EDF Scheduling of Parallel Task Systems". In: *30th IEEE Real-Time Systems Symposium, 2009, RTSS 2009.* Pp. 459–468. DOI: 10.1109/RTSS.2009.42.

Kim, Junsung et al. (2013). "Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car". In: *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*. ACM, pp. 31–40.

Lakshmanan, K., S. Kato, and R. Rajkumar (2010). "Scheduling Parallel Real-Time Tasks on Multi-core Processors". In: *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pp. 259–268. DOI: 10.1109/RTSS.2010.42.

Lee, Insup et al. (2012). "Challenges and research directions in medical cyber–physical systems". In: *Proceedings of the IEEE* 100.1, pp. 75–90.

Leung, Joseph Y-T and Jennifer Whitehead (1982). "On the complexity of fixed-priority scheduling of periodic, real-time tasks". In: *Performance evaluation* 2.4, pp. 237–250.

Levin, Greg et al. (2010). "DP-FAIR: A simple model for understanding optimal multiprocessor scheduling". In: *2010 22nd Euromicro Conference on Real-Time Systems*. IEEE, pp. 3–13.

Li, Jing et al. (2014). "Analysis of federated and global scheduling for parallel real-time tasks". In: *26th Euromicro Conference on Real-Time Systems (ECRTS 2014)*. IEEE, pp. 85–96.

Liu, Chung Laung and James W Layland (1973a). "Scheduling algorithms for multiprogramming in a hard-real-time environment". In: *Journal of the ACM (JACM)* 20.1, pp. 46–61.

Liu, Chung Laung and James W. Layland (1973b). "Scheduling algorithms for multiprogramming in a hard-real-time environment". In: *Journal of the ACM (JACM)* 20.1, pp. 46–61. URL: http://dl.acm.org/citation.cfm?id=321743 (visited on 07/21/2016).

Mantegazza, Paolo, EL Dozio, and S Papacharalambous (2000). "RTAI: Real time application interface". In: *Linux Journal* 2000.72es, p. 10.

Maxfield, Max (2012). *Achronix new 22nm Speedster22i FPGA*. URL: http://www.embedded.com/electronics-products/electronic-product-reviews/fpga-pld-products/4371597/Achronix-announces-new-22nm-Speedster22i-FPGAs.

Mei, Jing et al. (2013). "Energy-aware preemptive scheduling algorithm for sporadic tasks on DVS platform". en. In: *Microprocessors and Microsystems* 37.1, pp. 99–112. ISSN: 01419331. DOI: 10.1016/j.micpro.2012.11.002. URL: http://linkinghub.elsevier.com/retrieve/pii/S0141933112001895 (visited on 12/25/2014).

Michael, J Quirm (2003). "Parallel Programming in C with MPI and OpenMP". In: *Dubuque, IA: McGraw-Hill*.

Mok, A. K. and D. Chen (1996). "A multiframe model for real-time tasks". In: *Real-Time Systems Symposium, 1996., 17th IEEE*, pp. 22–29. DOI: 10.1109/REAL.1996.563696.

Molnar, Ingo (2009). *Preempt-rt*.

Moore, Gordon (2005). "Excerpts from a conversation with Gordon Moore: Moores Law". In: *Video Transcript, Intel* 54.

Moore, Gordon E (2006). "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp. 114 ff." In: *IEEE Solid-State Circuits Newsletter* 3.20, pp. 33–35.

Morteza Mohaqeqi Jakaria Abdullah, Nan Guan and Wang Yi (2016). "Schedulability Analysis of Synchronous Digraph Real-Time Tasks". In: *Euromicro Conference on Real-Time Systems*.

Mounie, PF. Dutot G. and Denis Trystram M. Drozdowski (2004). *Scheduling Parallel Tasks Approximation Algorithms, chapter 26. Handbook of SCHEDULING Algorithms, Models and Performance Analysis*. CHAPMAN and HALL/CRC.

