KTH Industrial Engineering and Management

Analysis of task scheduling for multi-core embedded systems

Analys av schemaläggning för multikärniga inbyggda system

JOSÉ LUIS GONZÁLEZ-CONDE PÉREZ, MASTER THESIS

Examiner:
Martin Törngren, KTH

Supervisor:
De-Jiu Chen, KTH
Detlef Scholle, XDIN AB
Barbro Claesson, XDIN AB

MMK 2013:49 MDA 462

Acknowledgements

I would like to thank my supervisors Detlef Scholle and Barbro Claesson for giving me the opportunity of doing the Master thesis at XDIN. I appreciate the kindness of Barbro chatting with me in Spanish and the support of Detlef no matter how much time it was required. I want to thank Sebastian, David and the other people at XDIN for the nice environment I lived in during these 20 weeks. I would like to thank the support and guidance of my supervisor at KTH DJ Chen and the help of my examiner Martin Törngren in the last stage of the thesis.

I want to thank very much the other thesis colleagues at XDIN Joanna, Cheuk, Amir, Robin and Tobias. You have done this experience a lot more enriching. I would like to say merci! to my friends from Tyresö Benoit, Perrine, Simon, Audrey, Pierre, Marie-Line, Roberto, Alberto, Iván, Vincent, Olivier, Achour, Maxime, Simon, Emilie, Adelie, Siim and all the others. I have had great memories with you during the first year at KTH. I thank Osman and Tarek for this year in Midsommarkransen.

I thank all the professors and staff from the Mechatronics department Mike, Bengt, Chen, Kalle, Jad and the others for making this programme possible, especially Martin Edin Grimheden for his commitment with the students. I want to thank my friends from Mechatronics Eidur, René, Erik, Joanna, Marcus, Andreas, Mazda, Henrik, Oskar, Daniel and all the others. I would also like to thank other friends at KTH Lars, Hari, Maria, Sofia, Carl-Johan, Magnus, Ali and my tandem Daniel.

I want to thank my friends from Spain Héctor, Javi, Dani, Rubén, Raúl, Silvia, Jesús, Emilio, Carolina, Marga, Belén, Juanjo, David, Luis and all the others because you are a big part of my life.

Finally, I would like to thank the support of my parents Joselé and Maite and my grandparents Domingo and Carmen because it makes me stronger in the difficult times. The love of my sisters Raquel and Cristina reminds me how lucky I am. I want to thank all my family because you have always given me the best.

Abstract

This thesis performs a research on scheduling algorithms for parallel applications. The main focus is their usage on multi-core embedded systems' applications. A parallel application can be described by a directed acyclic graph. A directed acyclic graph is a mathematical model that represents the parallel application as a set of nodes or tasks and a set of edges or communication messages between nodes.

In this thesis scheduling is limited to the management of multiple cores on a multi-core platform for the execution of application tasks. Tasks are mapped onto the cores and their start times are determined afterwards. A toolchain is implemented to develop and schedule parallel applications on a Epiphany E16 developing board, which is a low-cost board with a 16 core chip called Epiphany. The toolchain is limited to the usage of offline scheduling algorithms which compute a schedule before running the application.

The programmer has to draw a directed acyclic graph with the main attributes of the application. The toolchain then generates the code for the target which automatically handles the inter-task communication. Some metrics are established to help evaluate the performance of applications on the target platform, such as the execution time and the energy consumption. Measurements on the Epiphany E16 developing board are performed to estimate the energy consumption of the multi-core chip as a function of the number of idle cores.

A set of 12 directed acyclic graphs are used to verify that the toolchain works correctly. They cover different aspects: join nodes, fork nodes, more than one entry node, more than one exit node, different tasks weights and different communication costs.

A use case is given, the development of a brake-by-wire demonstration platform. The platform aims to use the Epiphany board. Three experiments are performed to analyze the performance of parallel computing for the use case. Three brake-by-wire applications are implemented, one for a single core system and two for a multi-core system. The parallel application scheduled with a list-based algorithm requires 266% more time and 1346% more energy than the serial application. The parallel application scheduled with a task duplication algorithm requires 46% less time and 134% more energy than the serial application.

The toolchain system has proven to be a useful tool for developing parallel applications since it automatically handles the inter-task communication. However, future work can be done to automatize the decomposition of serial applications from the source code. The conclusion is that this communication system is suitable for coarse granularity, where the communication overhead does not affect so much. Task duplication is better to use for fine granularity since inter-core communication is avoided by doing extra computations.

Sammanfattning

Detta examensarbete utför en studie av om schemaläggningsalgoritmer för parallella applikationer. Huvudfokus är deras användning för flerkärniga inbyggda systemapplikationer. En parallell applikation kan beskrivas genom en riktad acyklisk graf. En riktad acyklisk graf är en matematisk modell som representerar den parallella applikationen som en uppsättning av noder, eller uppgifter, och en uppsättning av pilar, eller meddelanden, mellan noder.

I denna uppsats är schemaläggning begränsad till hanteringen av flera kärnor på en multikärnig plattform för genomförandet av applikationens uppgifter. Uppgifter mappas på kärnorna och deras starttider bestäms efteråt. En speciell verktygskedja kallad ett ”toolchain system” har tagits fram för att utveckla och schemalägga parallella applikationer på ett Epiphany E16 kort, vilket är ett billigt kort med ett 16-kärnigt chip som kallas Epiphany. Toolchain systemet är begränsat till användningen av offline schemaläggningsalgoritmer som beräknar ett schema innan du kör programmet.

Programmeraren måste rita en riktad acyklisk graf med de viktigaste attributen. Toolchain systemet genererar därefter kod som automatiskt hanterar kommunikationen mellan uppgifterna. Ett antal prestandamått defineras för att kunna utvärdera applikationer på målplattformen, såsom genomförandetid och energiförbrukning. Mätningar på Epiphany E16 kortet genomförs för att uppskatta energiförbrukningen som en funktion av antalet lediga kärnor.

En uppsättning av 12 riktade acykliska grafer används för att kontrollera att toolchain systemet fungerar korrekt. De täcker olika aspekter: noder som går ihop, noder som går isär, fler än en ingångsnod, fler än en utgångsnod, olika vikter på uppgifterna och olika kommunikationskostnader.

Ett användningsfall ges, utveckling av en brake-by-wire demonstrations plattform. Plattformen syftar till att använda Epiphany kortet. Tre experiment utförs för att analysera resultatet av parallella beräkningar för användningsfallet. Tre brake-by-wire applikationer genomförs, en för ett enda kärnsystem och två för ett multikärnigt system. Den parallella applikationen som var schemalagd med en algoritm baserad på listor kräver 266% mer tid och 1346% mer energi än den seriella applikationen. Den parallella applikationen som var schemalagd med en uppgiftsduplicerings-algoritm kräver 46% mindre tid och 134% mer energi än den seriella applikationen.

Toolchain systemet har visat sig att vara ett användbart verktyg för att utveckla parallella applikationer eftersom det automatiskt hanterar kommunikation mellan uppgifter. Däremot kan framtida arbete göras för att automatisera nedbrytningen av seriella program från källkod. Slutsatsen är att detta kommunikationssystem är lämpligt för grovkorning parallellism, där kommunikationskostnaden inte påverkar lika mycket. Uppgiftsdupliceringen är bättre att använda för finkorning parallellism eftersom kommunikation mellan kärnor undviks genom att göra extra beräkningar.

Contents

List of Figures

List of Tables

List of Abbreviations

ABS Anti-lock Braking System

AEST Absolute Earliest Start Time

ALAP As Late As Possible

ALST Absolute Latest Start Time

AMP Asymmetric MultiProcessing

API Application Programming Interface

ARTEMIS Advanced Research & Technology for EMbedded Intelligence and Systems

ASAP As Soon As Possible

CCR Communication to Computation Ration

CP Critical Path

CPU Central Processing Unit

DAG Directed Acyclic Graph

DL Dynamic Level

DMA Direct Memory Access

DS Dominant Sequence

ECU Electronic Control Unit

EEST Earliest Execution Start Time

ESP Electronic Stability Program

EST Earliest Start Time

FET Finishing Execution Time

FPU Floating-Point Unit

GPU Graphics Processing Unit

GUI Graphical User Interface

HWA HardWare Accelerator

ISR Interrupt Service Routine

ITEA2 Information Technology for European Advancement

IVT Interrupt Vector Table

LST Latest Start Time

MANY MANY-core programming and resource management for high-performance embedded systems

MBAT combined Model-Based Analysis and Testing of embedded systems

MIMD Multiple Instruction, Multiple Data streams

MISD Multiple Instruction, Single Data stream

OS Operating System

PaPP Portable and Predictable Performance on heterogeneous embedded many-cores

RPC Remote Procedure Call

RTOS Real-Time Operating System

SIMD Single Instruction, Multiple Data streams

SISD Single Instruction, Single Data stream

SLR Scheduling Length Ratio

SMP Symmetric MultiProcessing

TCS Traction Control System

TG Topology Graph

Part I

Analytical phase

Chapter 1

Introduction

1.1 Background

The thesis work has been conducted at XDIN AB, an engineering and IT consulting firm. The main customers belong to the energy, telecommunication, manufacturing and automotive industries. The thesis is part of the MANY-core programming and resource management for high-performance embedded systems (MANY), combined Model-Based Analysis and Testing of embedded systems (MBAT) and Portable and Predictable Performance on heterogeneous embedded manycores (PaPP) projects. MANY is hosted by Information Technology for European Advancement (ITEA2). MBAT and PaPP are hosted by Advanced Research & Technology for EMbedded Intelligence and Systems (ARTEMIS) (industry association and joint undertaking in the field of embedded systems).

Traditionally computing architectures have been made up of one processing unit. This paradigm was good enough to cope with the software requirements. Factors such as the growing software complexity and size or the amount of data to be processed demanded processors with higher frequency. Hardware manufactures were able to launch to the market faster processors until a limit was reached.

The dynamic power that a chip consumes is given by the equation $P = ACV^2F$. Where A is the activity factor, C is the switched capacitance, V is the supply voltage and F is the clock frequency. This means that the power consumed by the chip is proportional to the clock frequency. Therefore higher frequency processors are more power greedy.

This is reflected by the fact that Intel canceled Tejas and Jayhawk processors in May 2004 [2]. The power consumption of the chip became prohibitive. Another side effect was the increase of heat. The required architectures to dissipate the heat became too complex and expensive. Frequency scaling was not viable any more.

Chip makers such as Intel and AMD started to develop multi-core architectures. Multi-core architectures are composed of more than one processing unit. The speed up of the application is achieved by parallel computing, a new paradigm in programming. Applications are able to run faster in more energy efficient architectures. However, effective parallel computing has a cost. It requires either more programming expertise or to develop automatic code generation tools [3]. In the first approach the programmer has to specify in the code which parts can be parallelized and on which processing elements they will run. The second approach is to come up with new tools that abstract the programmers from parallelizing the application and from scheduling it onto the platform. The goal is to go for the second approach but more work has to be done by the academy and the industry in order to cope with problems such as hardware heterogeneity.

Embedded systems are often required to be real-time and energy efficient. They are broadly used in space and military applications for executing specific tasks that are time constrained. A wide range of control and signal processing applications in the industry are run on them. They are also found in everyday life such as consumer electronics.

Embedded systems with multi-core processors are growing due to the diversity of applications they can run. Industrial applications with high potential are machine vision, CAD systems, CNC machines, automated test systems and motion control [4]. Some of them do a lot of math computations over a given dataset and can be decomposed into smaller tasks by applying data partitioning. Multi-core platforms are specially appropriate for battery-powered embedded systems. Multiple energy-efficient cores can give the same performance as a powerful core with a smaller budget of energy [5] and [6]. They also offer other benefits such as bounded determinism, dedicated CPU cores, decreased clock jitter, expanded resources for scaling and optimal contention for resources [7].

Parallel computing is a concept to be explored in the embedded systems' world. A variety of new programming languages, compilers and scheduling strategies are arising for parallel computing ([8] and [9]). It is especially interesting the area of scheduling. Scheduling has become a complex issue since there is now more than one processing element. It handles the mapping of application tasks onto the processors and the determination of their start times.

1.2 Problem statement

The first goal of the thesis is to make an in-depth study of the state of the art scheduling algorithms for parallel computing in embedded systems. The second goal is to design and implement a scheduling algorithm based on the previous study to deploy tasks on the target platform for a brake-by-wire application.

The scheduler will be implemented in a Epiphany E16 developing board. The final target is a low cost parallel platform called Parallella from Adapteva. The main components of the architecture are a Xilinx Zynq7000 FPGA and the Epiphany chip. The Xilinx Zynq7000 has a Dual-Core ARM-A9 programmed inside and a bus to communicate to the Epiphany chip. The Epiphany chip is a 16-core co-processor. It contains an e-Mesh Network-on-Chip which connects a 2D array of e-Nodes.

The demonstration platform is made up of one Epiphany E16 developing board among other things. Communication between tasks will be based on message passing. The following questions are of special interest:

- *Which scheduling algorithm is going to be used? Why?*
- *Which metrics are going to be used to measure the performance of the scheduler?*

1.3 System requirements

This section states the requirements that the ideal scheduler should fulfill. These requirements were set up during the whole analytical phase. To set up the requirements it was necessary to have a good understanding of scheduling and parallel computing. Chapter 5 will elaborate and clarify the requirements. The aim of the thesis is to design and implement a system that complies with the following requirements:

REQ_1 The scheduler shall cope with both serial and parallel applications.

REQ_2 The scheduler shall do automatic parallelization of the application. Parallelization shall be transparent to the programmer.

REQ_3 The scheduler shall parallelize tasks as much as possible.

REQ_4 The scheduler shall avoid communication as much as possible.

REQ_5 The scheduler shall provide to an application an end-to-end guarantee. An end-to-end guarantee is a kind of timing requirement that might be given by a control requirement of the application, such as stability or settling time.

REQ_6 The scheduler shall generate a schedule with the minimum execution time.

REQ_7 The scheduler shall be energy efficient.

REQ_8 The scheduler shall be target independent. The algorithm shall be easily portable to other platforms.

Some of the requirements, such as **REQ_3** and **REQ_4**, may actually be in conflict. Section 5.1 will present how this requirements can be fulfilled and to which degree.

1.4 Team goal

There are four master thesis workers at XDIN in the spring of 2013, all working separately but within the same topic. The goal for the team is to in the end combine their knowledge and implement their respective work on the same distributed embedded system. The aim is to implement a brake-by-wire system based on a new middle-ware layer.

1.5 Method

The thesis is split into two main phases: the analytical part and the practical part.

1.5.1 Analytical phase

A study about task scheduling on a multi-core embedded system will be carried out. Documentation about the subject will be read and analyzed. The result of this phase will be a report presenting the state of the art and some conclusions for the next phase.

1.5.2 Practical phase

A design and an implementation of a task scheduler for a multi-core embedded system will be performed based on the knowledge gained in the analytical phase. The system developing method will be SCRUM. The result of this phase will be a demonstration of the system through a Graphical User Interface (GUI) and a report presenting the results of the scheduling algorithm.

1.6 Delimitation

The time limit for the thesis work is 20 weeks in which the analytical phase, the practical phase and the presentation shall be completed. The hardware platform considered for the demonstration is a prototype board from Adapteva. Design, implementation and testing will be done according to XDIN AB standards. This thesis aims to make a design of a toolchain for developing parallel applications and to implement the required features to make a demonstration. This work should serve as a foundation for future Master thesis works.

The scheduling is limited to the management of multiple cores on a multi-core platform for the execution of multiple application tasks. Tasks will be mapped onto the cores and their start times will be determined afterwards. The toolchain is limited to the usage of offline scheduling algorithms which compute a schedule before running the application.

According to the Flynn's taxonomy [10] there are for kinds of computer architectures:

- **Single Instruction, Single Data stream (SISD)**. A single processing element processes a single data stream.

- **Single Instruction, Multiple Data streams (SIMD)**. Multiple processing elements process multiple data streams against a single instruction stream.

- **Multiple Instruction, Single Data stream (MISD)**. Multiple processing elements process a single data stream against multiple instruction streams.

- **Multiple Instruction, Multiple Data streams (MIMD)**. Multiple processing elements process multiple data streams against multiple instruction streams.

The toolchain is limited to MIMD architectures, because it needs multiple cores executing different instructions on different data.

Chapter 2

Use case description and requirements

There are many embedded system applications that could take advantage of a multi-core platform. Brake-by-wire has been decided to be the use case. This chapter provides a detailed description of a brake-by-wire system. Then the requirements of the system will be extracted and discussed.

Some efforts has been done by the academy in order to introduce multi-core platforms in the automotive industry. A modern vehicle can have more than a hundred Electronic Control Units (ECUs) [11] and in the future some of them will be likely replaced by multi-core ECU for high safety and reliability applications [12]. The behavior has to be predictable and thorough timing and schedulability analysis are carried out. The complexity of novel control applications require the use of multi-core processors (see Figure 2.1, taken from [13]).

At the same time the automotive industry is pushing for replacing mechanical, hydraulic or pneumatic transmissions by the drive-by-wire technology. Traditional systems are substituted by electronic controllers coupled to electromechanical actuators and human-machine interfaces. Even though a car equipped with this technology is today more expensive than a traditional car it provides a number of advantages such as weight reduction, space saving, no problems with wear or leakage, shorter response time and reduction of vibrations and noise.

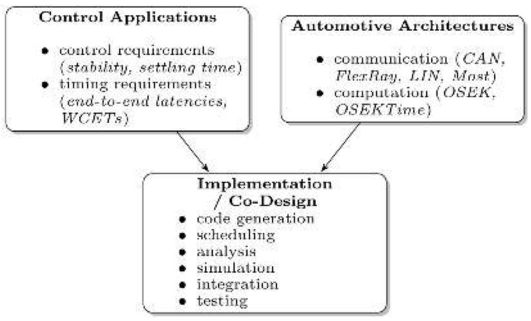
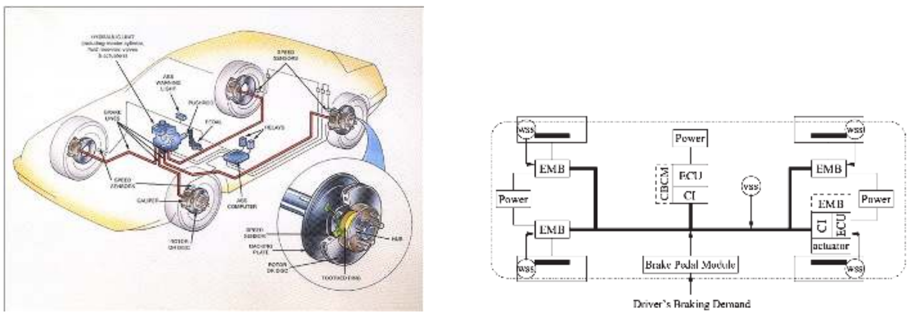

Control Applications
• control requirements (*stability, settling time*)
• timing requirements (*end-to-end latencies, WCETs*)
Automotive Architectures
• communication (*CAN, FlexRay, LIN, Most*)
• computation (*OSEK, OSEKTime*)
Implementation / Co-Design
• code generation
• scheduling
• analysis
• simulation
• integration
• testing

Figure 2.1: Design of automotive control applications



(a) Traditional brake system (b) Brake-by-wire system

Figure 2.2: Brake system technologies

Some ECUs in the vehicle are in charge of providing safety functionalities such as the Anti-lock Braking System (ABS), the Electronic Stability Program (ESP) and the Traction Control System (TCS). They run complex control structures and filters such as Kalman Filters. This kind of algorithms do a lot of math calculations, for instance matrix operations and signal processing. Many of them are subject to be parallelized in order to speed up the performance or reduce the energy consumption. In [14] a study is performed to evaluate the suitability of a multi-core for an ABS system. They compared a TMS470 single core with a TMS570 dual core. The test was performed for three different speeds: 60, 80 and 90 km/h. The stopping time of the dual core outperformed by 180, 130 and 41%, inversely related with the speed of the vehicle. The consumption was bigger for the dual-core due to peripherals such as the CAN interface unit. However, the processor cores consumed much less power. Multi-core platforms for automotive applications is a field where current research is being done ([15], [16], [17], [18], [19], [20], [21] and [22])

2.1 Introduction to brake-by-wire

Brake-by-wire is a system that tries to replace the traditional brake system (see both technologies in Figure 2.2, taken from [23] and [24]). The traditional brake system has been proven to work correctly along the years. The main components are the brake pedal, the master cylinder, the hydraulic circuit, the four calipers and the four brake disks. The mechanism is operated by pressing the brake pedal. The master cylinder then transforms the pressure from the pedal into hydraulic pressure. The hydraulic circuit transmits the pressure to the brake cylinders inside the calipers. The brake cylinders push the brake pads against the brake disk causing a brake force that decelerates the vehicle.

Brake-by-wire aims to substitute the hydraulic mechanism by using electric wires and electric actuators. This approach enhance the already available safety features, such as ABS, ESP, TCS, ACC and so on. There is a reduction in space and weight

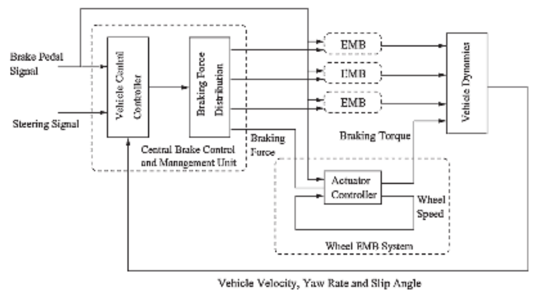

Brake Pedal Signal
Steering Signal
Vehicle Control Controller
Braking Force Distribution
Central Brake Control and Management Unit
Braking Force
EMB
EMB
EMB
Vehicle Dynamics
Braking Torque
Actuator Controller
Wheel Speed
Wheel EMB System
Vehicle Velocity, Yaw Rate and Slip Angle

Figure 2.3: Brake-by-wire control structure

and the weight is easier to distribute. Therefore the fuel efficiency increases. Finally there is more operational accuracy and a lot less maintenance since there are less moving parts and no circuit leakages.

However, traditional brake systems are still cheaper than using brake-by-wire. The reason behind this is that the system is more complex. Exhaustive verifications and validations have to be done before launching a new system to the market. The safety requirements to be met are strict. Redundant hardware components are deployed in order the ensure the fault-tolerant capabilities. Thus the overall process is expensive. The question is whether the users are willing to pay this overprice.

2.2 Use case description

Complex systems cannot be described from only one point of view. The full description of the system is given by a set of views. In the brake-by-wire the control view and physical view are needed to get an overview of the system.

2.2.1 Control view

Several control schemes have been suggested for the brake-by-wire application. A good example is described in [24]. There are three main kinds of control blocks: the vehicle central controller block, the braking force distribution block and four instances of an actuator controller block (one per wheel). The vehicle central controller and the braking force distribution form the central brake control and management unit.

A sensor transforms the pressure in the brake pedal into a electrical signal which is sent to the central ECU. The inertial navigation system is made up of gyroscopes and accelerometers that are used to estimate to state of the vehicle. That information is also fed back to the central ECU. The vehicle central controller then calculates how the vehicle should brake. The braking force distribution module calculates the braking force that should be applied to each wheel. A block diagram control structure is shown in Figure 2.3, taken from [24].

2.2.2 Physical view

The physical view of the system has been presented in Figure 2.2. It describes a distributed system with five nodes: one central ECU and four local ECU located at each wheel. They are connected via a network that runs a hard real-time communication protocol such as FlexRay or TTP/C [25]. The communication has to be deterministic and support fault-tolerant features (redundant buses). There is a communication interface in every node to access the medium. The central brake control and management unit is allocated in the central ECU. Whereas the actuator controllers are allocated in the local ECUs. A wheel plus the electrical actuator plus the local ECU plus the communication interface form the wheel electromechanical brake system.

2.3 Use case requirements

The brake-by-wire system is a safety-critical system. A failure may cause loss of human lives or serious injures. The development of a brake-by-wire system has to comply with regulations such as IEC 61508, ISO 26262, and EC directive 71/320/EEC or UN/ECE Regulations 13. IEC 61508 is an international standard for functional safety of electrical/electronic/programmable electronic safety-related systems. ISO 26262 is an adaptation of IEC 61508 for automotive electric/electronic systems. EC directive 71/320/EEC and UN/ECE Regulations 13 are the conventional braking regulations.

Functional and real-time requirements can be extracted from the regulations. Based on this regulations and the stakeholders the non-functional requirements of our application are going to be defined.

2.3.1 Real-time guarantees

In real-time systems tasks are activated by events. Events come from internal timers, internal interrupts and external interrupts. Tasks have real-time constraints, that are defined by deadlines. There exists relative deadlines (budget of time from the release time of the task) and absolute deadlines (an absolute time point).

Another type of time constraints are end-to-end latencies ([26] and [27]). An end-to-end latency is the required time by a task/message chain. Timing requirements such as deadlines/WCETs and end-to-end latencies are given by control requirements such as stability or the settling time [28].

2.3.2 Fault-tolerance

As it was mentioned before brake-by-wire is a safety critical system [29]. This kind of systems shall still work after a component failure, known as fault-tolerance. This is often achieved by using redundancy at different levels. There are sensor redundancy, signal redundancy an hardware redundancy. When the output of redundant hardware is contradictory a voting algorithm is used to decide.

2.3.3 Energy efficiency

The brake-by-wire system should be energy efficient for economical and environmental reasons. If we take into account the whole life of the car and multiply by the number of sold cars, it can make a difference. Multi-core platforms should be investigated as a mean to reduce the energy consumption [30].

Another reason is to decrease the amount of heat generated, which is proportional to the power consumed. Heat increases failure rate of electronic components and shortens their lifespan [31].

2.4 Summary

The brake-by-wire technology will be deployed at large scale in future vehicles. A brake-by-wire system has to comply with international standards and regulations to prove that it provides real-time guarantees and fault-tolerant capabilities.

Chapter 3

Parallel computing

This chapter provides the basic concepts of parallel computing and an overall overview of the development process of a parallel application.

3.1 Introduction to parallel computing

In the early days of computer science computations were performed sequentially. There were mainly two reasons. First, there was only one processing element. Second, programs were not thought to run in parallel. The situation has changed over the years. Nowadays multiprocessor platforms and distributed systems surround us. Exploiting all their power has become a challenge.

Parallel systems have potential to be more energy-efficient than a powerful single-core system. Reducing the frequency by two the power is reduced by eight and the energy by four. If we assume perfect level of parallelization, two processors running at half the frequency can substitute a normal processor. Half of the energy has been consumed for the same amount of work.

The decrease of energy consumption is not free. Serial applications need to be modified and Operating Systems (OSs) have to provide new services. Contention of resources such as the memory or the communication network can degrade the performance.

Parallel computing is an extensive field and there is no book covering all the details. A classic reference book on the subject is [32]. It covers all the basics although other alternatives are recommended, such as [33]. Books that are also worth a look are [34], [35] and [36]. They cover some of the related technology.

3.1.1 Parallel system

A parallel system is a system composed of two or more processing elements. When the processing elements are located in the same platform, the system is called multi-core platform (a few processing elements) or many-core platform. The processing elements are connected via a network that usually is very fast. If the processing elements are located in different platforms connected via a network, the system is called distributed system. For instance, the brake-by-wire application runs over a five node distributed system (a central ECU and four local ECU). When all the processing units are the same, the system is homogeneous. Otherwise the system is heterogeneous.

In parallel computing there are two ways to model a parallel system: the classic model and the contention aware model. They are presented below.

Classic model

The system is composed of P processors connected via a network. Full connection between processors is provided. Communication and computation can be performed concurrently thanks to a dedicated communication system.

Contention aware model

The topology of the system and its communication is represented by a Topology Graph (TG)= (P, L). Processors P are modelled by nodes and communication links L by edges [37]. This model is closer to reality at expense of more complexity.

3.2 Task models

A task model is a description of the kind of tasks running in the system and their attributes. This section introduces two task models: a task model for independent tasks and a task model for interdependent tasks with communication costs. The first model is used for real-time applications, whereas the second model is used in parallel computing applications. The algorithms presented in the next chapter will use the second model.

3.2.1 Task model for independent tasks

This model is used for real-time systems. In real-time systems tasks have to be executed with both logical correctness and temporal correctness. This model does not consider interdependencies between tasks such as communications.

Task τ_i	Set of instructions that have to be executed in sequence.
Task set τ	Set of tasks τ_i that must be executed on the system.
Job J	Execution of a task.

Classification of tasks according to their releasing pattern:

- **Periodic task:** A new job is released every period.

- **Sporadic task:** The time between two consecutive jobs is equal or greater to the minimum inter-arrival time.

- **Aperiodic task:** An aperiodic task does not have a period or minimum inter-arrival time. They usually model non real-time tasks.

Real-time tasks are defined by a set of attributes:

Release time r_i	Time when a task is asked to be executed.
Period T_i	Fixed amount of time between two consecutive releases of a periodic task.
Minimum inter-arrival time T_i	Minimum amount of time between two consecutive releases of a sporadic task.
Worst-case execution time C_i	It is the maximum execution time of a job. It can vary in function of the structure of the program, the hardware platform, Real-Time Operating System (RTOS) and its scheduling policy, and state of the system and data to be treated.
Deadline D_i	Time constrain that defines when a job must be completed.

3.2.2 Task model for interdependent tasks with communication costs

A parallel application is a set of tasks in which two or more can be executed in parallel. Often the tasks present dependencies among each other that prevent them to execute in parallel. This dependencies can be precedence constraints, intertask data dependencies and exclusive relations. A parallel application can be modeled by a DAG. A DAG is a mathematical tool that is used in many parallel application scheduling algorithms (see Figure 3.1). The first number in a node is the node id and the second one is its weight. The number on the edges are the communication costs.

Directed acyclic graph G(V, E, C, W)

A DAG is used when the characteristics of an application are known a priori (static knowledge). Nodes represent the application tasks and edges the precedence constraints and inter-task data dependencies. Some typical topologies are: out-tree, in-tree, fork, join, fork-join, series-parallel and mixes of the previous ones. A DAG is described by the following attributes [1]:

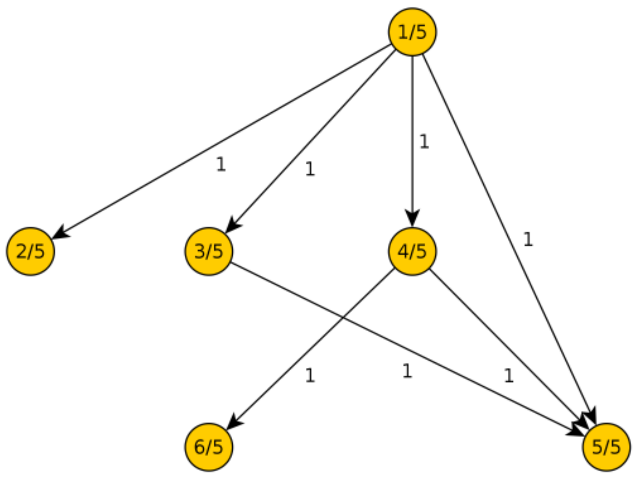


Figure 3.1: A sample DAG

v_i	Node. A node without a parent is and entry node and a node without a child is an exit node.
V	Set of v_i nodes.
w_i	Execution time of v_i.
W	Set of w_i execution times.
$e_{i,j}$	Communication edge from the parent node v_i to the child node v_j. v_j cannot start until v_i is finished.
E	Set of $e_{i,j}$ communication edges.
$suc(v_i)$	Successors of v_i. Set of v_j nodes such there exists $e_{i,j}$.
$pred(v_i)$	Predecessors of v_i. Set of v_j nodes such there exists $e_{j,i}$.
$c_{i,j}$	Communication cost associated to $e_{i,j}$. Its value is given by $c_{i,j} = S + \mu_{i,j}/R$.
S	Cost of starting the communication.
$\mu_{i,j}$	Amount of data transmitted from v_i to v_j.
R	Speed of the communication channel.
C	Set of $c_{i,j}$ communication costs.

Some parameters will now be defined:

- $t-level(v_i)$ is the length of the longest path from the top to v_i (excluding v_i).

$$t-level(v_i) = \max_{v_m \in pred(v_i)} \{t-level(v_m) + w_m + c_{m,i}\} \tag{3.1}$$

- $b-level(v_i)$ is the length of the longest path from v_i to the bottom.

$$b-level(v_i) = w_i + \max_{v_m \in succ(v_i)} \{b-level(v_m) + c_{i,m}\} \tag{3.2}$$

- $static\, b-level(v_i)$ is the same as the $b-level(v_i)$ but without the communication costs.

$$static\, b-level(v_i) = w_i + \max_{v_m \in succ(v_i)} \{static\, b-level(v_m)\} \tag{3.3}$$

- As Soon As Possible (ASAP) or Earliest Start Time (EST) of v_i to be started.

$$EST(v_i) = \max_{v_m \in pred(v_i)} \{EST(v_m) + w_m + c_{m,i}\} \tag{3.4}$$

- As Late As Possible (ALAP) or Latest Start Time (LST) of v_i to be started.

$$LST(v_i) = \min_{v_m \in suc(v_i)} \{LST(v_m) - c_{i,m}\} - w_i \tag{3.5}$$

- Critical Path (CP) is the longest path from the entry node to the exit node.
- Earliest Execution Start Time (EEST) of v_i to be started, being v_i a ready node. A ready node is a node with all its parents completed.
- Dynamic Level (DL) is the difference between the $static\, b-level(v_i)$ and the $EEST(v_i)$.
- Finishing Execution Time (FET) of v_i.
- Communication to Computation Ration (CCR) measures the importance of communication versus the computation.

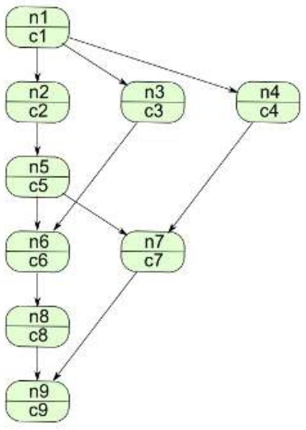
$$CCR = \frac{\sum\limits_{c_{i,j} \in C} c_{i,j}}{\sum\limits_{w_i \in W} w_i} \tag{3.6}$$

3.3 Examples of parallel applications

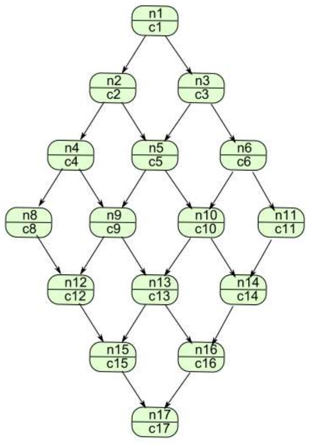
In this section some well known algorithms will be introduced (see Figure 3.2, taken from [38]). They have one thing in common, they have been adapted to make use of parallel systems. This kind of algorithms are very suitable for embedded system applications. They are fed with some input data which is processed and produce an output. Processing the data involves a lot of small and easy operations that can be parallelized.

3.3.1 LU decomposition graph

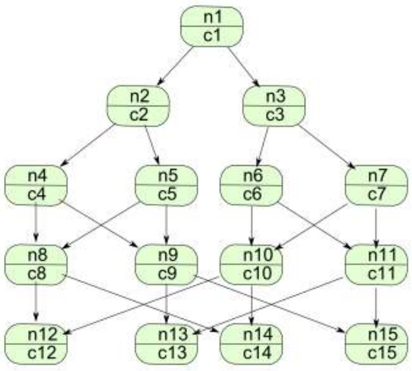
LU decomposition has three main applications. They are solving systems of linear equations, inverting a matrix and computing a determinant.



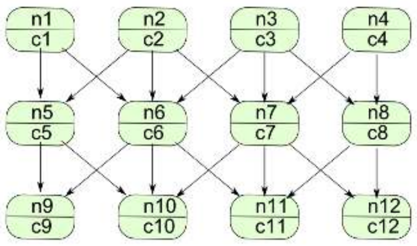
(a) LU decomposition graph



(b) Laplace algorithm graph



(c) FFT algorithm graph



(d) Stencil algorithm graph

Figure 3.2: Parallel algorithms

3.3.2 Laplace algorithm graph

Laplace algorithm is used to solve differential equations and to solve circuit analysis problems.

3.3.3 FFT algorithm graph

The FFT algorithm is a key tool in the field of signal processing. There is a wide variety of applications, especially for sound and image processing. Audio signal processing is performed in phones, sound synthesizers and audio players. Image processing has been developed a lot but still has a huge potential. Medical imaging or cameras in the automotive industry are some examples. It can also be applied to solve partial differential equations or perform quick multiplications of large integers.

3.3.4 Stencil algorithm graph

Stencil algorithm is used for linear and non-linear image processing operations, such as linear convolution and non-linear noise reduction. It also computes explicit solutions to partial differential equations. Other application which is out of our scope is the simulation and seismic reconstruction.

3.4 Application decomposition and dependency analysis

Application decomposition is the process of dividing the application into smaller tasks that can be run in parallel. How this decomposition is done has a hard relation with the performance of the parallelized application. In this process it is very important the size of the partitions, which is called granularity. Fine granularity has high parallelization but high communication overhead. Coarse granularity has low communication overhead but low parallelization. It is necessary to find the right balance.

Manual decomposition. It is more error prone. The code is usually divided into tasks that communicate each other by means of message passing. Communication is used to synchronize or transfer data.

Automatic decomposition. The desire of the industry is to automatize the process by having an intermediate tool. The tool shall abstract the developer so that he does his job as it was a serial program. All the applications in the automotive industry has been developed for single core ECUs. The manual migration of the code to multi-core platforms is very costly and error prone as it is stated in [21].

There are some systematized ways of doing the decomposition [32]. A general classification is shown below:

- **Data decomposition.** When the amount of data to process is large. First the data is partitioned. Second the application is partitioned according to the data. Either input, intermediate or output data is partitioned. The application is partitioned into tasks that are in charge of a specific block of data. They are intensively used in scientific applications with high load of mathematical operations.

- **Recursive decomposition.** When the problem that has to be solved can be split into subproblems (divide-and-conquer). Some examples are quicksort, finding the minimum.

- **Exploratory decomposition.** When the problem to solve is a search of a space of solutions, normally represented by a tree.

- **Speculative decomposition.** When the next step is one of many possible actions, and it is only known when the current task ends. This decomposition assumes it is known and executes some of the next steps.

- **Hybrid decomposition.** The aforementioned strategies can be combined together.

Depending on the result of the decomposition, some dependencies between tasks emerge. For instance, precedence constraints are represented by edges in the DAG. They fall into two families, data dependencies and control dependencies [39]. Control dependencies can be transformed into data dependencies for doing analysis.

An example of data dependencies is shown in the following sample program, $x = a * 2 + a * 3$. b and c depend on a, whereas x depends on b and c.

```
a = 1;
b = a * 2;
c = a * 3;
x = b + c;
```

In control dependencies there is not transference of data. They are produced by control statement such as if-else statements.

```
if (a = 1) {
b = a * 2;
} else {
b = a * 3;
}
```

3.5 Task scheduling

In a serial application task scheduling consists in determining the start time of the task/application. If another application with higher priority becomes ready for execution and there is a preemptive scheduler, the running application is preempted. The preempted application continues the execution when the one with highest priority is finished or blocked. For parallel computing we are not going to consider preemptive scheduling because of its complexity. Besides an increase of performance is not very clear since there is overhead. In parallel applications as in serial applications the scheduler determines the start time of the tasks. In order to do that, tasks first need to be mapped onto the processing elements.

Static scheduling. The processing element selection and the start times are decided at compile time. The advantage of this approach is that the scheduling overhead is low. The disadvantage is that is very rigid and can not take advantage of the run-time state of the system.

Dynamic scheduling. Processing element selection is done at run-time. Load balancing enhance the performance of the system. This approach is used with independent tasks.

3.5.1 Task mapping onto processing elements

Tasks are grouped onto processing elements in a special way. The goal is to reduce the makespan of the application. There is a trade-off between parallelization and inter-task communication. If all tasks are grouped onto the same processing element, the parallel application runs as a serial program. It is not an optimal solution. On the other hand, if all tasks are executed in different processing elements, the inter-task communication can slow down the execution. This is due to the fact that when two tasks are executed on the same processing element the cost of the communication becomes zero. The optimal solution is a balance between parallelization and inter-task communication.

3.5.2 Task temporal arrangement

Once the tasks are located onto their specific processing element, their start time has to be determined. There are two constraints, precedence constraint and exclusive processor allocation constraint.

Precedence constraint. They are the edges of the DAG. They correspond to data dependencies or control dependencies.

Exclusive processor allocation constraint. Two tasks can not run simultaneously onto the same processor. Let A and B be two tasks allocated onto the same processor. Either A executes before B or B executes before A.

3.5.3 Scheduling metrics

Scheduling metrics are scalar values that give information about the performance of the scheduling algorithm.

Makespan

Makespan is the completion time of a parallel application.

$$makespan = FET(v_{exit}) \tag{3.7}$$

Scheduling length ratio, SLR

The main performance measurement of a scheduler is the makespan of the schedule. This value depends on both the scheduling algorithm and the DAG. A normalization is needed to make comparisons when different DAGs are used.

$$SLR = \frac{makespan}{\sum_{i \in CP} w_i} \tag{3.8}$$

Speedup

Speedup is a ratio that shows the gain of performance by executing tasks in parallel instead of in sequence.

$$Speedup = \frac{makespan_{serial\ execution}}{makespan_{parallel\ execution}} \tag{3.9}$$

3.5.4 Optimality, feasibility and schedulability

A scheduling algorithm for parallel applications is optimal if the makespan is the minimum possible. Feasibility and schedulability are terms applied to task models for independent tasks in real-time applications. They can also be applied to parallel applications in case that deadlines are defined for the whole application or for the individual tasks.

Feasibility

An application is said to be feasible, if there exists a schedule that respects the deadlines.

Schedulability

An application is said to be schedulable by a given algorithm S, if the schedule constructed by S respects all the deadlines.

3.6 Related technology

3.6.1 OpenMP

OpenMP is an Application Programming Interface (API) for multi-platform shared-memory parallel programming in C/C++ and Fortran (see Figure 3.3, taken from [40]). The range of platforms supported includes Unix-based systems and Windows NT systems. OpenMP aims for portability and scalability. These goals are measured by the simplicity and flexibility that OpenMP offers to the developer during the implementation phase. OpenMP is basically a specification for a set of compiler directives, library routines and environment variables.

In the early 90´s there were several vendors providing their specifications for parallel programming shared-memory Symmetric MultiProcessing (SMP) architectures. OpenMP was then born because the industry was claiming for standardization. OpenMP ARB is a non-profit consortium taking care of the development of OpenMP. Consortium partners include AMD, IBM, Intel, Cray, HP, Fujitsu, Nvidia, NEC, Microsoft, Texas Instruments and Oracle Corporation.

The parallel programming model defined by OpenMP can be extended to non shared-memory parallel programming, for instance by using message passing. Typical non shared-memory systems are computer clusters. Two solutions have been

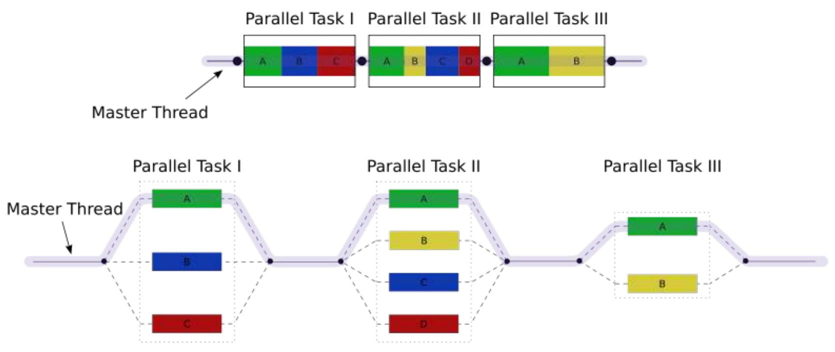


Figure 3.3: OpenMP parallelization

proposed. The first solution stands for the use of MPI. The second one advocates to use new OpenMP extensions.

3.6.2 MPI

It is a standard message passing system that aimed to work in parallel computing applications. The standard currently supports message-passing programs in Fortran, C and C++. Two implementations of the MPI standard will be introduced.

MPICH

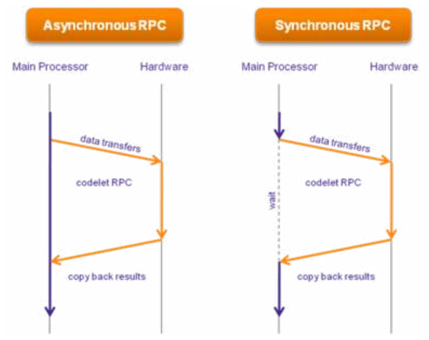
It is a very popular and free implementation of MPI. It is thought for distributed memory applications and is supported in Unix-based systems and Windows OS. There is an implementation of the MPI-2 standard called MPICH2 or simply MPICH and another of the MPI-3.0 standard called MPICH v3.0.