MPI itemplementations (2016). URL: http://www.mcs.anl.gov/research/projects/mpi/implementations.html (visited on 08/01/2016).

Nichols, Bradford, Dick Buttlar, and Jacqueline Farrell (1996). *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc."

Object Management Group (2009). *UML Profile for MARTE: Modeling and Analysis of Real-time Embedded Systems*. Object Management Group. URL: http://www.omg.org/spec/MARTE/1.0 (visited on 02/23/2011).

OpenMP Architecture Review Board (2008). *OpenMP Application Program Interface Version 3.0*. URL: http://www.openmp.org/mp-documents/spec30.pdf.

Ousterhout, John K (1982). "Scheduling Techniques for Concurrebt Systems." In: *International Conference on Distributed Computing Systems, ICDCS*. Vol. 82, pp. 22–30.

Paolillo, Alfredo et al. (2014). "Power minimization for parallel real-time systems with malleable jobs and homogeneous frequencies". In: *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*. IEEE, pp. 1–10.

Parolini, Luca et al. (2012). "A cyber–physical systems approach to data center modeling and control for energy efficiency". In: *Proceedings of the IEEE* 100.1, pp. 254–268.

Pfister, Greg (2008). *Larrabee vs. Nvidia, MIMD vs. SIMD*. URL: http://perilsofparallel.blogspot.gr/2008/09/larrabee-vs-nvidia-mimd-vs-simd.html.

Pheatt, Chuck (2008). "Intel® threading building blocks". In: *Journal of Computing Sciences in Colleges* 23.4, pp. 298–298.

Puschner, Peter and Alan Burns (2000). "Guest editorial: A review of worst-case execution-time analysis". In: *Real-Time Systems* 18.2, pp. 115–128.

Rajkumar, Ragunathan Raj et al. (2010). "Cyber-physical systems: the next computing revolution". In: *Proceedings of the 47th Design Automation Conference*. ACM, pp. 731–736.

Rao, Lei et al. (2012). "Distributed coordination of internet data centers under multiregional electricity markets". In: *Proceedings of the IEEE* 100.1, pp. 269–282.

Saifullah, Abusayeed et al. (2013). "Multi-core real-time scheduling for generalized parallel task models". en. In: *Real-Time Systems* 49.4, pp. 404–435. ISSN: 0922-6443, 1573-1383. DOI: 10.1007/s11241-012-9166-9. URL: http://link.springer.com/10.1007/s11241-012-9166-9 (visited on 12/25/2014).

Salman, Emre and Qi Qi (2011). "Path specific register design to reduce standby power consumption". In: *Journal of Low Power Electronics and Applications* 1.1, pp. 131–149.

Samsung (2016). *Exynos 5422 Announcement*. URL: http://www.samsung.com/semiconductor/minisite/Exynos/w/solution/mobile_ap/5422/ (visited on 08/14/2016).

Sanjoy Baruah Marko Bertogna, Giorgio Buttazzo (2015). *Multiprocessor Scheduling for Real-Time Systems*.

Seth, Kiran et al. (2006). "Fast: Frequency-aware static timing analysis". In: *ACM Transactions on Embedded Computing Systems (TECS)* 5.1, pp. 200–224.

Spuri, Marco (1996). "Analysis of deadline scheduled real-time systems". PhD thesis. Inria.

Stigge, Martin et al. (2011). "The digraph real-time task model". In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*. IEEE, pp. 71–80.

Ward, James (1994). *Space-time adaptive processing for airborne radar*. Tech. rep. DTIC Document.

Wolf, Marilyn (2012). *Computers as components: principles of embedded computing system design*. Elsevier.

Zahaf, H.E et al. (2016a). "Energy-Efficient Scheduling for Moldable Real-Time Tasks on Heterogeneous Computing Platforms". In: *Under revision for : Journal of Systems Architecture*.

— (2016b). "Modelling the Energy Consumption of Real-Time Tasks on Heterogeneous Computing Architectures". In: *Hipeac, ECCHO*.