Open MPI

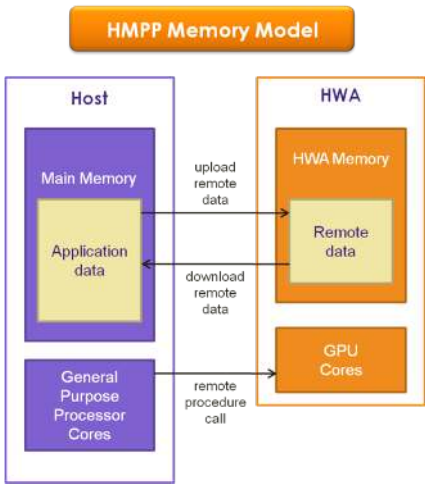
It is other popular open source implementation of MPI. Three previous big MPI projects merged into Open MPI. They are FT-MPI, LA-MPI and LAM/MPI. Many supercomputers are currently using it.

3.6.3 OpenHMPP

OpenHMPP stands for Hybrid Multicore Parallel Programming. CAPS, a Many-Core Programming company, started the project in 2007. It is a set of directives and a compiler that ease the use of HardWare Accelerators (HWAs) such as Graphics Processing Units (GPUs) to the developers. Computations are offloaded to HWAs because their architecture is specialized for parallel computations (see Figure 3.4, taken from [41]). The transference of procedures and data from the general purpose processor to the HWA is hardware dependent. OpenHMPP provides a standard



(a) Synchronous versus asynchronous RPC



(b) OpenHMPP Memory Model

Figure 3.4: OpenHMPP model

API for reducing the complexity. Its directives provide a specification of the Remote Procedure Calls (RPCs) on the HWA.

3.6.4 OpenACC

OpenACC is a set of directives and a compiler to manage HWAs. OpenHMPP is a superset of the OpenACC API with additional features. At the present there are new languages for programming HWAs, such as CUDA and OpenCL. OpenACC basically provides a new layer of abstraction to ease the use of HWAs. It is used to specify loops and parallel regions of the code to be offloaded to the HWA.

3.6.5 CAPS compilers and CodeletFinder

CAPS compilers support the OpenHMPP standard. It allows to build portable applications for many-core platforms, such as Nvidia GPU, AMD GPU and Intel MIC. The workflow of the compiler is described in Figure 3.6, taken from [42] and [43]. CAPS compilers try to partition the code into standalone pieces that can run in parallel, called codelets. The process have two auto-tuning phases. The first one is done offline by a tool called CodeletFinder. It identifies the hotspots and isolates them in order to check if they can be parallelized (see Figure 3.5, taken from [43]). The second phase is done by using machine learning techniques with online profile data.

Performance optimization and portability are conflicting requirements. This approach tackles both problems. OpenHMPP provides the portability and the two step-compiler the performance optimization for a specific target.

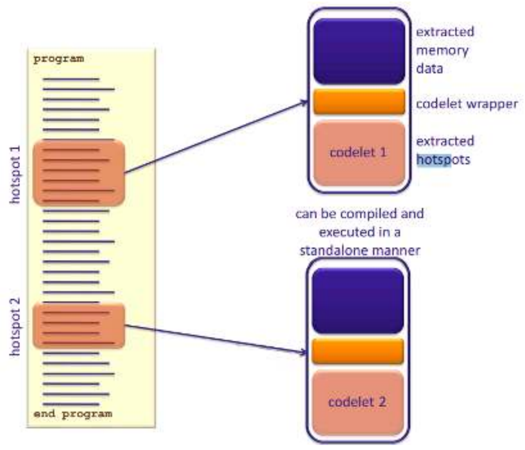
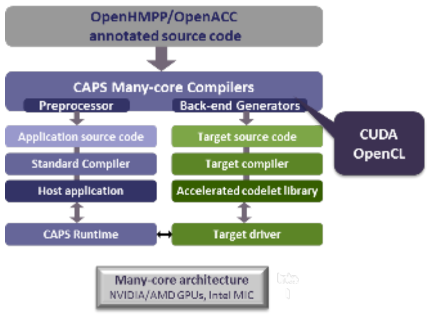
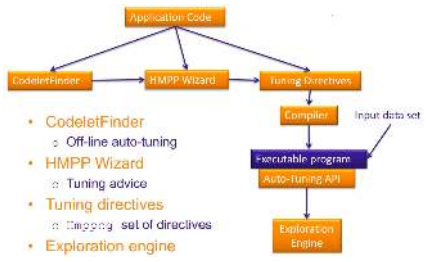


Figure 3.5: CodeletFinder



(a) CAPS compiler model



(b) CAPS compiler workflow

Figure 3.6: CAPS compiler

3.7 Summary

In this section the automatic development of a parallel application will be described. First, the programmer writes the code of the application according to his design. The next step is to compile the code. Here, there is a big difference respect to serial applications. The programmer has placed some directives in the code to help the compiler to partition and schedule the application. The compiler breaks the application into blocks called tasks. New instructions are inserted in the code to manage the transfer of data from one task to another and to synchronize the tasks. All this process is transparent to the programmer. The compiler also makes a static mapping of the tasks onto the processing elements and arranges their temporal execution. The outputs are binary files for the processing elements involved.

Chapter 4

Scheduling for interdependent tasks with communication costs

Research has been performed on the scheduling complexity of DAGs [44]. The different formulations of the scheduling problem have been classified into three complexity classes: P, NP-complete and NP-hard. P contains decision problems that can be solved by a deterministic Turing machine in polynomial time in the input size. NP is a class of decisions problems that can be solved by a non-deterministic Turing machine in polynomial time. NP-complete is a subset of NP, for which no fast solution is known yet. NP-hard problems are decision problems, search problems and optimization problems that are at least as hard as the hardest problems in NP.

The first formulations of the scheduling problem did not consider any communication cost. Scheduling on a limited number of processors was proved to be NP-complete by [45]. However, a polynomial time solution exists for an unlimited number of processors.

Nowadays the formulations consider communication costs in the model. Scheduling on a limited number of processors is NP-hard [46]. The same problem with an unlimited number of processors is still NP-complete [47].

Several tens of algorithms have been developed for scheduling on multi-processor systems. They are based on heuristics and use some assumptions in order to deal with the otherwise NP-hard problem. A taxonomy has been created in order to classify them. More information about them can be found in [48], [49], [50] and [51]. This chapter presents four families of algorithms to schedule a DAG into a multi-core system. Some comparisons done by researchers on the field will be introduced in order to have an overview of their performance. The families that we are talking about are the following.

- **List-based algorithms**: Schedule the tasks onto a limited number of processing elements.
- **Clustering algorithms**: Schedule the tasks onto an unlimited number of processing elements.

- **Arbitrary processor network algorithms**: Schedule the tasks taking into account the network architecture.
- **Duplication algorithms**: Schedule the tasks by duplicating some of them in order to enhance the performance.

4.1 List-based algorithms

List-based algorithms are meant to schedule the nodes of the DAG into a limited number of processors. They are a popular approach for their low complexity and their good results. They give each node a priority and sort them in a list [1]. This family of algorithms can be further divided into two subfamilies.

- **Static list-based algorithms.** Node priorities are computed before scheduling and do not change during the scheduling process.
  - Highest level first with estimated times, HLFET.
  - Modified Critical Path, MCP.
- **Dynamic list-based algorithms.** Node priorities are subject to change during the scheduling process.
  - Earliest time first, ETF.
  - Dynamic Level Scheduling, DLS.
  - Cluster ready Children First, CCF.
  - Hybrid Re-mapper minimum partial completion time static priority.

Some list-based algorithms will be explained in the following. Parameters from subsection 3.2.2 will be used in order to describe them.

4.1.1 Highest level first with estimated times, HLFET

It is a static b-level based algorithm proposed in [52]. It schedules a task to a processor that allows the EST. Scheduling first the nodes with highest b-level gives more priority to the critical path nodes.

The complexity of the algorithm is $O(PV^2)$. This means that the algorithm computes the schedule in polynomial time. The required time is linearly proportional to the number of processors (P) and quadratically proportional to the number of nodes (V). In the following, the complexity of the other algorithms should be interpreted in the same way.

4.1.2 Modified Critical Path, MCP

It is a ALAP based algorithm presented in [53]. The complexity of the algorithm is $O(V^2 log(V))$.

Ranking	**Position 1**	**Position 2**	**Position 3**	**Position 4**
Average makespan	DLS	ETF	HLFET	MCP
Average speedup	DLS	ETF	HLFET	MCP
Average SLR	ETF	DLS	MCP	HLFET
Best results	DLS	ETF	MCP	HLFET
Complexity	HLFET&MCP		DLS&ETF	

Table 4.1: HLFET, MCP, ETF and DLS performance evaluation

4.1.3 Earliest time first, ETF

It is a EEST based algorithm introduced in [54]. Processors are kept as busy as possible. It computes the EEST of all ready nodes and selects the one having the lowest value. The complexity of the algorithm is $O(PV^2)$.

4.1.4 Dynamic Level Scheduling, DLS

It is a DL based scheduling proposed in [55]. It behaves like HLFET in the first steps and like ETF in the last steps of the process. The complexity of the algorithm is $O(PV^3)$.

A performance comparison between HLFET, MCP, ETF and DLS was carried out in [1]. 90k random DAG were generated with CCR values between 0.5 and 2. According to those results the algorithms were ranked, see Table 4.1. The algorithm with the lowest position in the ranking is the one with the best performance. On the other side, the algorithm with the highest position is the one with the worst performance. A one to one comparison was also presented (see Figure 4.1, taken from [1]).

4.1.5 Cluster ready Children First, CCF

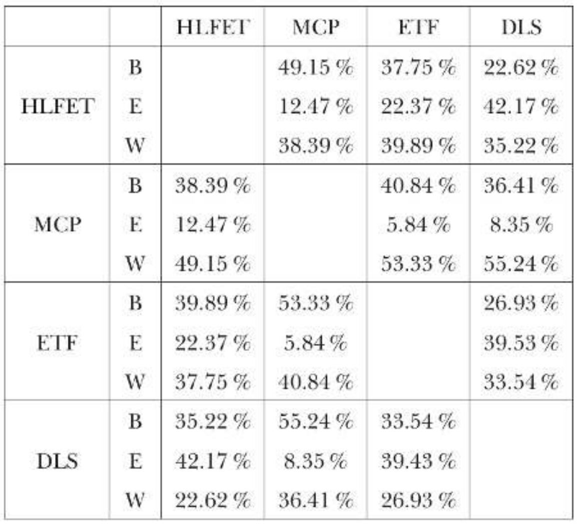
It is a dynamic scheduling algorithm based on lists [56]. The graph is visited in topological order, and tasks are submitted as soon as scheduling decisions are taken.

4.1.6 Hybrid Re-mapper minimum partial completion time Static Priority, Hybrid Re-mapper PS

It is a dynamic list scheduling algorithm for heterogeneous distributed systems [56]. The set of tasks is partitioned into blocks so that tasks in a block do not have any data dependencies among them. Blocks are executed sequentially.

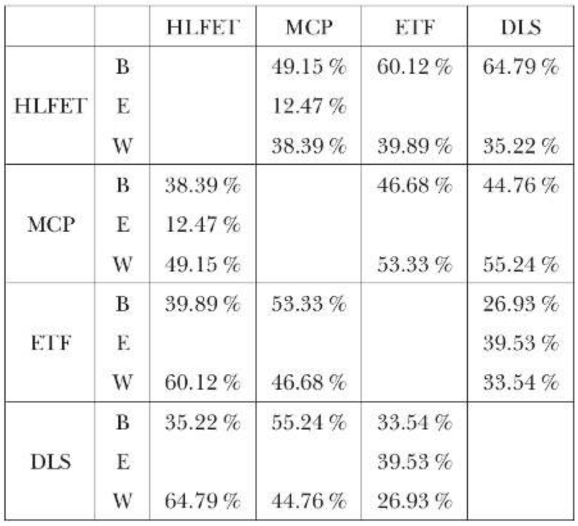
4.2 Clustering algorithms

These algorithms are meant to group the nodes of the DAG to an unbounded number of clusters. A cluster is a virtual processor. Clustering algorithms initially



		HLFET	MCP	ETF	DLS
	B		49.15 %	37.75 %	22.62 %
HLFET	E		12.47 %	22.37 %	42.17 %
	W		38.39 %	39.89 %	35.22 %
	B	38.39 %		40.84 %	36.41 %
MCP	E	12.47 %		5.84 %	8.35 %
	W	49.15 %		53.33 %	55.24 %
	B	39.89 %	53.33 %		26.93 %
ETF	E	22.37 %	5.84 %		39.53 %
	W	37.75 %	40.84 %		33.54 %
	B	35.22 %	55.24 %	33.54 %	
DLS	E	42.17 %	8.35 %	39.43 %	
	W	22.62 %	36.41 %	26.93 %	

(a) Algorithm complexity disregarded



		HLFET	MCP	ETF	DLS
	B		49.15 %	60.12 %	64.79 %
HLFET	E		12.47 %		
	W		38.39 %	39.89 %	35.22 %
	B	38.39 %		46.68 %	44.76 %
MCP	E	12.47 %			
	W	49.15 %		53.33 %	55.24 %
	B	39.89 %	53.33 %		26.93 %
ETF	E				39.53 %
	W	60.12 %	46.68 %		33.54 %
	B	35.22 %	55.24 %	33.54 %	
DLS	E			39.53 %	
	W	64.79 %	44.76 %	26.93 %	

(b) Algorithm complexity regarded

Figure 4.1: List-based scheduling algorithms' comparison, taken from [1]. B-rows stand for better performance than the compared algorithm. E-rows stand for equal performance as the compared algorithm. W-rows stand for worse performance than the compared algorithm.

schedule a node onto a different cluster. Then clusters are merged in an iterative process in order to reduce the makespan of the DAG. This is due to the fact that the edges across two merged clusters are zeroed, since the inter-process communication cost within the same processor is negligible. When clusters belonging to the CP are merged, the completion time of the DAG decreases. But merging has to be done carefully. Nodes inside the same cluster are executed sequentially. Merging may reduce the parallelism of the application and therefore increase the completion time. Optimal scheduling is a trade-off between minimizing inter-processor communication and maximizing the concurrency of the tasks. The goal of clustering algorithms is to reduce the length of the CP as much as possible by merging clusters.

It can happen that the required number of clusters or virtual processors is greater than the number of physical processors. A clustering based scheduling algorithm is a scheduling algorithm that makes use of a clustering algorithm. They schedule clusters into a bounded number of processors. They are comprised of three steps:

- Use a clustering algorithm to find a clustering.
- Map clusters to physical processors.
- Set the start time of each node based on the precedence constraints and the exclusive processor allocation constraints (see Section 3.5).

Precedence constraint. Let e_{ij} and v_j be a communication edge and a node that receives communication data. v_j can not start until e_{ij} has completed.

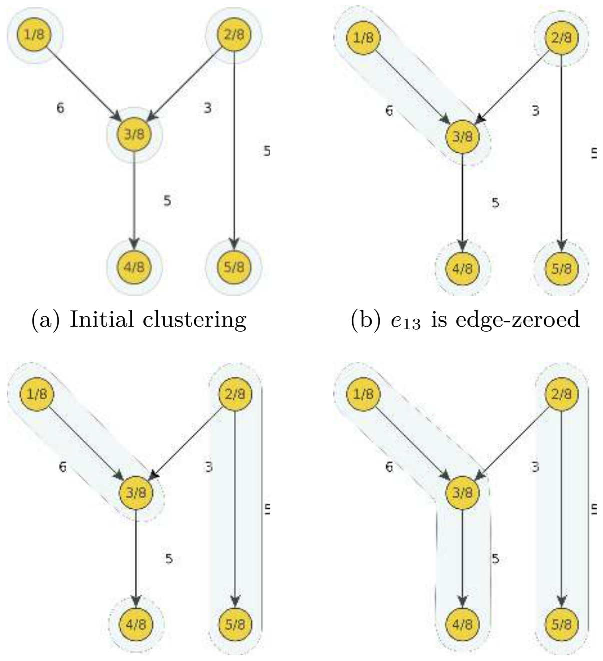
Exclusive processor allocation constraint. Let v_i and v_j be two nodes assigned to the same processor. Either v_i is executed before v_j or v_j is executed before v_i. They can not overlap.

Some clustering algorithms will be explained in the following. Parameters from subsection 3.2.2 will be used in order to describe them.

4.2.1 Edge-Zeroing or Single Edge, EZ or SE

Edge-zeroing was introduced by V. Sarkar in [57]. Initially every task is assigned to a different cluster. The edges of the DAG are given a priority according to some sort of heuristic, such as the cost of e_{ij} or “the b-level of e_{ij}”. The algorithm iterates over each edge and sets it to zero. If the resulting schedule length is smaller than or equal to the current schedule length, the clusters linked by the edge are merged together.

An example is shown in Figure 4.2. The CP consists of nodes 1, 3 and 4. The CP length is 35. The algorithm first tries to zero e_{13}. The resulting schedule length is 29. Since the schedule length is shorter, cluster 1 and cluster 3 are merged. Now e_{23} is zeroed. Nodes in a cluster are executed sequentially. Therefore a cluster containing 1, 2 and 3 takes 24 time units to be executed. The resulting schedule length is 37, greater than 29. Thus that configuration is discarded. e_{25} is then zeroed. The schedule length is still 29, therefore the new configuration is accepted. Finally e_{34} is zeroed.



(a) Initial clustering (b) e_{13} is edge-zeroed

(c) e_{25} is edge-zeroed (d) e_{34} is edge-zeroed

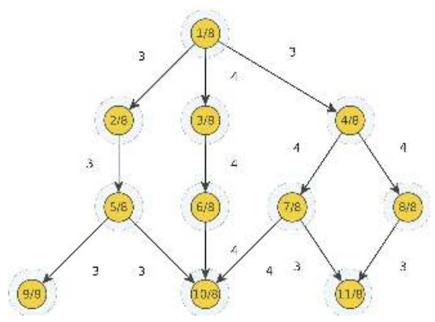
Figure 4.2: Edge-Zeroing

The schedule and CP length are 24 and 35 respectively. Thus the Scheduling Length Ratio (SLR) is 0.69. The algorithm started with an initial configuration of five clusters and finished with a configuration of two clusters. The initial configuration is slower because of the inter-task communication. A configuration with only one cluster is slower because there is no parallelism. This algorithm finds a solution which is a trade-off between inter-task communication and parallelism. The complexity is $O(E(V + E))$.

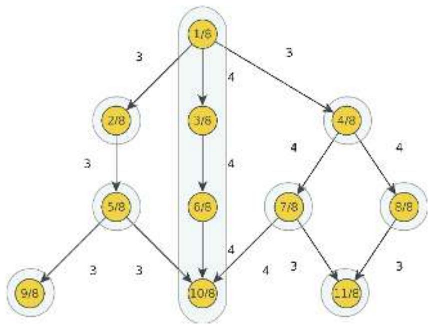
4.2.2 Linear Clustering, LC

Linear clustering was proposed in [58]. Firstly it is necessary to define what a linear cluster is. A cluster is linear if there exists a path that connect all the nodes belonging to the cluster. Linear clustering is an algorithm based on this concept. Basically, it decomposes the DAG into linear clusters. The algorithm starts by assigning one cluster to each node. Afterwards the CP is found and their nodes are merged into one cluster. These nodes and their communication edges are ignored hereafter. The CP of the remaining DAG is found and their nodes are merged into another cluster. The algorithm continues until all edges are explored.

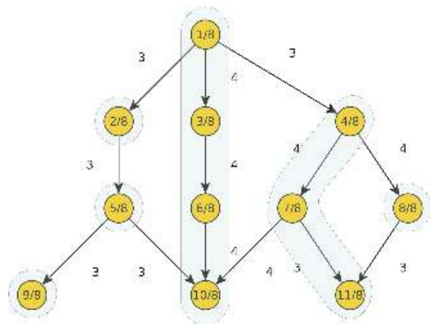
Figure 4.3 shows this algorithm. Initially the CP consists of nodes 1, 3, 6 and 10. They are merged into one cluster. From now on the nodes and edges of this cluster are ignored. The CP of the remaining graph is calculated. In this case there



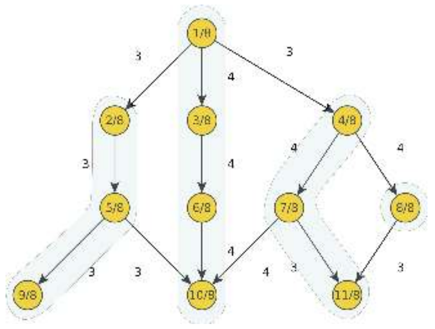
(a) Initial clustering



(b) Nodes 1, 3, 6 and 10 are clustered



(c) Nodes 4, 7 and 11 are clustered



(d) Nodes 2, 5 and 9 are clustered

Figure 4.3: Linear Clustering

are two solutions. The first solution is nodes 4, 7 and 11. The second solution is nodes 4, 8 and 11. The first solution is taken although the second one is also valid. Nodes 4, 7 and 11 are merged. Subsequently the algorithm merges nodes 2, 5 and 9 into another cluster. Node 8 is left alone.

The final solution consists of 4 linear clusters. In this case the length of the schedule is 36. The length of the CP is 44. Therefore the SLR is 0.82. The complexity is $O(V(V+E))$.

4.2.3 Dominant Sequence Clustering, DSC

Dominant sequence clustering was presented in [59]. The algorithm introduces the concept of scheduled DAG. The scheduled DAG is the original DAG plus a new set of edges with zero cost. The new edges link two independent nodes inside a cluster. They transform a non-linear cluster into a linear cluster. The schedule of a linear cluster is univocally defined. This concept serves to define the Dominant Sequence (DS). The DS is the CP of the scheduled DAG.

The algorithm incrementally explores the DAG with the help of two lists. One for the explored nodes and another one with the unexplored nodes. At every iteration a node is tried to merge with one or more predecessors, the node with highest

	MCP	**ETF**	**MD**	**DSC**
Task priority	b-level	t-level	relative mobility	t-level+b-level
DS task first	no	no	yes	yes
Processor selection	processor for EEST	processor for EEST	First processor satisfying moving interval condition	minimization procedure
Complexity	$O(V^2 log(V))$	$O(PV^2)$	$O(PV(V+E))$	$O((V+E)logV)$
Join/Fork	no	no	optimal	optimal
Coarse grain in-tree	optimal	optimal	no	optimal

Table 4.2: MCP, ETF, MD and DSC comparison

	$1-\frac{makespan(DSC)}{makespan(ETF)}$	$1-\frac{makespan(DSC)}{makespan(EZ)}$
Average group	3.30%	20.74%
Coarse grain group	0.06%	5.56%
Fine grain group	2.36%	19.39%
Average all	1.91%	15.23%

Table 4.3: ETF, EZ and DSC performance evaluation

priority among the unexplored free nodes and the unexplored partially free nodes. A free node is a node in which all its predecessors have been explored and a partial free node is a node in which at least one has been explored. The node is merged with the successors that minimize the t-level as long as it does not increase the t-level and satisfies the Dominant Sequence Length Reduction Warranty. Finally the priorities of its successors are updated. The algorithm iterates until the DAG is fully explored.

The algorithm provides optimal solutions for a subset of DAGs, namely fork, join, coarse grain trees and some fine grain trees. It has low complexity and high performance. The complexity is $O((V+E)logV)$.

T. Yang and A. Gerosoulis made a comparison of DSC with MCP, ETF and MD. It is shown in Table 4.2 and Table 4.3. They performed simulations with 180 graphs to evaluate the performance of DSC versus ETF and EZ. DSC and MD were not compared due to fact that they use similar strategies, thus the performance is expected to be similar. The graphs were divided in three subsets: average group (0.8-1.2 CCR), coarse grain group (3-10 CCR) and fine grain group (0.1-0.3). DSC performed better than ETF 56.67% of the times and better than EZ 93.89% of the times. It performed worse than ETF 20% of the times and worse than EZ 6.11% of the times.

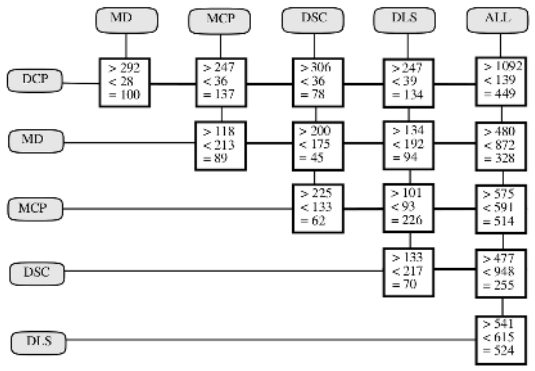


Figure 4.4: MD, MCP, DSC, DLS and DCP comparison

4.2.4 Mobility Directed, MD

Mobility directed was introduced in [53]. Scheduling is based on the concept of node mobility, $M(v_i) = ALAP(v_i) - ASAP(v_i)$. Relative mobility is defined as $M_r(v_i) = M(v_i)/w(v_i)$. This algorithm computes iteratively the relative mobilities of the nodes. It schedules into a processor the node with lowest mobility. When a node is scheduled before other node into the same processor a new edge with zero cost is added to the DAG. Then it is checked that there is no deadlock. The algorithm ends when all nodes are scheduled. The complexity is $O(V^3)$.

M. Wu and D. D. Gajski present a tool for parallel programming, called Hypertool. It is stated that up to 300% improvement in speed is achieved in problems such as Gaussian elimination, Laplace equations and Dynamic programming when Hypertool is used instead of manual scheduling. There is no such a improvement for Matrix multiplication and Bitonic sort.

4.2.5 Dynamic Critical Path, DCP

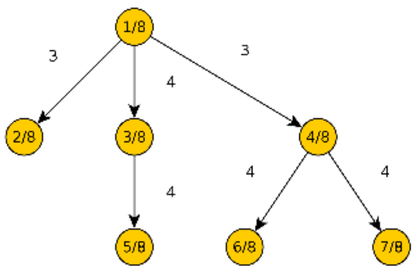
Dynamic critical path was proposed in [60]. The dynamic critical path is the critical path at a certain step in the clustering process. This concept is related to the dominant sequence. The nodes of the dynamic critical path have the same absolute earliest start time and absolute latest start time. The algorithm iterates over all the nodes. It clusters the node with highest priority at every iteration which is given by the minimum of the Absolute Latest Start Time (ALST) minus the Absolute Earliest Start Time (AEST) of v_i. If the value is zero it means it belongs to the dynamic critical path. The nodes are clustered to the cluster that gives best $AEST$ to the node and its critical child together. In each step $AESTs$ and $ALSTs$ are updated. The complexity of the algorithm is $O(PV(V+E))$.

An experiment with 420 DAGs and 5 different scheduling algorithms was performed (see Figure 4.4, taken from [60]). They tested MD, MCP, DSC, DLS and DCP. They ranked them according to the resulting makespan. The rank order was: DCP, MCP, DLS, MD and DSC.

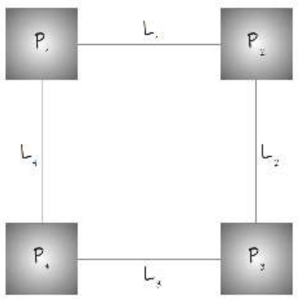
DSC required a larger amount of processors to be executed when compared to the others. Regarding the computation time of the scheduling, DLS was extremely slow. DSC and MCP were fast but the makespan was not so good. DCP and MD were in the middle. DCP is indicated to use with very large DAGs since it has low complexity compared to the short makespan produced.

4.3 Arbitrary processor network algorithms, APN

This kind of algorithms uses a more complex model of the system architecture in order to achieve a better performance. Therefore they have an increased complexity. They require two inputs: the static definition of the application (DAG) and the network topology (TG) of the target architecture. They include routing of messages into the scheduling. An example is shown in Figure 4.5.



(a) Directed acyclic graph



(b) Topology graph

Figure 4.5: Arbitrary processor network algorithm

Classical models assume that all communication can happen at the same time and that all processors are fully connected. These simplified models might not be enough in systems where communication is a scarce resource. Contention-aware scheduling algorithms take care of the communication resources scheduling.

Contention-aware scheduling make use of a TG= (P, L), where P represents the set of nodes/processors and L the set of edges/communication links. This approach affects the calculation of the finish time of a communication edge e_{ij} in the DAG. Lets assume $R =< L_1, L_2, ..., L_l >$ being the route from P_{src} to P_{dst} where $L_i \in L$ for $i = 1...l$ is the communication links of the route.

In the classical model, the finishing time is calculated as follows.

$$t_f(e_{ij}, P_{src}, P_{dst}) = t_f(v_i, P_{src}) + \begin{cases} 0 & if P_{src} = P_{dst} \\ c(e_{ij}) & otherwise \end{cases}$$

In the contention model, the communication edges of the DAG have to be scheduled in the TG.

$$t_f(e_{ij}, P_{src}, P_{dst}) = \begin{cases} t_f(v_i, P_{src}) & if P_{src} = P_{dst} \\ t_f(e_{ij}, L_l) & otherwise \end{cases}$$

Some arbitrary processor network algorithms will be explained in the following. Parameters from subsection 3.2.2 will be used in order to describe them.

4.3.1 Mapping Heuristic, MH

The mapping heuristic algorithm was presented in [61]. It uses a list-based scheduling algorithm that has been modified in order to produce a schedule which is suitable for a target machine with an arbitrary network topology. It has in mind the effects of different communication delays depending on the selected route and contention problems. Both scheduling and routing are done at every step. A new parameter called average degree was introduced, AD=number of edges/number of nodes. Both AD and CCR show the weight of the communication in the application.

An ideal model of the network is not good enough in communication intensive applications. MH introduces a more advanced model of the communication costs:

$$c(e_{ij}, p_1, p_2) = \left(\frac{Data(e_{ij})}{R} + I\right) \cdot H(p_1, p_2) + D(p_1, p_2) \tag{4.1}$$

where the following terms stand for

$\boldsymbol{Data(e_{ij})}$	Amount of sent data
$\boldsymbol{R}$	Speed of the link
$\boldsymbol{I}$	Time to initiate the transmission
$\boldsymbol{H(p_1, p_2)}$	Number of hops between p_1 and p_2
$\boldsymbol{D(p_1, p_2)}$	Contention time. Time which the transmission is delayed due to previously routed transmissions.

In the MH algorithm there is a routing table in every processor. The table keeps track of the communication links usage. It serves to avoid contention problems. The table has an entry for each processor of the system. An entry has three data cells. The first one contains the number of hopes to reach the processor. The second one has the preferred outgoing link to reach the processor. The third one is the delay due to contention. These tables are used to route the communication edges. The complexity of the algorithm is $O(P^3V^2 + PV^2 + P^3)$, and $O(V^2)$ for a constant number of processors.

H. El-Rewini and T. G. Lewis performed simulations with a tool called Task Grapher with different network topologies such as ring, star, mesh, hypercube and fully connected networks. They proved that in hypercubes the effects of communication should be taken into account due to the effect of contention in the communication channels. Traditional list-based scheduling algorithms with ideal model networks are incomplete and it affects the performance. This is especially noticeable in applications with high AD and CCR.

4.3.2 Bottom-Up, BU

The bottom-up algorithm was introduced in [62]. The algorithm first maps the nodes onto the processors. Finally it routes the inter-processor communication into the network.

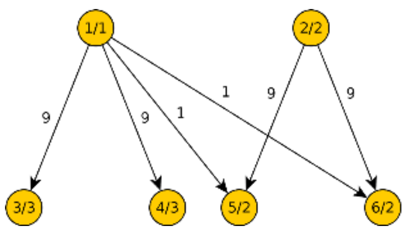
The algorithm starts mapping the deepest nodes and keep ascending the tree level by level until the entry nodes. Three mapping heuristics are suggested. The first one tries to map a task in the same processor when the communication cost is higher than the computation cost. The other two try to balance the load onto the different processors. The algorithm keeps track of the loads on every processor.

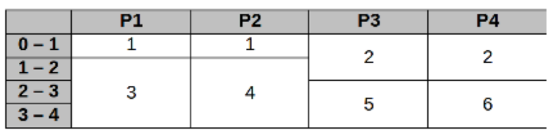
In the second phase communication links are assigned for the inter-processor communications. This is done starting from the entry nodes. An adaptive channel assignment heuristic is required. It takes into account the network topology and the link loads in order to avoid contention and deadlocks. Some routing algorithms have been proposed: hypercube topology [63], mesh topology [64] and torus topology.

N. Mehdiratta and K. Ghose performed some experiments with randomly generated task graphs. They compared BU with MH and a top-down TD scheduler (same algorithm as BU but starting from the entry node). MH and TD had lower performance than BU for high fan-out graphs. MH had lower performance than TD and BU for high fan-in graphs. TD performed a little bit better than BU in this case. BU also showed great scalability with the number of nodes per DAG. BU outperformed MH in a 4x4 torus topology and a 16 hypercube topology and the difference increased with the number of nodes.

4.4 Duplication algorithms

This family of algorithms duplicates a task in several nodes for different purposes. Task duplication is beneficial in DAGs that contains nodes with more than one child node (see Figure 4.6).





	P1	P2	P3	P4
0 – 1	1	1	2	2
1 – 2	3	4		
2 – 3			5	6
3 – 4				

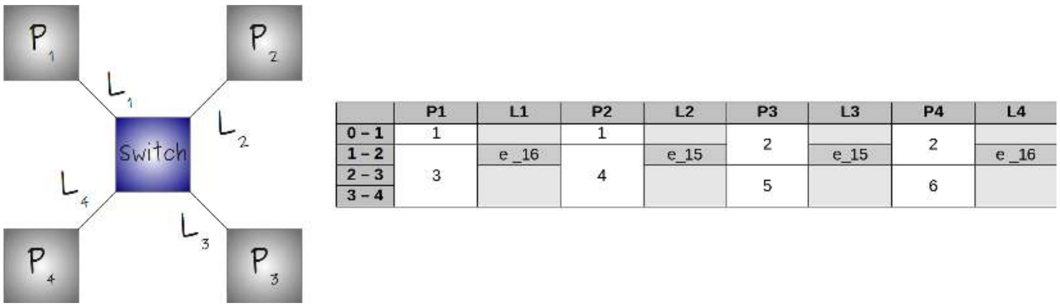
Figure 4.6: Task duplication

Some duplication algorithms will be explained in the following.

4.4.1 Contention-aware scheduling algorithm with task duplication

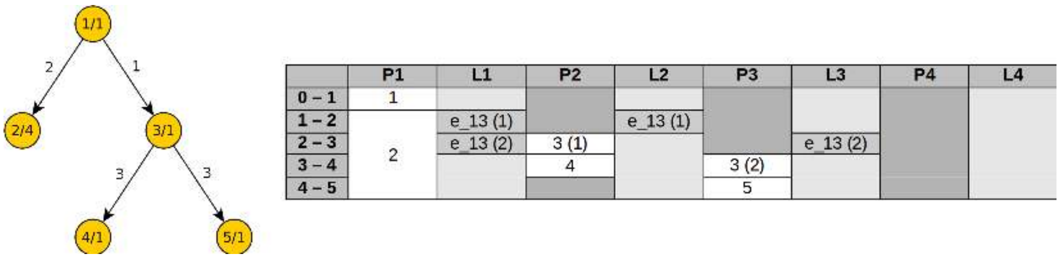
This algorithm belongs to both the arbitrary processor network family and the duplication family. The purpose of this algorithm is to avoid inter-node communication as much as possible (contention-aware scheduling). This can be done by means of task duplication (duplication scheduling). Task duplication increases its benefits in a contention-aware context.

There are two different approaches: origin duplication (see Figure 4.7) and destination duplication (see Figure 4.8). Both can be combined to improve the performance.



	P1	L1	P2	L2	P3	L3	P4	L4
0 – 1	1		1		2		2	
1 – 2	3	e_16	4	e_15		e_15		e_16
2 – 3					5		6	
3 – 4								

Figure 4.7: Origin duplication



	P1	L1	P2	L2	P3	L3	P4	L4
0 – 1	1							
1 – 2	2	e_13 (1)		e_13 (1)				
2 – 3		e_13 (2)	3 (1)			e_13 (2)		
3 – 4			4		3 (2)			
4 – 5					5			

Figure 4.8: Destination duplication

In [65], the implementation of this algorithm is explained. The algorithm first finds the critical parent of a task, the critical parent of the critical parent and so on. The critical parent of a node is the predecessor with highest finish time of the communication edge. The next step is to recursively duplicate the critical parent to minimize the finish time, which is done by tentative scheduling and redundant task/edge removal.

The complexity is $O(|P|^2|V|^2|E|^2O(routing))$. Where $O(routing)$ is the complexity of the routing algorithm.

Contention-aware scheduling with task duplication's benefits are more noticeable in applications with high amount of communications and networks that are far from the ideal case.

4.5 Summary

In this chapter many algorithms have been presented along with some comparisons between them. Some algorithms get better results than the others but are computationally more complex. Clustering algorithms get reasonably good results at an affordable complexity.

Arbitrary network processor algorithms and task duplication algorithms should be considered for high performance applications where communication is a bottleneck.

Chapter 5

Discussion and conclusions for scheduling

This chapter makes use of the knowledge gained in order to provide ideas on how to design and implement a scheduler that fulfills the requirements that were set up in Chapter 1.

5.1 Requirements analysis

An analysis of the requirements list given in Section 1.3 is performed below:

REQ_1 The scheduler shall cope with both serial and parallel applications.
The algorithms presented in Chapter 4 are by design able to cope with parallel applications. A serial application can be seen as a special case of a parallel application. A serial application is a parallel application with just one task and no edges. Therefore the algorithms presented in Chapter 4 can be easily adapted to cope with serial applications as well.

REQ_2 The scheduler shall do automatic parallelization of the application. Parallelization shall be transparent to the programmer.
Automatic parallelization requires two complex activities. The first one is the decomposition of the application into small tasks with the right granularity. Granularity is a key factor that seriously affects the trade-off between parallelization and communication. The second activity is to handle the dependencies. A tool for automatic parallelization has to insert commands in the code for communication and synchronization of tasks. There is not in the market a tool that completely abstracts the programmer from this activities. OpenMP is a very popular tool that was presented in Chapter 3. In OpenMP the programmer specifies the sections of code that can be parallelized by using compiler directives, which is a clean way

since it does not modify the code. The approach of CAP's compilers is to make a two step decomposition. The first step identifies the hotspots and tries to isolate them in independent pieces of code, called codelets. The second step is to profile the application and use machine learning techniques in order to improve the decomposition. This idea sounds very interesting and has an important benefit. It can adapt to different target platforms thanks to the machine learning techniques.

REQ_3 The scheduler shall parallelize tasks as much as possible.
REQ_4 The scheduler shall avoid communication as much as possible.
REQ_6 The scheduler shall generate a schedule with the minimum execution time.
Scheduling is a trade-off between parallelization and communication. The optimal schedule finds the right balance. It takes a lot of time to compute the optimal schedule, thus heuristics are used in order to find a solution with a performance close to the one of the optimal solution. The algorithms presented in Chapter 4 are not optimal, they use heuristics. One of the main advantages of the algorithms of Chapter 4 is that if two tasks are mapped to the same processing unit the communication cost is zero. This is true if the output of the predecessor is stored in the local memory of the processing unit. In that case the successor retrieves the output from the local memory. It reduces communication and increases performance. In the implementation the output of a predecessor shall be stored in local memory.

REQ_5 The scheduler shall provide to an application an end-to-end guarantee.
The DAG is a tool to model parallel applications. This tool can be used to calculate end-to-end latencies in multi-core systems. The makespan of the application is what gives the end-to-end latency. In real-time systems the end-to-end latency shall be smaller than a deadline.

REQ_7 The scheduler shall be energy efficient.
There are two ways of increasing energy efficiency. The first method is by increasing parallelization. Two small energy efficient cores can perform the same task as a powerful processor with less energy. The second method is by using the power saving mechanisms provided by the target platform. The prototype board provides an idle mode to sleep cores if they are not used.

REQ_8 The scheduler shall be target independent. The algorithm shall be easily portable to other platforms.
List-based algorithms, clustering algorithms and duplication-based algorithms do not have dependencies with the platform. It means that the algorithms are easily portable from a target system to a different target system. However, arbitrary processor network algorithms are target dependent. These algorithms need an extra input, the TG. Extra performance is gained in exchange of portability.

5.2 Performance comparison

There are two basic parameters to measure the performance of the algorithms: the complexity and the makespan. The complexity tells how much time is needed to construct the schedule. The makespan is the length of the schedule produced.

List-based algorithms are based on very simple ideas or heuristics. However, the makespan and the time to construct the schedule is not very good. Clustering algorithms are normally based on list-based algorithms. Their complexity is usually lower than the one of list-based algorithms and the produced makespan is still good. This is due to the fact that they assume an unlimited number of processors to schedule the tasks. LC is an algorithm that gives optimal solutions for coarse grain in-tree and join/fork DAGs at a low complexity (see table 5.1).

Arbitrary processor network algorithms are recommended to be used in application with high CCR and where the communication network is far from the ideal case. The complexity of the implementation is greater than in the other algorithm families, specially with complex network topologies. Only the previous conditions justify their usage. BSA is the algorithm with better performance among the algorithms studied according to [66] (see table 5.2).

Duplication-based algorithms are more complex in order to reduce the makespan. They should be used only in cases where a very short makespan is required.

Novel algorithms combined the ideas from arbitrary processor network and duplication-based algorithms [65]. It turns out that there is a synergy between the ideas and quite good results are obtained.

Source		**List-based**				**Clustering**				
		HLFET	**MCP**	**ETF**	**DLS**	**EZ**	**LC**	**DSC**	**MD**	**DCP**
[1]	**Makespan**	Longest	Medium-long	Medium-short	Shortest	N/A	N/A	N/A	N/A	N/A
	Complexity	$O(PV^2)$	$O(PV^2)$	$O(PV^3)$	$O(PV^3)$	N/A	N/A	N/A	N/A	N/A
[59]	**Makespan**	N/A	Optimal in-tree	Optimal in-tree	N/A	N/A	Optimal in-tree/ join-fork	Optimal join-fork	N/A	N/A
	Complexity	N/A	$O(V^2 \log V)$	$O(PV^2)$	N/A	N/A	$O((V+E)\log V)$	$O(PV(V+E))$	N/A	N/A

Table 5.1: List-based algorithms and clustering algorithms: Performance comparison

Source	**Arbitrary processor network**	**Reduction of makespan by using BSA**
[65]	MH	40%
	DLS	50%
	BU	71%

Table 5.2: Arbitrary processor network: Performance comparison

5.3 Design decisions

The aim of this thesis is to implement a code generation toolchain for parallel applications. In this section the design decisions to fulfill the requirements will be listed. The implementation process will be iterative.

The aim is to use a simple algorithm first in order to get to know the hardware. A simple algorithm is also simpler to debug. Once the first prototype is working new features will be implemented.

The first algorithm to be implemented is linear clustering. In table 5.3 the main features of the toolchain are listed and mapped with the requirements they meet. **REQ_2** is only partially fulfilled because the dependencies are handled but the decomposition has to be done manually by the programmer.

Item	**Description**	**Fulfilled requirements**
Implementation of LC on the prototype board	LC provides optimum schedules for optimal coarse grain in-tree and join/fork DAGs. The algorithm is also independent from the target platform.	REQ_3, REQ_4, REQ_6 and REQ_8
Adaptation for serial applications		REQ_1
Utilization of the target energy modes	Energy modes will sleep the cores when not being used.	REQ_7
Implementation of a DAG generator	The output DAG from the generator will be the input to the scheduler. The DAG is drawn by the programmer in a GUI.	REQ_2 (partially)
Offline calculation of the makespan	The calculation will serve as an end-to-end latency guarantee.	REQ_5

Table 5.3: First prototype specification

A new prototype will be implemented once the first prototype works correctly. There is room for improvement changing the scheduling algorithm. A duplication based algorithm could reduce the makespan. The algorithm would duplicate tasks with more than one successor.

The ideal system would be an arbitrary processor network algorithm with task duplication (see table 5.4). The Epiphany network-on-chip would be taken into account in order to avoid communication contention due to the side-effects of inter-task communication. Something important to notice is that this prototype would not meet **REQ_8**. This is accepted because the previous prototype met REQ_8. The programmer would implement the application as a serial application and the toolchain would identify the optimal decomposition of the application into tasks and would handle their scheduling and dependencies. No graph would be needed to draw. One possibility to be considered is that the final system would run several algorithms and select the one with the best performance.

Item	**Description**	**Fulfilled requirements**
Customized scheduler for the prototype board	Arbitrary processor network + task duplication algorithm	REQ_3, REQ_4 and REQ_6
Adaptation for serial applications		REQ_1
Utilization of the target energy modes	Energy modes will sleep the cores when not being used.	REQ_7
Implementation of a DAG generator	The output DAG from the generator will be the input to the scheduler. The DAG is generated from the serial version of the application.	REQ_2
Offline calculation of the makespan	The calculation will serve as an end-to-end latency guarantee.	REQ_5

Table 5.4: Ideal system specification

Part II

Practical phase

Chapter 6

System design and architecture: Toolchain

The system to be implemented has been thought as a toolchain. The main use case of the system consists in giving a graph representing a parallel application as the input and getting some files with the skeleton of the application as the output. A node is a part of code that cannot run in parallel but sequentially on a specific core. Figure 6.1 shows the toolchain modules and how they are connected together. It also shows the outputs of each module, which are the inputs to the next module.

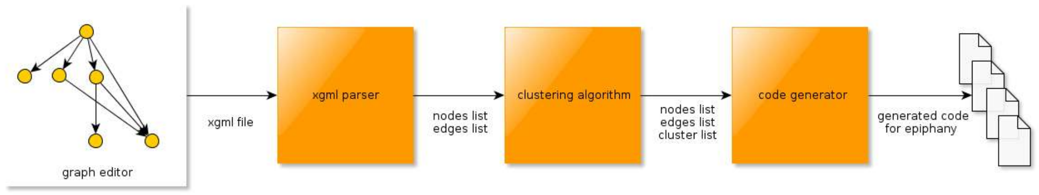


Figure 6.1: Toolchain

6.1 Graph editor

The graph editor used is yEd. It is a freeware software that runs on Windows, Linux and Mac OS. The program is able to load and save diagrams from/to XML-based format files.

Figure 6.2 shows an example of a graph drawn in yEd. It has 6 nodes and 7 edges. Each node contains the node id and the node weight separated by the slash character. Each edge has the communication cost.

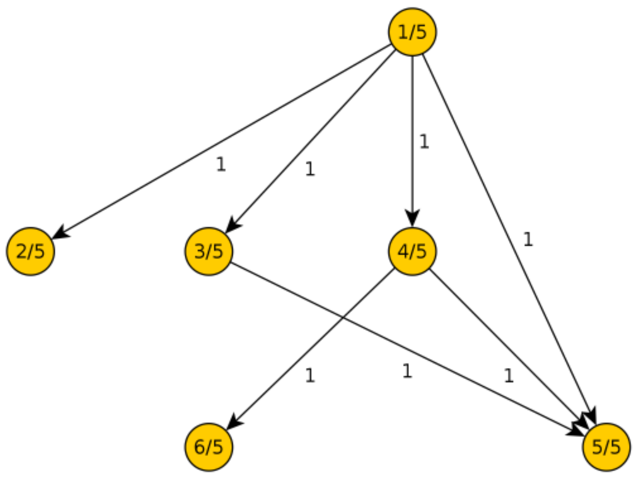


Figure 6.2: yEd graph

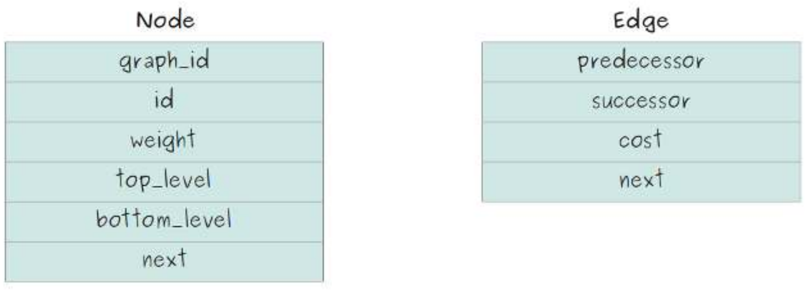


Figure 6.3: Node and edge structures

6.2 Xgml parser module

The xgml parser initializes the DAG. It reads the xgml file generated by the graph editor and generates two linked lists: a node list and an edge list (see Figure 6.3).

The module uses libxml2, which is a software library written in C for parsing XML documents. It was developed for the GNOME project and is available under the MIT License.

A node is a structure with a *graph id*, an *id*, a *weight*, a *top level*, a *bottom level* and a pointer to the *next* node in the linked list. The *graph id* is a field defined by the graph editor and that is required to identify the edges that link nodes. The *id* is a number given by the user. The *weight* is the execution time of the task. The *top level* and the *bottom level* are two fields used by the linear clustering algorithm.

An edge is a structure with a *predecessor*, a *successor*, a *cost* and a pointer to the *next* edge in the linked list. The *predecessor* is the source node where the edge starts from. The *successor* is the sink node where the edge is pointing at. The *cost* is the communication time.

6.3 Clustering algorithm module

The clustering algorithm module implements linear clustering, proposed by [58]. Many algorithms are described in [46], among them linear clustering. Its description is presented below.

Algorithm 1 Linear clustering

```
$\mathbf{E_{unex}}$: set of unexamined edges
$\mathbf{V_{unex}}$: set of nodes $v \in \mathbf{V}$ on which edges $\mathbf{E_{unex}}$ are incident
Create initial clustering $C_0$: allocate each node $v \in \mathbf{V}$ to a distinct cluster $C \in \mathbf{C}$
$|\mathbf{C}| = |\mathbf{V}|$
$\mathbf{E_{unex}} \leftarrow \mathbf{E}$
while $\mathbf{E_{unex}} \neq \emptyset$ do
  Find a critical path $cp$ of graph $G_{unex} = (\mathbf{V_{unex}}, \mathbf{E_{unex}})$
  Merge all nodes $\mathbf{V_{cp}}$ of $cp$ into one cluster
  $\mathbf{E_{unex}} \leftarrow \mathbf{E_{unex}} - \{e_{ij} \in \mathbf{E_{unex}} : v_i \in \mathbf{V_{cp}} \vee v_j \in \mathbf{V_{cp}}\}$
end while
```

In order to find the critical path of the DAG, the top level and the bottom level of each node are calculated. To do that it is necessary to sort the nodes topologically. The implementation uses the algorithm proposed by [67]. The complexity of the algorithm is $O(|V| + |E|)$.

Algorithm 2 Topological sorting

```
$\mathbf{L} \leftarrow$ Empty list that will contain the sorted elements
$\mathbf{S} \leftarrow$ Set of all nodes with no incoming edges
while $\mathbf{S} \neq \emptyset$ do
  Remove a node $v$ from $\mathbf{S}$
  Insert $v$ into $\mathbf{L}$
  for all node $m$ with an edge $e$ from $v$ to $m$ do
    Remove edge $e$ from the graph
    if $m$ has no other incoming edges then
      Insert $m$ into $\mathbf{S}$
    end if
  end for
  if graph has edges then
    Return error (graph has at least one cycle)
  else
    Return $L$ (a topologically sorted order)
  end if
end while
```

The output of this module is a linked list per cluster and a list with the head nodes of the clusters. Each cluster contains a list of tasks that will be mapped to a

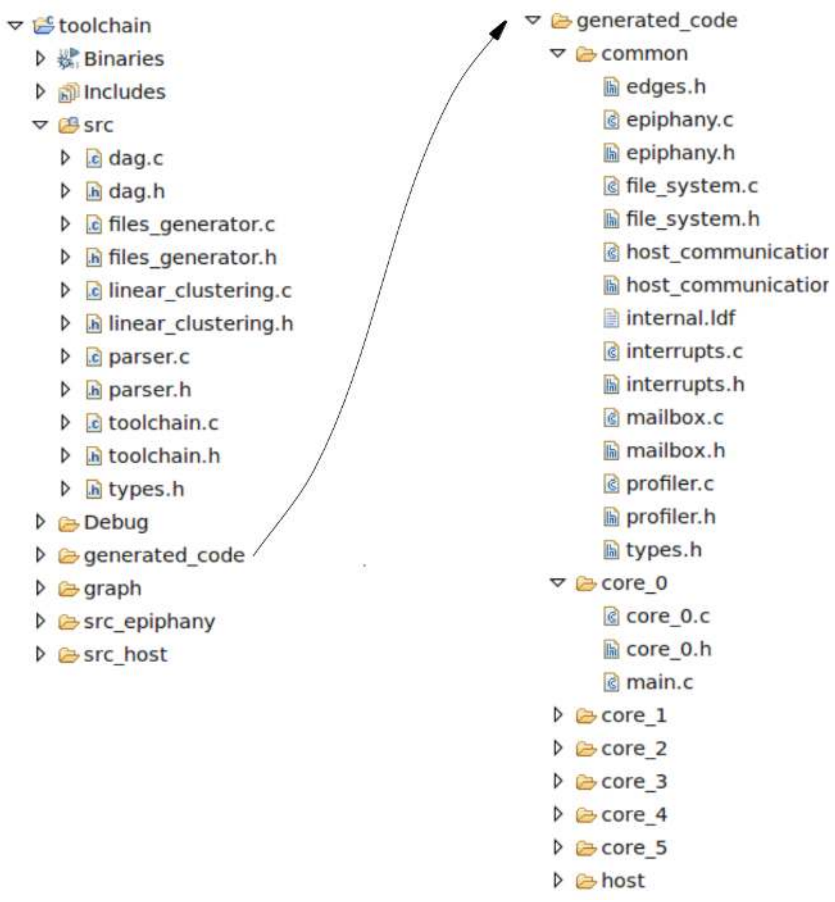


Figure 6.4: Generated code

specific core.

6.4 Code generator module

The code generator module is in charged of generating the skeleton code of the application. Once it is generated, it has to be loaded onto the Epiphany projects.

Figure 6.4 shows the generated code for the brake-by-wire application. It is created in a folder called *generated_code*. Inside *generated_code* there are three kinds of folders: *common*, *core_i* and *host*. The application shown in the figure requires 6 cores for the execution.

The code in folder *common* goes to a common project for all the cores. It contains the software modules that allow us to run the parallel application on Epiphany. The *core_i* folder contains specific code for a core, mainly the tasks that the core has to execute. The *host* folder is the generated code that the host computer has to run to start Epiphany and fetch the results once Epiphany has finished.

6.5 Application development

After the skeleton code has been generated, the user has to define the following sections.

- The *inputs* of the parallel application.
- The *outputs* of the parallel application.
- The *calculations section* of the tasks.
- The *communication edges* between tasks.

The *inputs* and the *outputs* are empty structures. Inside the structures the programmer creates the fields for the specific application.

```
typedef struct {

} input_i_t;

typedef struct {

} output_j_t;
```

Below it is presented an example of the code generated for a task. The sections inside a task are always the same: *input*, *output*, *retrieve the input*, *start task profiler*, *calculations*, *end task profiler* and *transmit results*. All sections are generated by the code generator module except the *calculations section* that is specific of the application.

```
void task_j(void)
{
  // Input
  edge_ij_t edge_ij;

  // Output
  edge_jk_t edge_jk;

  // Retrieve the input
  while(0 != retrieve_message(i, j, &edge_ij))
    __asm__("IDLE;");

  start_task_profiler();

  // Do calculations and store the results
  // input = f(output);
  //
  // User code

  Log_tasks_core[j] = end_task_profiler();

  // Transmit the results
  send_message(sizeof(edge_jk_t), j, k, &edge_jk,
      CORE_NUMBER_k);
  return;
}
```

The last step is to define the *communication edges*, which are also empty structures where the user creates customized fields.

```
typedef struct {

} edge_ij_t, *pointer_edge_ij_t;
```

Chapter 7

System design and architecture: Epiphany application

In this chapter it will be described the software modules that allow us to run a parallel application on Epiphany. The system is made up of six software modules: *epiphany module*, *file system module*, *interrupts module*, *mailbox module*, *host communication module* and *profiler module*. Figure 7.1 shows the stack diagram of the system.

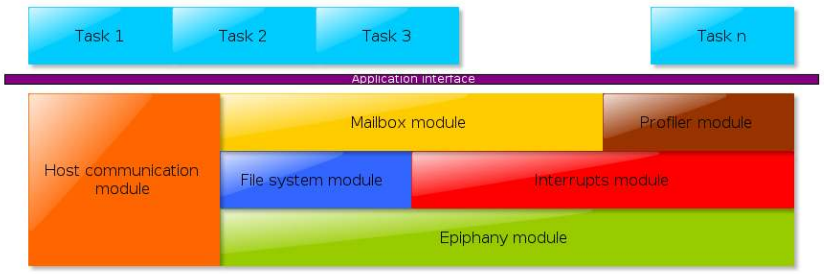


Figure 7.1: Software modules stack

Some software modules requires the definition of a set of data structures in the local memory. The location of these data structures is specified in a linker definition file. That file also tells the compiler to place the stack in the local memory instead of the external DRAM and to place the Interrupt Vector Table (IVT) at the beginning of the local memory. Figure 7.2 shows how the local memory of a core and the shared memory of the board are used.

7.1 Epiphany module

This module defines a global variable called *Me*. *Me* contains relevant information about the core that is used by other modules, see Table 7.1.

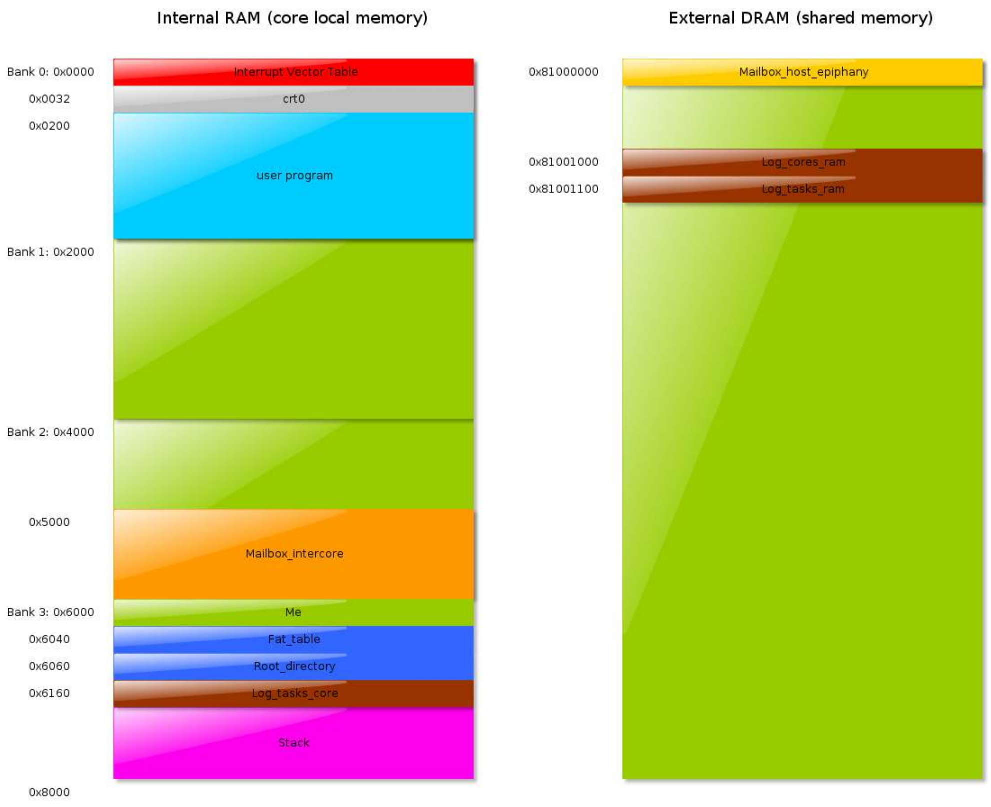


Figure 7.2: Board memory

7.2 Interrupts module

The interrupts module is where the ISRs of the system are declared and defined. An ISR is a callback function which is triggered by an interrupt. An interrupt is a signal to the processor indicating the arrival of a new event. There are two kinds of interrupts depending on the nature of the event: hardware interrupts and software interrupts.

Hardware interrupt: is generated by either an internal peripheral or an external peripheral to the processor. The hardware interrupts in epiphany are: *synchronization/reset*, *timer 0 expired*, *timer 1 expired*, *Direct Memory Access (DMA) channel 0 end of transfer* and *DMA channel 1 end of transfer*.

Software interrupt: is generated by an exceptional condition or a special instruction. The software interrupts in epiphany are: *software exception*, *memory protection fault* and *software interrupt*.

Each core has an interrupt controller that is in charged of executing the appropriate ISR depending on the interrupt generated. Figure 7.3, taken from [68], shows the operation workflow. ILAT is a system register that records all interrupt events. IMASK is a system register for blocking interrupts. IPEND is a system register

core__id	A unique id for calling some library functions.
core__number	A unique number to identify the core. The value range is from 0 to 15.
row	The row which the core is located in. The value range is from 0 to 3.
col	The column which the core is located in. The value range is from 0 to 3.
pending__data__ transfer__array	Synchronization array for the mailbox module. A core that wants to send or retrieve a message puts its element to 0xFF.
core__with__token	Synchronization variable for the mailbox module. It stores the core number of the core with grant access to the mailbox.
pointer__core__mailbox	Pointer to the mailbox.
task__sec__counter	It counts the number of seconds elapsed during the execution of a task.
core__sec__counter	It counts the number of seconds elapsed during the execution in a core.
target__core__id	Auxiliary variable for the mailbox module.
target__pending__ data__transfer__array	Auxiliary variable for the mailbox module.

Table 7.1: Me structure's fields

that keeps track of the ISRs currently being processed. IVT is the interrupt vector table. It contains a set of branch instructions that point to the appropriate ISRs. IRET stores the program counter of the next instruction to be executed when an interrupt starts being serviced. PC is the program counter and has the address of the instruction to be executed. GIE and GID are global interrupts enable and disable.

The interrupts module makes use of the *timer 0 expired*, the *timer 1 expired*, the *DMA channel 0 end of transfer* and the *software interrupts*.

The *timer 0* and *timer 1 expired* interrupts are used to increment a counter to count the number of seconds elapsed during a task execution or a core execution. These counters are used by the profiler module.

The *DMA channel 0 end of transfer* interrupt is used to synchronize with other cores and unlock the mailbox, which is a shared resource.

The *software interrupt* is used to synchronize the cores when accessing the mailbox and give the token to only one of them at a time.

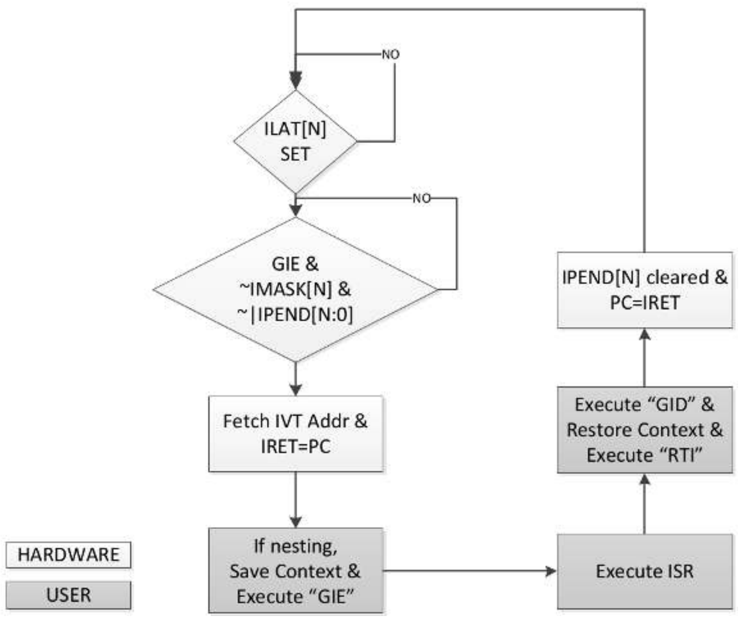

NO
ILAT[N] SET
NO
GIE & ~IMASK[N] & ~|IPEND[N:0]
Fetch IVT Addr & IRET=PC
If nesting, Save Context & Execute "GIE"
Execute ISR
Execute "GID" & Restore Context & Execute "RTI"
IPEND[N] cleared & PC=IRET
HARDWARE
USER

Figure 7.3: ISR operation

7.3 File system module

The file system module handles the management of the second half of the second memory bank. The size of this chunk of memory is 4kB and it is divided in 32 blocks of 128B size. The file system module is managed by two arrays: the *Fat_table* and the *Root_directory* (see Figure 7.4).

Fat_table: is a 32 length array. An entry in the array is an unsigned char. Each entry corresponds to a block in the memory. The unsigned char can either be 0x00 or 0xFF. 0xFF means that the block has been allocated. 0x00 means that the block is free.

Root_directory: is a 32 length array. An entry in the array is a structure that contains the following information: *entry_used*, *sender*, *recipient*, *start_block* and *message_size*. Each entry corresponds to a message sent to the core. *Entry_used* specifies whether the entry is in use. A message is stored in memory continuously from the block *start_block* and the required number of blocks is calculated with the *message_size*.

The file system module provides two functions to control the *Fat_table*: *allocate_memory_blocks* and *free_memory_blocks*. The first method writes 0xFF in the first set of contiguous blocks which the message fits in. The second method writes 0x00 in the blocks that were previously used by a message.

The file system module also provides two functions to control the *Root_directory*: *create_entry* and *delete_entry*. The first method creates a new entry in the first entry that is not in use. The second method deletes an entry in the *Root_directory*

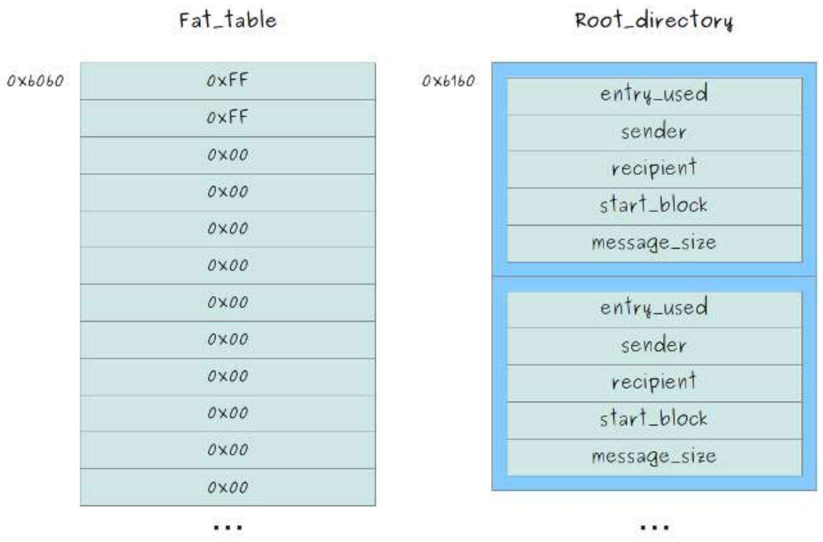


Figure 7.4: Fat_table and the Root_directory

by putting the field *entry_used* into not in use.

7.4 Mailbox module

The mailbox module provides a system service to send and receive inter-core messages. This communication service has a high level of abstraction. Beneath it uses the interrupts module and the file system module. It implements two functions: *send_message* and *retrieve_message*.

The mailbox system stores the messages in the memory space managed by the file system module. It is a 4kB memory area located at address 0x5000. All the cores have access to the mailbox of a specific core. This fact makes it a shared resource. Therefore special precautions have to be taken. A synchronization mechanism needs to be implemented. The synchronization mechanism designed is made up of two variables in each core: *pending_data_transfer_array* and *core_with_token*.

pending_data_transfer_array: is an sixteen-size array. The first element of the array corresponds to the first core, the second element to the second core and so on. When a core wants to send or retrieve a message from another core, it has to write 0xFF into its element. Once the operation has been completed the core writes 0x00 back into the element.

core_with_token: contains the number of the core with grant access to send or retrieve messages. The value of this variable is re-evaluated every time there is a software interrupt.

The function *send_message* is used to transfer a message into a specific core. Figure 7.5 shows a description of the *send_message* function.

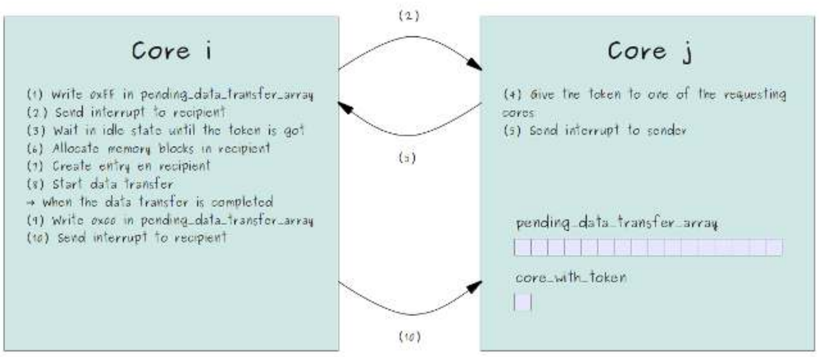

Core i
(1) Write 0xFF in pending_data_transfer_array
(2) Send interrupt to recipient
(3) Wait in idle state until the token is got
(6) Allocate memory blocks in recipient
(7) Create entry en recipient
(8) Start data transfer
→ when the data transfer is completed
(9) Write 0x00 in pending_data_transfer_array
(10) Send interrupt to recipient
(2)
(5)
(10)
Core j
(4) Give the token to one of the requesting cores
(5) Send interrupt to sender
pending_data_transfer_array
core_with_token

Figure 7.5: *Send_message* function

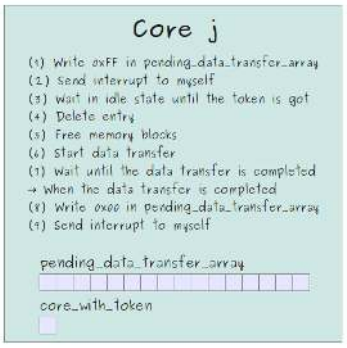

Core j
(1) Write 0xFF in pending_data_transfer_array
(2) Send interrupt to myself
(3) Wait in idle state until the token is got
(4) Delete entry
(5) Free memory blocks
(6) Start data transfer
(7) Wait until the data transfer is completed
→ When the data transfer is completed
(8) Write 0x00 in pending_data_transfer_array
(9) Send interrupt to myself
pending_data_transfer_array
core_with_token

Figure 7.6: *Retrieve_message* function

The function *retrieve_message* is used to retrieve a message from the mailbox. Figure 7.6 shows a description of the *retrieve_message* function.

7.5 Host communication module

The host communication module is the place where the programmer defines the inputs and the outputs of the parallel application.

The module defines the variable *Mailbox_host_epiphany* in external DRAM (address 0x81000000), which contains the *inputs*, the *outputs*, the *input flag* and the *output flags*.

The flags are variables to synchronize the cores with the host. The cores do not start the execution until the *input flag* is set to 1. The host do not fetch the results until all the *output flags* are set to 1.

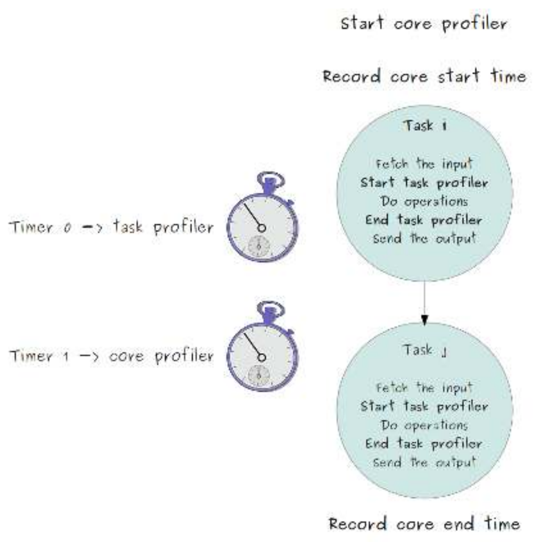


Figure 7.7: Profiler functions

7.6 Profiler module

The profiler module measures the execution time of the cores and the execution time of the tasks (see Figure 7.7). The measurements are stored in a log.

A core has two available timers: timer 0 and timer 1. A timer has a 32-bit register to count and the frequency of the clock is 600Mhz. Both timers are configured to generate an interrupt every second. The interrupts increment a counter.

Timer 1 is used to profile a core. *Start_core_profiler* is a function that resets the counter and starts the timer. It is called just after the *input flag* is set to 1. After the core has executed all task assigned to it, the timer value is recorded in the log with the function *record_core_time*. The value of the timer is given in clock ticks. Therefore it has to be scaled to nanosecond when storing it into the log.

Timer 0 is used to profile the tasks in a core. *Start_task_profiler* function resets the counter and starts the timer. *Record_task_time* records its value.

Two arrays are allocated in external DRAM. *Log_cores_ram* (address 0x81001000) stores the cores' execution time. 8 bytes are needed for each of the 16 cores. *Log_tasks_ram* (address 0x81001000) stores the tasks' execution times. Writing into the external RAM adds extra traffic into the e-mesh, therefore the tasks' execution times are first stored into *Log_tasks_core* (address 0x6160) located in internal RAM and after moved to *Log_cores_ram*.

Chapter 8

Implementation

This chapter provides the decisions taken during the implementation phase that are not covered in Chapters 6 or 7.

8.1 Power management

The Epiphany chip provides the *IDLE* instruction to put a core into a low-power or standby state. After a reset a core is in idle. An external interrupt event activates the core. The core keeps active until an *IDLE* instruction. The core returns to the normal execution after another interrupt event.

Every core has a 32-bit STATUS register that contains information regarding the execution status. Bit 0 is a flag called ACTIVE. 0 indicates that the core is idle and 1 indicates that the core is active. When the core is in the standby state, the clocks are disabled and the power consumption is minimized.

This mode is used when retrieving messages from the mailbox. If the message has not arrived by the time it has been tried to be retrieved, the core is deactivated. The actual implementation is shown below.

```
// Retrieve the input
while(0 != retrieve_message(i, j, &edge_ij))
  __asm__("IDLE;");
```

The core is woken up when another core sends a message to him. First the sender raises a remote interrupt to get the token. The recipient goes to the idle mode again since the transfer is not completed yet. Once the transfer is completed the sender raise another remote interrupt to the recipient. This time the message is in the mailbox and the core can resume the execution.

```
  // Send interrupt to recipient
  e_irq_remote_raise(coreID, (E_SW_INT + SOFF));

  // Wait until the token is got
```

```
while(Me.core_number != *target_core_with_token)
      __asm__("IDLE;");

// Copy the data to the destination

// Send interrupt to recipient after the transfer
e_irq_remote_raise(coreID, (E_SW_INT + SOFF))
```

8.2 Inter-core communication

Each core is equipped with a DMA engine. Its function is to offload a core from data movement. It is capable of a double word transaction on every clock cycle at a 600Mhz frequency, which means a sustained data transfer rate of 8 GB/sec.

A high level of abstraction schematic for the epiphany chip is given in Figure 8.1, taken from [68].

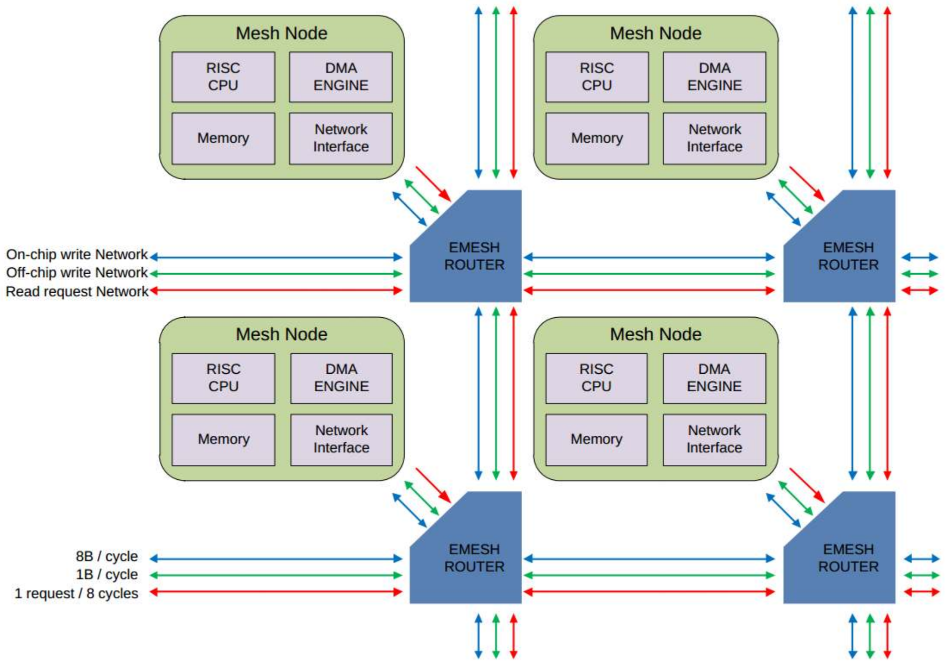


Figure 8.1: Epiphany schematic

The engine has two independent channels, *channel 0* and *channel 1*. *Channel 0* is used for sending and retrieving messages, while *channel 1* stays unused. It has been configured as a master and it generates an event interrupt when the transfer finishes. A data movement does not start until a previous data movement completes and the channel is free.

The configuration of the DMA to send or retrieve a message is shown below.

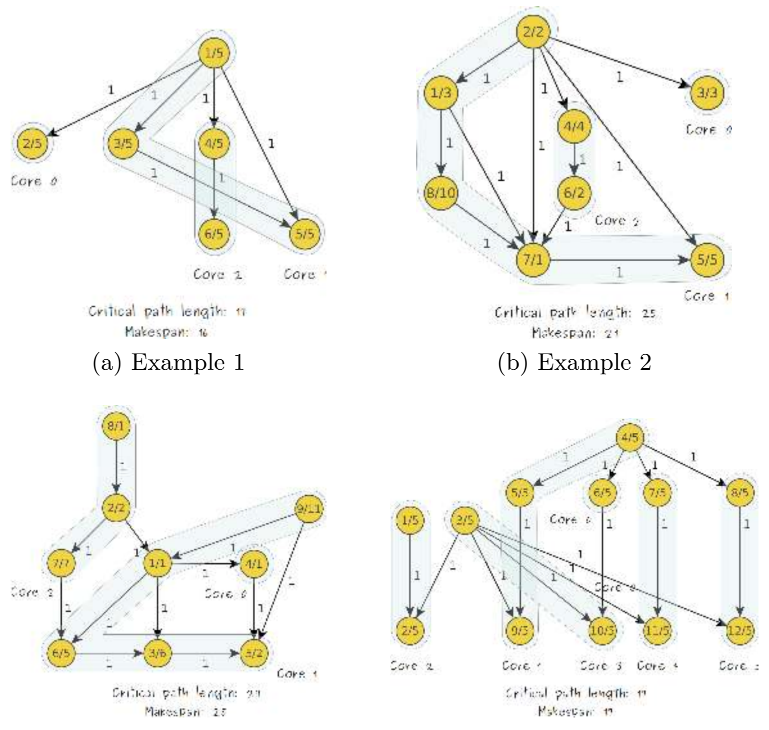
```
// Copy the data
e_tcb_t usr_tcb;
usr_tcb.config = E_DMA_ENABLE | E_DMA_MASTER | E_DMA_IRQEN
    | E_DMA_BYTE;
usr_tcb.inner_stride = 0x00010001;
usr_tcb.count = 0x10000 | size;
usr_tcb.outer_stride = 0x00000000;
usr_tcb.src_addr = source;
usr_tcb.dst_addr = destination;
e_dma_start(E_DMA_0, &usr_tcb);
```

Chapter 9

Results

This chapter will introduce the tests performed to the system and the implemented brake-by-wire application to help evaluate the performance. Then the collected data will be presented and analyzed.

9.1 Toolchain testing



(a) Example 1 (b) Example 2

(c) Example 3 (d) Example 4

Figure 9.1: Toolchain testing set

The toolchain testing procedure has consisted in introducing representative DAGs into the toolchain. The results have been compared with the expected result in order to verify the system (see Figure 9.1).

The first example is a 6-node DAG. The toolchain divides it into three clusters. It also calculates the critical path length and the makespan, 17 and 16 respectively. There are some interesting aspects. For instance, node 1 forks into four branches whereas two branches join into node 5. It can also be highlighted that there is more than one exit node, three in this case.

Example 2 is rather similar to example 1 but with a higher degree of complexity. The critical path length and the makespan are 25 and 21. Example 3 is more special because there is more than one entry node, two in this case. The critical path length and the makespan are 29 and 25.

Example 4 is the one used for the brake-by-wire application. The DAG contains three entry nodes and five exit nodes (one for the traction motor and four the brake motors at the wheels). The critical path length and the makespan are 17 and 17. In this case the makespan is not shorter than the critical path length. Linear clustering does not obtain an optimal solution. There exist other solutions that outperform the one obtained by the toolchain. The solution presented in Figure 9.2 has a makespan of 16, one time unit less.

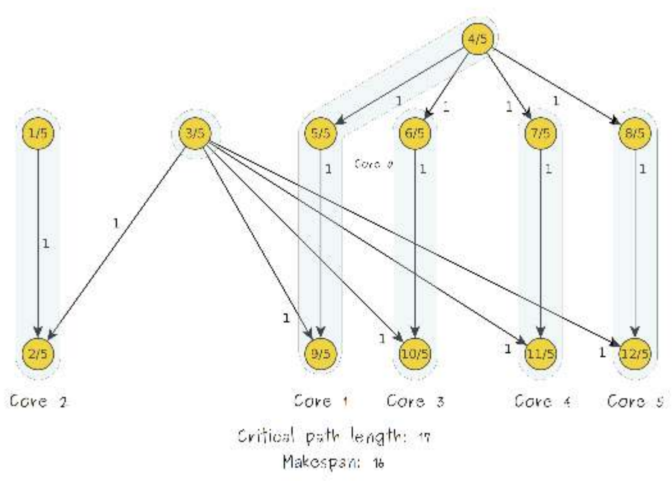


Figure 9.2: Alternative DAG with shorter makespan

The toolchain has been evaluated with 12 DAGs in total. Four representative examples have been shown in this section. In each example different aspects were evaluated. The covered aspects are join nodes, fork nodes, entry nodes, exit nodes, different tasks weights and different communication costs. A summary of the evaluated aspects is presented in Table 9.1. The last column of the table corresponds to the other 8 DAGs that are not presented in this report.

	Ex. 1	Ex. 2	Ex. 3	Ex. 4	Rest of DAGs
Join nodes	✓	✓	✓	✓	✓
Fork nodes	✓	✓	✓	✓	✓
More than one entry node	✗	✗	✓	✓	✓
More than one exit node	✓	✓	✗	✓	✓
Different weights	✗	✓	✓	✗	✓
Different costs	✗	✗	✗	✗	✓

Table 9.1: Summary of the evaluated aspects with each example DAG

9.2 Brake by wire application

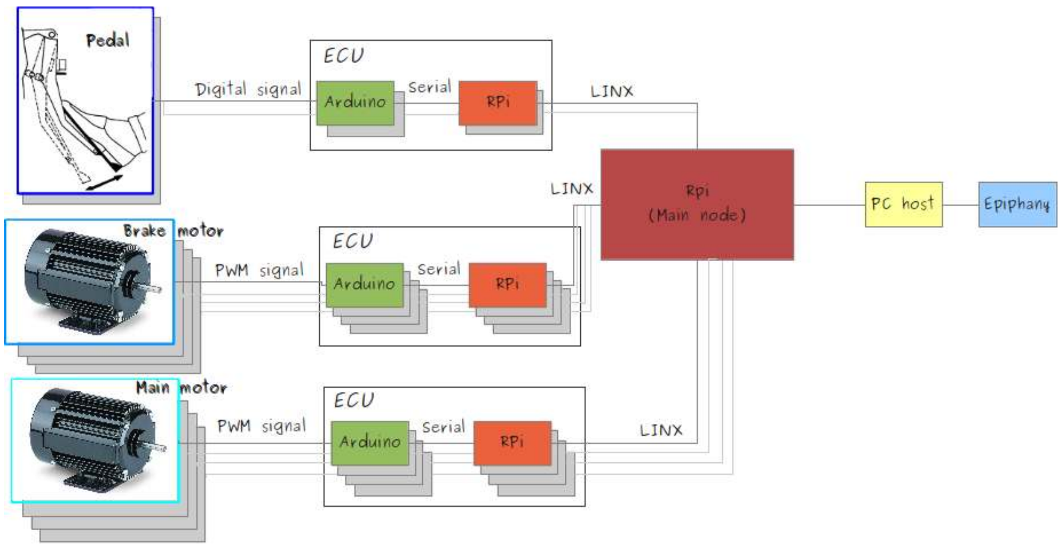


Figure 9.3: Brake-by-wire physical view

A brake-by-wire application has been implemented for the final demonstration platform. The application runs on the Epiphany E16 development board. The system's physical view is presented in Figure 9.3. It is made up of the following physical components.

- A brake pedal sensor and a throttle pedal sensor.
- A main motor and its controller.
- A brake motor and its controller.
- Four Arduino boards.

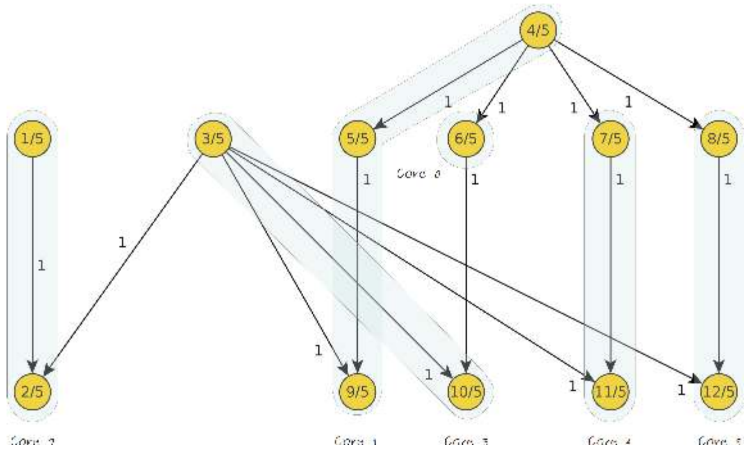


Figure 9.4: Brake-by-wire epiphany application

- Four Raspberry Pi.
- A desktop computer.
- A Epiphany E16 development board.

The inputs of the parallel application are the required brake torque, the required throttle torque and the wheel speeds in rpm. The inputs are obtained from the three sensors: the two pedal sensors and the hall sensors from the main motor. A pedal sensor consist of a potentiometer. The voltage value is read by a Arduino board and processed to get a percentage value. The motor controller has been configured to have an output whose voltage is proportional to the main motor speed. The voltage value is read by another Arduino board and processed to get a percentage value (100% represents 3400 rpm). The speed of the other three wheels is simulated with a random function.

Figure 9.4 shows a graph of the brake-by-wire application. The code for the Epiphany E16 development board is generated automatically by the toolchain. The calculation's part of each task has to be fill in by the programmer. The code used in the test is attached in Appendix C. A short description of the tasks is given below.

- **Task 1**: transforms the throttle percentage value to the real value.
- **Task 2**: calculates the electrical signal for the main motor.
- **Task 3**: transforms the brake percentage value to the real value.
- **Task 4**: calculates the speed of the car (average speed of the four wheels).
- **Task 5, 6, 7 and 8**: calculates the slip value of one wheel.

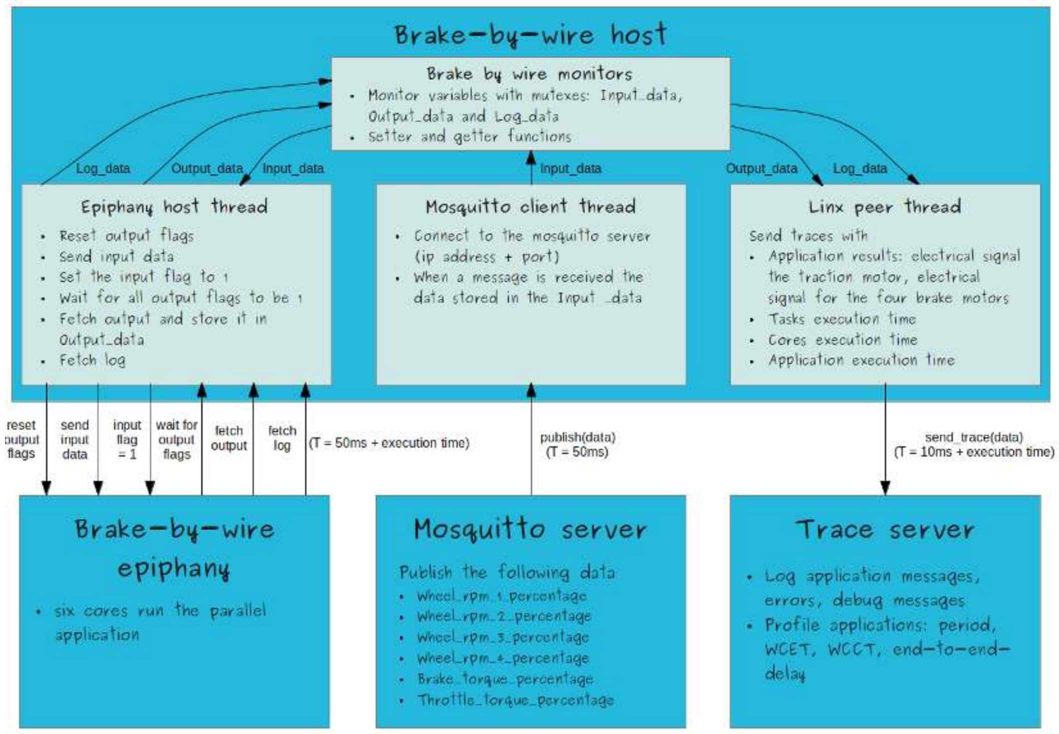

Brake-by-wire host
Brake by wire monitors
• Monitor variables with mutexes: Input_data, Output_data and Log_data
• Setter and getter functions
Log_data
Output_data
Input_data
Input_data
Output_data
Log_data
Epiphany host thread
• Reset output flags
• Send input data
• Set the input flag to 1
• Wait for all output flags to be 1
• Fetch output and store it in Output_data
• Fetch log
Mosquitto client thread
• Connect to the mosquitto server (ip address + port)
• When a message is received the data stored in the Input_data
Linx peer thread
Send traces with
• Application results: electrical signal the traction motor, electrical signal for the four brake motors
• Tasks execution time
• Cores execution time
• Application execution time
reset output flags
send input data
input flag = 1
wait for output flags
fetch output
fetch log
(T = 50ms + execution time)
publish(data) (T = 50ms)
send_trace(data) (T = 10ms + execution time)
Brake-by-wire epiphany
• six cores run the parallel application
Mosquitto server
Publish the following data
• Wheel_rpm_1_percentage
• Wheel_rpm_2_percentage
• Wheel_rpm_3_percentage
• Wheel_rpm_4_percentage
• Brake_torque_percentage
• Throttle_torque_percentage
Trace server
• Log application messages, errors, debug messages
• Profile applications: period, WCET, WCCT, end-to-end-delay

Figure 9.5: Brake-by-wire host application

- **Task 9, 10, 11 and 12**: calculates the electrical signal for the brake motor of one wheel.

Both the execution time of the tasks and the communication time between them are unknown. Therefore it is assumed that all the tasks have the same execution time, 5 time units. The second assumption is that the communication time is always the same, 1 time unit.

On the host computer there is other deployed program (see Figure 9.5). This program has three monitor variables protected by pthread mutexes: *Input_data*, *Output_data* and *Log_data*. There are also three threads: *epiphany_host_thread*, *mosquitto_client_thread* and *linx_peer_thread*. The *mosquitto_client_thread* is in charged of updating the *Input_data* with the information published by the Mosquitto server, which runs on the main Raspberry Pi node. The *epiphany_host_thread* is in charged of starting the execution of epiphany and updating the *Output_data* and *Log_data* when the execution has completed. The *linx_peer_thread* sends the *Output_data* and *Log_data* to the trace server, which runs on the main Raspberry Pi node.

9.3 Collected data

9.3.1 Power consumption's data

The Epiphany chip is equipped with a current sense resistor. The value of the resistor is 0.02Ω. The sense gain of the system is 50. Therefore, the voltage measured on the measurement point $V_{core(A)}$ is equal to the core current. The voltage supply for Epiphany is V_{core}, 1089mV. The power consumption is given by the following expression.

$$\begin{aligned} P_{Epiphany} &= V_{core} \cdot I_{core} \\ &= V_{core} \cdot V_{core(A)} = 1.089 \cdot V_{core(A)} \end{aligned}$$

Figure 9.6 shows the power consumption in Epiphany as a function of the idle cores. The power consumption when all the cores are in idle is 382.239mW. The power saving when idling a core is between 21.79mW and 22.869mW. It means a 5.98% of the static consumption per core and about a 98% with all the cores. The program used to test Epiphany is attached in Appendix A.

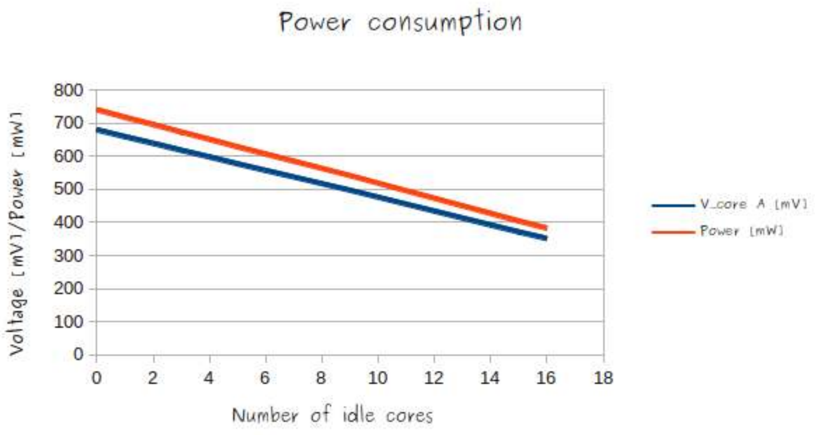


Figure 9.6: Epiphany power consumption

9.3.2 Timers' data

The profiler module has to kind of functions: *start_core_profiler/start_task_profiler* and *record_core_time/record_task_time*. A core is using timer 1 to profile and a task is using timer 0 to profile.

Some tests have been performed to measure the accuracy of the profiling system. In the first test the time when no operation is performed is measured. *time_1* is 101ns.

```
start_core_profiler();
```

```
log_t time_1 = record_core_time();
```

The next test starts the other timer. *time_1* is 245ns. Therefore the time necessary for executing *start_task_profiler* is 144ns (101ns subtracted).

```
start_core_profiler();
start_task_profiler();
log_t time_1 = record_core_time();
```

The next test starts the task timer and record a time. *time_1* is 105ns and *time_2* is 6008ns (if task and core functions are inverted the result is 100ns and 5986ns). Therefore the time to start a timer and record a value is 5907ns (101ns subtracted). Then 144ns are subtracted for *start_task_profiler*, 5763ns. The counter counted 105ns. It means that the timer did not start until 5658ns (105ns subtracted).

```
start_core_profiler();
start_task_profiler();
log_t time_1 = record_task_time();
log_t time_2 = record_core_time();
```

The last test consists of recording the task time twice. *time_1* is 105ns, *time_2* is 5870ns and *time_3* is 11658ns. Therefore *record_task_time* takes 5788ns. This seems to be correct since 5788ns is very close to 5658ns plus 105ns (5763ns).

```
start_core_profiler();
start_task_profiler();
log_t time_1 = record_task_time();
log_t time_2 = record_task_time();
log_t time_3 = record_core_time();
```

The times can be summarized as follows:

- **start_task_profiler**: 144ns.
- **timer start time after start_task_profiler**: 5658ns.
- **record_task_time**: 5788ns.

9.3.3 Single core application's data

In order to evaluate the performance of the parallel application it is necessary to collect data of an equivalent serial application (see Figure 9.7). The code is attached in Appendix B.

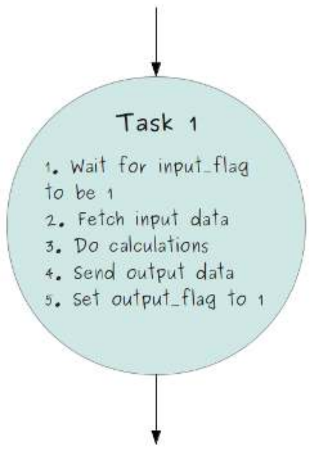


Figure 9.7: Brake-by-wire application (single core)

Both core profiler and task profiler are used at the same time. The task profiler only measures the time for the calculations whereas the core profiler measures the time to fetch the *input*, send the *output* and put the *output_flag* to 1 as well. The measurement of the core profiler is affected by the task profiler measurement. Therefore another experiment is performed without the task profiler. The result is around 6000ns less, see 9.2.

	Task profiler	**Core profiler**	**Core prof. alone**
Mean time [ns]	13368	22627	16623

Table 9.2: Single core application's task and core profiler

Other experiments are performed in order to identify where the other 3000ns are spent. The results are shown in Table 9.3.

	Mean time [ns]
Before fetching input data	123
After fetching input data	3128
Before sending output data	16529
After sending output data	16602
After setting output flag to 1	16623

Table 9.3: Single core application's timing

The times can be summarized as follows:

- **Calculations**: 13368ns.

- **Application**: 16623ns.
- **Fetching input data**: 3105ns.
- **Sending output data**: 73ns.
- **Setting output flag to 1**: 21ns.

The energy consumption is:

$$\begin{aligned} Energy &= \left(\frac{Power_{idle}}{16} + Power_{1core}\right) \cdot time \\ &= \left(\frac{382mW}{16} + 22mW\right) \cdot 16623ns = 0.762\mu J \end{aligned}$$

9.3.4 Parallel application's data

The execution times of the cores are shown in Table 9.4.

	Core 0	**Core 1**	**Core 2**	**Core 3**	**Core 4**	**Core 5**
Mean time [ns]	35126	45399	24182	55508	52824	60908

Table 9.4: Parallel application's core profiler

The total execution time of the application is 60908ns. The energy consumption of the application is calculated as follows.

$$\begin{aligned} Energy &= \sum \left(\frac{Power_{idle}}{16} + Power_{1core}\right) \cdot time_{core_i} \\ &= \sum \left(\frac{382mW}{16} + 22mW\right) \cdot time_{core_i} \\ &= 12.567\mu J \end{aligned}$$

When the idle capability is included in the code, the energy consumption is decreased. The cores get into the idle mode because they are blocked waiting for a message. There is an option in the timers to count the number of idle cycles. The results are shown in Table 9.5.

	Core 0	**Core 1**	**Core 2**	**Core 3**	**Core 4**	**Core 5**
Idle time [ns]	7589	2906	6141	12233	15403	26042

Table 9.5: Number of idle cycles

The idle time needs to be included in the calculations of the energy consumption. The following equation is used.

$$
\begin{aligned}
Energy &= \sum \frac{Power_{idle}}{16} \cdot time_{core_i} + Power_{1core} \cdot (time_{core_i} - idle\_time_{core_i}) \\
&= \sum \frac{382mW}{16} \cdot time_{core_i} + 22mW \cdot (time_{core_i} - idle\_time_{core_i}) \\
&= 11.020\mu J
\end{aligned}
$$

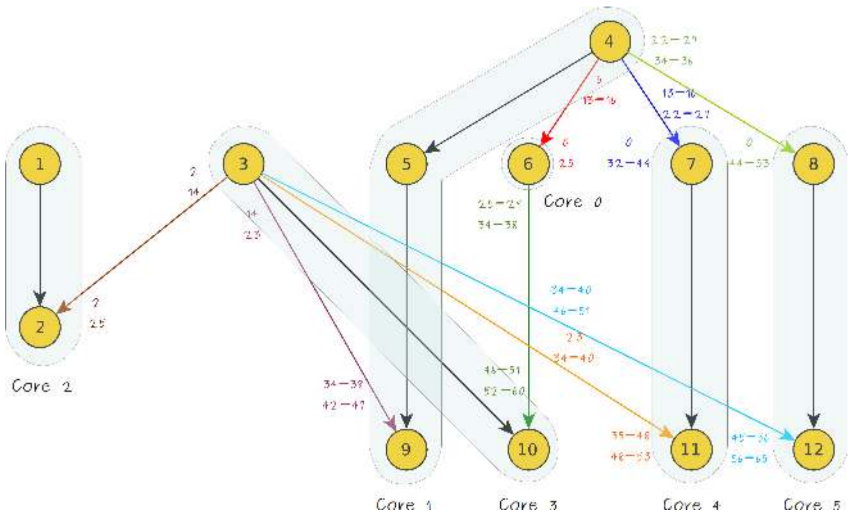


Figure 9.8: Brake-by-wire application timing

Figure 9.8 shows the timing of the parallel application measured with the timers. The colored edges represent inter-core communication and make use of the mailbox system. At the departing point there are two rows. The first row contains the system time in microseconds before sending the message. The second row contains the system time after sending the message. At the arrival point there are another two rows. The first row contains the system time at which the core first tries to retrieve the message. The second row contains the system time after retrieving the message. Sometimes the measured value is not constant, then the minimum and maximum measured value are put instead.

Table 9.6 shows the time elapsed in each task.

	Mean time [ns]
Task 1	350
Task 2	190
Task 3	373
Task 4	2853
Task 5	2301
Task 6	2301
Task 7	2301
Task 8	2308
Task 9	990
Task 10	988
Task 11	988
Task 12	988

Table 9.6: Parallel application's task profiler

9.3.5 Parallel application with task duplication's data

Another parallel application has been implemented (see Figure 9.9). The code is attached in Appendix D. The functionality and the calculations are the same. It is based on a different approach, task duplication. The goal is to avoid inter-core communication. A short description of the tasks is given below.

- **Task 1**: transforms the brake percentage value to the real value.
- **Task 2**: transforms the throttle percentage value to the real value and calculates the electrical signal for the main motor (if the brake is more than 10% activated the throttle is deactivated).
- **Task 3**: calculates the speed of the car (average speed of the four wheels).
- **Task 4**: calculates the slip value of one wheel and the electrical signal for the brake motor of the same wheel.
- **Task 5**: calculates the slip value of one wheel and the electrical signal for the brake motor of the same wheel.
- **Task 6**: calculates the slip value of one wheel and the electrical signal for the brake motor of the same wheel.
- **Task 7**: calculates the slip value of one wheel and the electrical signal for the brake motor of the same wheel.

The execution times of the cores are shown in Table 9.7.

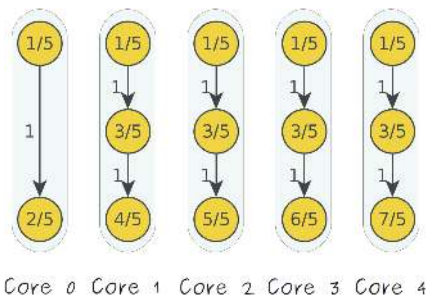


Figure 9.9: Brake-by-wire application with task duplication

	Core 0	**Core 1**	**Core 2**	**Core 3**	**Core 4**
Mean time [ns]	2813	8985	9006	9017	9019

Table 9.7: Parallel application with task duplication's core profiler

The total execution time of the application is 9019ns. In this case tasks do not get blocked and the idle capability is not used. The energy consumption of the application is calculated as follows.

$$\begin{aligned} Energy &= \sum \left(\frac{Power_{idle}}{16} + Power_{1core} \right) \cdot time_{core_i} \\ &= \sum \left(\frac{382mW}{16} + 22mW \right) \cdot time_{core_i} \\ &= 1.781\mu J \end{aligned}$$

Table 9.8 shows the time elapsed in each task.

	Mean time [ns]
Task 1	355
Task 2	430
Task 3	2466
Task 4	3170
Task 5	3170
Task 6	3170
Task 7	3170

Table 9.8: Parallel application with task duplication's task profiler

9.4 Data analysis

9.4.1 Timers analysis

The data collected from the timers is very relevant for the analysis of the other parts' data, since they constitute the profiling system. The performance is worse than expected because it takes around $6\mu s$ to read a timer. The effects of these results were very severe. The profiler module had to be removed in order not to affect the performance of the brake-by-wire application. It was not possible to compare the serial application with the parallel application otherwise.

In order not to perturb the measurements a timer was only used once in the application. This made the measurement process much more tedious.

9.4.2 Single core application analysis

This experiment highlighted that reading from shared DRAM is more time consuming than writing in shared DRAM. It took about $3\mu s$ to read and about $0.1\mu s$ to write. The difference is about one order of magnitude.

9.4.3 Parallel application analysis

The parallel application has an execution time that is 266% greater than the one from the serial application and is 1346% more energy consuming using the idle mode. The energy saving achieved by using the idle instruction is about 12.3%. The performance is far worse than the one from the serial application.

The explanation is that the communication time is greater than the computation time. The brake-by-wire application is very simple and the computation time of the tasks is very small. Therefore the overhead of the communication system cannot be neglected. Node 3 in Figure 9.8 needs around $10\mu s$ to transmit a message to other cores. It has to transmit the same message to four different cores, thus the communication delay is around $40\mu s$. Node 4 is in a similar situation. It has to transmit a message to three different cores. The transmission cannot be done in parallel but sequentially. The order in which the messages are sent affects greatly to the performance of the application, because the delays are accumulated.

Other point to be remarked is that the mailbox is a complex and non-time-deterministic mechanism. For instance, core 1 is receiving the message e_{39} from core 3 while it is sending the message e_{46} to core 0. Receiving the message requires to raise an interrupt in core 1 from core 3. That it is why there is a variability in the times.

The mailbox is a shared resource that can lock the communication. This fact adds non deterministic extra delays. For example, core 3 has to wait while sending message e_{311} to core 4. This happens because core 1 is already sending message e_{47} to core 4.

9.4.4 Parallel application with task duplication analysis

Task duplication is a good alternative for applications with low computation compared to the communication. Instead of executing a task and transmitting the results to the successors, the task is replicated in each core and the results are got locally. The mailbox is not used. In this case the application needs 45.7% less time but 133.7% more energy.

9.5 Requirements evaluation

An evaluation of the requirements list given in Section 1.3 and analyzed in Section 5.1 is performed below:

REQ_1 The scheduler shall cope with both serial and parallel applications.
This requirement has been fulfilled. A serial application can be modeled as a single node parallel application.

REQ_2 The scheduler shall do automatic parallelization of the application. Parallelization shall be transparent to the programmer.
This requirement has been partially fulfilled. As mentioned in Section 5.1, automatic parallelization takes care of the application decomposition and the handling of dependencies. In the implemented prototype, the programmer has to draw the DAG with the execution and communication times of the tasks as a first step. In this sense, the first part of the automation process has not been accomplished. However, the toolchain generates the code for the target which automatically handles the inter-task communication. The programmer only has to fill in the content of the tasks and the communication interfaces between them.

The profiler module improves the parallelization of the application by feeding back the actual execution and communication times of the tasks. This means that the parallelization may change once the real execution and communication times are used in a second iteration of the toolchain.

REQ_3 The scheduler shall parallelize tasks as much as possible.
REQ_4 The scheduler shall avoid communication as much as possible.
REQ_6 The scheduler shall generate a schedule with the minimum execution time.
This requirement has been partially fulfilled. The toolchain is limited by the scheduling algorithm, linear clustering. Better results could be achieved with other algorithms. The toolchain could use several algorithms and choose the one that gives the shortest makespan.

REQ_5 The scheduler shall provide to an application an end-to-end guarantee.

This requirement has been fulfilled. The toolchain provides the theoretical makespan of the application. Then there is a profiler module to get the real computation weight of the nodes and the communication costs to feed back the toolchain. However, it has been proved that the profiler module can alter the performance of the application. It is especially noticeable if the computation times of the tasks are very small.

REQ_7 The scheduler shall be energy efficient.
This requirement has been partially fulfilled. The use of the idle mode saves energy compared to the case of not using the idle mode. However, energy savings have not been achieved by parallelizing. Parallelization can only reduce the execution time of the brake-by-wire application.

REQ_8 The scheduler shall be target independent. The algorithm shall be easily portable to another platform.
This requirement has been partially fulfilled. The graph editor, the xgml parser module and the scheduler module with the linear clustering algorithm are target independent. However, the code generator module is dependent on the developing board. This module should be implemented again for another target. The input interface of that module would be the same: the nodes list, the edges list and the clusters lists.

9.6 Summary

The toolchain system has shown to be a useful tool for developing parallel applications. Linear clustering was not the best scheduling algorithm for the brake-by-wire application because the makespan was 17 and an alternative solution was presented in this chapter that had a lower makespan. Apart from that, the programmer can generate error-free parallel code very quickly.

Regarding the Epiphany code, the mailbox system is quite heavy for the brake-by-wire application because the granularity of the tasks is very fine. The execution time of the serial application is around $16\mu s$ and transmitting a message costs around $10\mu s$. Therefore no more than two messages can be sent in a non-parallel way to get a smaller execution time. Besides, messages are not transmitted in parallel from the same node, but sequentially. The conclusion is that this communication system is suitable for coarse granularity tasks, where the communication overhead does not affect so much.

In applications with fine granularity tasks is advisable to go for other approaches such as task duplication, where inter-core communication is avoided by doing extra computations. A comparison of the parallel applications with the serial application is shown in the Table 9.9.

	Execution time	**Energy consumption**
Parallel app	+266%	+1346%
Parallel app with task duplication	-45.7%	+133.7%

Table 9.9: Performance comparison between brake-by-wire applications

An striking result is that even though more calculations are being done with the task duplication approach, the energy consumption is far less. This is due to the fact that the execution time is smaller. In the first approach the tasks spend a lot of time in the idle mode waiting for a message, which consumes a lot of energy. There is also extra overhead from the communication system that make the whole system less energy efficient.

Chapter 10

Conclusions and future work

10.1 Conclusions

The goal of this project was to research scheduling algorithms for multi-core embedded systems and to make an implementation of a scheduler for a parallel application. In the analytical phase several algorithms have been studied and compared. The linear clustering algorithm was chosen to be implemented.

In the practical phase a toolchain for programming parallel applications was implemented. The target platform was the Epiphany E16 development board. Different software modules for that target had to be implemented: *epiphany module*, *file system module*, *interrupts module*, *mailbox module*, *host communication module* and *profiler module*.

Several experiments were carried out in order to help evaluate the performance of the system. Parallel computing should only be used when there is enough computation to be parallelized. If there is little parallelization to be done, the mailbox system is not worthy to use. However, the execution can still be sped up by using algorithms based on task duplication. A mailbox system is worthy to use in applications with a lot of parallelization because the communication overhead can then be neglected.

10.2 Limitations

At the current stage there are some limitations in the system:

- An application with more than 16 clusters cannot be handled since only one cluster is assigned to a core.
- No more than 32 blocks or 32 messages can be stored in the mailbox.

10.3 Future work

Some lines of future research are:

- Evaluation of the toolchain on other applications with different characteristics, such as execution times, communication times, CCR or DAG topology.
- Design of a more scalable system (limitation to 16 clusters, limitation to 32 messages simultaneously in the mailbox).
- Implementation of other scheduling algorithms in the toolchain. Selection of the algorithm with the shortest makespan for each case.
- Automation of the generation of DAGs by extracting parallelism from the source code of serial applications. This could be done by profiling the application at run-time (hotspots in the code, accesses to the memory) and using machine learning techniques.
- Implementation of an automatic feedback mechanism with the profiled data. It would get hold of the real execution and communication times of the application, assuming constant execution times. The original DAG would be updated with these values increasing its fidelity. These feedback mechanism would also validate the parameters in each execution.
- Dealing with varying execution and communication times in the system.
- Implementation of a multicast service in order not to send the same message several times.
- Dynamic loading of tasks and evaluation of its performance.

Appendix A

Epiphany power consumption test application

In this appendix the program to measure the power consumption of the Epiphany chip as a function of the number of idle cores is attached. The program has been run 17 times, changing the number of cores in idle. The cores that are not in idle state perform operations in the Floating-Point Unit (FPU). The timers are also started in order to consume more.

Power consumption test

```
int mc_core_common_go()
{
  int status = 0;

  int coreId = e_get_coreid();

  if(core_num(coreId)>=0)
  {
    asm volatile ("idle");
  } else {
    double i , k , l , m , n;

    l =3.14;
    m = 3.14567;
    n = 3.014;
    k = 0;
    i = 1.34;

    while(1)
    {
      m = (i * 3.14) / 0.57 + 32.545;
```

```
        k = i / 1.746 - 3.23;
        l = (m * 3.14) / 0.57 + 32.545;
        i = m / 1.746 - 3.23;
        n = (i * 3.14) / 0.57 + 32.545;
        i = i * i / 1.746 - 3.23;
        k = (i * 3.14) / 0.57 + 32.545;
        i = i / 1.746 - 3.23;
        m = (n * 3.14) / 0.57 + 32.545 * l;
        l = l * l / 1.746 - 3.23;
        i = (i * m * 3.14) / 0.57 + 32.545;
        k = m / 1.746 - 3.23;
        l = (k * 3.14) / 0.57 + 32.545;
        m = k / 1.746 - 3.23;

        e_ctimer_set(E_CTIMER_0, E_CTIMER_CLK, 100000000);
        e_ctimer_set(E_CTIMER_1, E_CTIMER_CLK, 100000000);
      }
    }

    return status;
}
```

Appendix B

Brake-by-wire application: single core

In this appendix a serial version of the brake-by-wire application is attached.

Single core application

```
int driver_required_brake_torque =     (MAX_BRAKE_TORQUE *
    input_1.brake_pedal_percentage) / 100;
int driver_required_throttle_torque =
    (MAX_THROTTLE_TORQUE *
    input_1.throttle_pedal_percentage) / 100;

int wheel_torque = -1;
if(MAX_BRAKE_TORQUE/10 > driver_required_brake_torque)
{
  wheel_torque = driver_required_throttle_torque / 4;
} else {
  wheel_torque = 0;
}
output_1.el_signal_traction_motor = (wheel_torque *
    245)/750 + 10;

int wheel_1_speed_rpm = (MAX_RPM *
    input_1.wheel_1_speed_rpm_percentage) / 100;
int wheel_2_speed_rpm = (MAX_RPM *
    input_1.wheel_2_speed_rpm_percentage) / 100;
int wheel_3_speed_rpm = (MAX_RPM *
    input_1.wheel_3_speed_rpm_percentage) / 100;
int wheel_4_speed_rpm = (MAX_RPM *
    input_1.wheel_4_speed_rpm_percentage) / 100;

double vehicle_speed_mps = (wheel_1_speed_rpm +
    wheel_2_speed_rpm + wheel_3_speed_rpm +
wheel_4_speed_rpm)
      * WHEEL_RADIOUS * 2 * 3.14 / 1000 / 60 / 4;

double wheel_1_slip = (vehicle_speed_mps - wheel_1_speed_rpm * WHEEL_RADIOUS * 2 * 3.14 / 60 / 1000) / vehicle_speed_mps;
if(0 > wheel_1_slip)
  wheel_1_slip = 0;
double wheel_2_slip = (vehicle_speed_mps - wheel_2_speed_rpm * WHEEL_RADIOUS * 2 * 3.14 / 60 / 1000) / vehicle_speed_mps;
if(0 > wheel_2_slip)
  wheel_2_slip = 0;
double wheel_3_slip = (vehicle_speed_mps - wheel_3_speed_rpm * WHEEL_RADIOUS * 2 * 3.14 / 60 / 1000) / vehicle_speed_mps;
if(0 > wheel_3_slip)
  wheel_3_slip = 0;
double wheel_4_slip = (vehicle_speed_mps - wheel_4_speed_rpm * WHEEL_RADIOUS * 2 * 3.14 / 60 / 1000) / vehicle_speed_mps;
if(0 > wheel_4_slip)
  wheel_4_slip = 0;

int abs_brake_torque_1, abs_brake_torque_2, abs_brake_torque_3, abs_brake_torque_4 = -1;

if((10 <= vehicle_speed_mps*3600/1000) && (0.2 < wheel_1_slip))
{
  abs_brake_torque_1 = 0;
} else {
  abs_brake_torque_1 = driver_required_brake_torque / 4;
}
output_1.el_signal_brake_motor_1 = (abs_brake_torque_1 * 245)/750 + 10;
if((10 <= vehicle_speed_mps*3600/1000) && (0.2 < wheel_2_slip))
{
  abs_brake_torque_2 = 0;
} else {
  abs_brake_torque_2 = driver_required_brake_torque / 4;
}
output_1.el_signal_brake_motor_2 = (abs_brake_torque_2 *
    245)/750 + 10;
if((10 <= vehicle_speed_mps*3600/1000) && (0.2 <
    wheel_3_slip))
{
  abs_brake_torque_3 = 0;
} else {
  abs_brake_torque_3 = driver_required_brake_torque / 4;
}
output_1.el_signal_brake_motor_3 = (abs_brake_torque_3 *
    245)/750 + 10;
if((10 <= vehicle_speed_mps*3600/1000) && (0.2 <
    wheel_4_slip))
{
  abs_brake_torque_4 = 0;
} else {
  abs_brake_torque_4 = driver_required_brake_torque / 4;
}
output_1.el_signal_brake_motor_4 = (abs_brake_torque_4 *
    245)/750 + 10;
```

Appendix C

Brake-by-wire application: tasks

In this appendix the calculations' sections of the tasks are listed. Task 6, 7 and 8 are omitted due to their similarity to task 5. Task 10, 11 and 12 are omitted due to their similarity to task 9.

Task 1

```
edge_12.driver_required_throttle_torque =
    (MAX_THROTTLE_TORQUE *
    input_1.throttle_pedal_percentage) / 100;
```

Task 2

```
int wheel_torque = -1;
if(MAX_BRAKE_TORQUE/10 >
    edge_32.driver_required_brake_torque)
{
  wheel_torque = edge_12.driver_required_throttle_torque /
      4;
} else {
  wheel_torque = 0;
}
output_2.el_signal_traction_motor = (wheel_torque *
    245)/750 + 10;
```

Task 3

```
int driver_required_brake_torque = (MAX_BRAKE_TORQUE *
    input_3.brake_pedal_percentage) / 100;
edge_32.driver_required_brake_torque =
    driver_required_brake_torque;
```

```
edge_39.driver_required_brake_torque =
    driver_required_brake_torque;
edge_310.driver_required_brake_torque =
    driver_required_brake_torque;
edge_311.driver_required_brake_torque =
    driver_required_brake_torque;
edge_312.driver_required_brake_torque =
    driver_required_brake_torque;
```

Task 4

```
int wheel_1_speed_rpm = (MAX_RPM *
    input_4.wheel_1_speed_rpm_percentage) / 100;
int wheel_2_speed_rpm = (MAX_RPM *
    input_4.wheel_2_speed_rpm_percentage) / 100;
int wheel_3_speed_rpm = (MAX_RPM *
    input_4.wheel_3_speed_rpm_percentage) / 100;
int wheel_4_speed_rpm = (MAX_RPM *
    input_4.wheel_4_speed_rpm_percentage) / 100;

double vehicle_speed_mps = (wheel_1_speed_rpm +
    wheel_2_speed_rpm + wheel_3_speed_rpm +
    wheel_4_speed_rpm)
  * WHEEL_RADIOUS * 2 * 3.14 / 1000 / 60 / 4;

edge_45.wheel_1_speed_rpm = wheel_1_speed_rpm;
edge_45.vehicle_speed_mps = vehicle_speed_mps;
edge_46.wheel_2_speed_rpm = wheel_2_speed_rpm;
edge_46.vehicle_speed_mps = vehicle_speed_mps;
edge_47.wheel_3_speed_rpm = wheel_3_speed_rpm;
edge_47.vehicle_speed_mps = vehicle_speed_mps;
edge_48.wheel_4_speed_rpm = wheel_4_speed_rpm;
edge_48.vehicle_speed_mps = vehicle_speed_mps;
```

Task 5

```
edge_59.vehicle_speed_mps = edge_45_copy.vehicle_speed_mps;

double wheel_1_slip = (edge_45_copy.vehicle_speed_mps -
    edge_45_copy.wheel_1_speed_rpm * WHEEL_RADIOUS * 2 *
    3.14 / 60 / 1000) / edge_45_copy.vehicle_speed_mps;
if(0 <= wheel_1_slip)
{
  edge_59.wheel_1_slip = wheel_1_slip;
} else {
  edge_59.wheel_1_slip = 0;
}
```

Task 9

```
int abs_brake_torque = -1;
if((10 <= edge_59_copy.vehicle_speed_mps*3600/1000) && (0.2
    < edge_59_copy.wheel_1_slip))
{
  abs_brake_torque = 0;
} else {
  abs_brake_torque = edge_39.driver_required_brake_torque /
      4;
}
output_9.el_signal_brake_motor_1 = (abs_brake_torque *
    245)/750 + 10;
```

Appendix D

Brake-by-wire application with task duplication: tasks

In this appendix the calculations' sections of the tasks are listed. Task 5, 6 and 7 are omitted due to their similarity to task 4.

Task 1

```
int driver_required_brake_torque = (MAX_BRAKE_TORQUE *
    input_1.brake_pedal_percentage) / 100;
edge_1.driver_required_brake_torque =
    driver_required_brake_torque;
```

Task 2

```
int wheel_torque = -1;
int driver_required_throttle_torque = (MAX_THROTTLE_TORQUE
    * input_2.throttle_pedal_percentage) / 100;
if(MAX_BRAKE_TORQUE/10 >
    edge_1.driver_required_brake_torque)
{
  wheel_torque = driver_required_throttle_torque / 4;
} else {
  wheel_torque = 0;
}
output_2.el_signal_traction_motor = (wheel_torque *
    245)/750 + 10;
```

Task 3

```
int wheel_1_speed_rpm = (MAX_RPM *
    input_3456.wheel_1_speed_rpm_percentage) / 100;
```

```
int wheel_2_speed_rpm = (MAX_RPM * input_3456.wheel_2_speed_rpm_percentage) / 100;
int wheel_3_speed_rpm = (MAX_RPM * input_3456.wheel_3_speed_rpm_percentage) / 100;
int wheel_4_speed_rpm = (MAX_RPM * input_3456.wheel_4_speed_rpm_percentage) / 100;

double vehicle_speed_mps = (wheel_1_speed_rpm + wheel_2_speed_rpm + wheel_3_speed_rpm + wheel_4_speed_rpm)
      * WHEEL_RADIOUS * 2 * 3.14 / 1000 / 60 / 4;

edge_34.wheel_1_speed_rpm = wheel_1_speed_rpm;
edge_34.vehicle_speed_mps = vehicle_speed_mps;
```

Task 4

```
double wheel_1_slip = (edge_34.vehicle_speed_mps - edge_34.wheel_1_speed_rpm * WHEEL_RADIOUS * 2 * 3.14 / 60 / 1000) / edge_34.vehicle_speed_mps;
if(0 > wheel_1_slip)
{
  wheel_1_slip = 0;
}

int abs_brake_torque = -1;
if((10 <= edge_34.vehicle_speed_mps*3600/1000) && (0.2 < wheel_1_slip))
{
  abs_brake_torque = 0;
} else {
  abs_brake_torque = edge_1.driver_required_brake_torque / 4;
}
output_4.el_signal_brake_motor_1 = (abs_brake_torque * 245)/750 + 10;
```

Bibliography

[1] T. Hagras and J. Janeek, “Static vs. dinamic list-scheduling performance comparison,” *Acta Polytechnica*, vol. 43, no. 6, 2003.

[2] J. Markoff, “Intel’s big shift after hitting technical wall,” *New York Times*, 2004.

[3] M. J. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. I. August, “Revisiting the sequential programming model for multi-core,” in *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pp. 69–84, IEEE, 2007.

[4] C. E. Frank J. Bartos, P.E., “Growing applications for multi-core processors.” http://www.controleng.com/index.php?id=483&cHash=081010&tx_ttnews%5Btt_news%5D=42947.

[5] D. Geer, “Chip makers turn to multicore processors,” *Computer*, vol. 38, no. 5, pp. 11–13, 2005.

[6] C. E. Frank J. Bartos, P.E., “Computing power: Multi-core processors help industrial automation.” http://www.controleng.com/index.php?id=483&cHash=081010&tx_ttnews[tt_news]=43096.

[7] I. C. Ian Gilvarry, “Insights on multi-core processors.” http://www.controleng.com/single-article/insights-on-multi-core-processors/db06fb5f14b0f011155f96c80259038b.html.

[8] Y. Choi, Y. Lin, N. Chong, S. Mahlke, and T. Mudge, “Stream compilation for real-time embedded multicore systems,” in *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*, pp. 210–220, IEEE, 2009.

[9] C. LUCAR, *Dataflow Programming for Systems Design Space Exploration for Multicore Platforms.* PhD thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2011.

[10] M. J. Flynn, “Very high-speed computing systems,” *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.

[11] C. W. page, "Autosar." http://www.autosar.org.

[12] A. Sangiovanni-Vincentelli and M. Di Natale, "Embedded system design for automotive applications," *Computer*, vol. 40, no. 10, pp. 42–51, 2007.

[13] S. Chakraborty, M. Di Natale, H. Falk, M. Lukasiewycz, and F. Slomka, "Timing and schedulability analysis for distributed automotive control applications," in *Proceedings of the ninth ACM international conference on Embedded software*, pp. 349–350, ACM, 2011.

[14] S. K. Jena and M. Srinivas, "On the suitability of multi-core processing for embedded automotive systems," in *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2012 International Conference on*, pp. 315–322, IEEE, 2012.

[15] A. Monot, N. Navet, B. Bavoux, F. Simonot-Lion, *et al.*, "Multicore scheduling in automotive ecus," in *Embedded Real Time Software and Systems-ERTSS 2010*, 2010.

[16] N. Navet, A. Monot, B. Bavoux, F. Simonot-Lion, *et al.*, "Multi-source and multicore automotive ecus-os protection mechanisms and scheduling," in *International Symposium on Industrial Electronics-ISIE 2010*, 2010.

[17] A. Monot, N. Navet, B. Bavoux, and F. Simonot-Lion, "Multisource software on multicore automotive ecus—combining runnable sequencing with task scheduling," *Industrial Electronics, IEEE Transactions on*, vol. 59, no. 10, pp. 3934–3942, 2012.

[18] H. Kopetz, R. Obermaisser, C. El Salloum, and B. Huber, "Automotive software development for a multi-core system-on-a-chip," in *Software Engineering for Automotive Systems, 2007. ICSE Workshops SEAS'07. Fourth International Workshop on*, pp. 2–2, IEEE, 2007.

[19] S. Schliecker, J. Rox, M. Negrean, K. Richter, M. Jersak, and R. Ernst, "System level performance analysis for real-time automotive multicore and network architectures," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 7, pp. 979–992, 2009.

[20] N. Böhm, D. Lohmann, W. Schröder-Preikschat, and F. Erlangen-Nürnberg, "Multi-core processors in the automotive domain: An autosar case study," *Proceedings Work-in-Progress Session*, 2010.

[21] J. Schneider, M. Bohn, and R. Rößger, "Migration of automotive real-time software to multicore systems: First steps towards an automated solution," in *the Proceedings of the Work-in-Progress Session of the 22nd Euromicro Conference on Real-Time Systems (ECRTS)*, 2010.

[22] T. Wei, Z. Qiu, C. Young, and D. Chang, "Development of heterogeneous multicore embedded platform for automotive applications," in *2011 International Conference on Circuits, System and Simulation (IPCSIT)*, vol. 7, 2011.

[23] 2carpros Web page, "How abs anti-lock brake systems work." http://www.2carpros.com/articles/how-abs-anti-lock-brakes-work.

[24] W. Xiang, P. C. Richardson, C. Zhao, and S. Mohammad, "Automobile brake-by-wire control system design and analysis," *Vehicular Technology, IEEE Transactions on*, vol. 57, no. 1, pp. 138–145, 2008.

[25] T. Nolte, H. Hansson, and L. L. Bello, "Automotive communications-past, current and future," in *Emerging Technologies and Factory Automation, 2005. ETFA 2005. 10th IEEE Conference on*, vol. 1, pp. 8–pp, IEEE, 2005.

[26] A. Davare, Q. Zhu, M. Di Natale, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli, "Period optimization for hard real-time distributed automotive systems," in *Proceedings of the 44th annual Design Automation Conference*, pp. 278–283, ACM, 2007.

[27] A. Rajeev, S. Mohalik, M. G. Dixit, D. B. Chokshi, and S. Ramesh, "Schedulability and end-to-end latency in distributed ecu networks: formal modeling and precise estimation," in *Proceedings of the tenth ACM international conference on Embedded software*, pp. 129–138, ACM, 2010.

[28] D. Goswami, R. Schneider, and S. Chakraborty, "Co-design of cyber-physical systems via controllers with flexible delay constraints," in *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, pp. 225–230, IEEE Press, 2011.

[29] P. Sinha, "Architectural design and reliability analysis of a fail-operational brake-by-wire system from iso 26262 perspectives," *Reliability Engineering & System Safety*, vol. 96, no. 10, pp. 1349–1359, 2011.

[30] A. Suleman, "Q & a: Do multicores save energy? not really." http://www.futurechips.org/chipdesignforall/amulticoresaveenergy.html, 2011.

[31] B. Kreeley and S. Coulton, "Increasing the lifespan and reliability of electrical components." http://www.ospmag.com/files/pdf/whitepaper/Kooltronics_WP.pdf.

[32] A. Grama, *Introduction to parallel computing.* Addison Wesley, 2003.

[33] P. Pacheco, *An introduction to parallel programming.* Morgan Kaufmann, 2011.

[34] G. R. Andrews, "Foundations of multithreaded, parallel, and distributed programming. 2000," *Wesley, University of Arizona, USA*.

[35] T. G. Mattson, B. A. Sanders, and B. L. Massingill, *Patterns for parallel programming.* Addison-Wesley Professional, 2004.

[36] W. Barry, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, 2/E.* Pearson Education India, 2006.

[37] O. Sinnen, L. A. Sousa, and F. E. Sandnes, "Toward a realistic task scheduling model," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 17, no. 3, pp. 263–275, 2006.

[38] A. Olteanu and A. Marin, "Generation and evaluation of scheduling dags: How to provide similar evaluation conditions," *Computer Science Master Research*, vol. 1, no. 1, pp. 57–66, 2011.

[39] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua, "Automatic program parallelization," *Proceedings of the IEEE*, vol. 81, no. 2, pp. 211–243, 1993.

[40] Wikipedia, "Openmp." http://en.wikipedia.org/wiki/OpenMP.

[41] Wikipedia, "Openhmpp." http://en.wikipedia.org/wiki/OpenHMPP.

[42] C. T. M.-C. P. Company, "Caps compilers." http://www.caps-entreprise.com/products/caps-compilers/.

[43] C. C. F. Bodin, "Programming heterogeneous many-cores using directives." http://cs.ioc.ee/etaps12/invited/bodin-slides.pdf.

[44] J. D. Ullman, "< i> np</i>-complete scheduling problems," *Journal of Computer and System sciences*, vol. 10, no. 3, pp. 384–393, 1975.

[45] M. T. C. S. JIS, "Computers and intractability a guide to the theory of np-completeness," 1979.

[46] O. Sinnen, *Task scheduling for parallel systems*, vol. 60. Wiley-Interscience, 2007.

[47] P. Chrétienne, "Task scheduling with interprocessor communication delays," *European Journal of Operational Research*, vol. 57, no. 3, pp. 348–354, 1992.

[48] I. Ahmad, Y.-K. Kwok, and M.-Y. Wu, "Analysis, evaluation, and comparison of algorithms for scheduling task graphs on parallel processors," in *Parallel Architectures, Algorithms, and Networks, 1996. Proceedings. Second International Symposium on*, pp. 207–213, IEEE, 1996.

[49] Y.-K. Kwok and I. Ahmad, "Benchmarking and comparison of the task graph scheduling algorithms," *Journal of Parallel and Distributed Computing*, vol. 59, no. 3, pp. 381–422, 1999.

[50] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys (CSUR)*, vol. 31, no. 4, pp. 406–471, 1999.

[51] N. Arora, "Analysis and performance comparison of algorithms for scheduling directed task graphs to parallel processors," *ANALYSIS*, vol. 4, no. 2, 2012.

[52] T. L. Adam, K. M. Chandy, and J. Dickson, "A comparison of list schedules for parallel processing systems," *Communications of the ACM*, vol. 17, no. 12, pp. 685–690, 1974.

[53] M.-Y. Wu and D. D. Gajski, "Hypertool: A programming aid for message-passing systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 1, no. 3, pp. 330–343, 1990.

[54] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee, "Scheduling precedence graphs in systems with interprocessor communication times," *SIAM Journal on Computing*, vol. 18, no. 2, pp. 244–257, 1989.

[55] G. C. Sih and E. A. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 4, no. 2, pp. 175–187, 1993.

[56] A. Olteanu, F. Pop, C. Dobre, and V. Cristea, "A dynamic rescheduling algorithm for resource management in large scale dependable distributed systems," *Computers & Mathematics with Applications*, vol. 63, no. 9, pp. 1409 – 1423, 2012.

[57] V. Sarkar, *Partitioning and scheduling parallel programs for multiprocessors.* MIT press, 1989.

[58] S. Kim and J. Browne, "A general approach to mapping of parallel computation upon multiprocessor architectures," in *International Conference on Parallel Processing*, vol. 3, p. 8, 1988.

[59] T. Yang and A. Gerasoulis, "Dsc: Scheduling parallel tasks on an unbounded number of processors," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 5, no. 9, pp. 951–967, 1994.

[60] Y.-K. Kwok and I. Ahmad, "A static scheduling algorithm using dynamic critical path for assigning parallel algorithms onto multiprocessors," in *Parallel Processing, 1994. ICPP 1994. International Conference on*, vol. 2, pp. 155–159, IEEE, 1994.

[61] H. El-Rewini and T. G. Lewis, "Scheduling parallel program tasks onto arbitrary target machines," *Journal of parallel and Distributed Computing*, vol. 9, no. 2, pp. 138–153, 1990.

[62] N. Mehdiratta and K. Ghose, "A bottom-up approach to task scheduling on distributed memory multiprocessors," in *Parallel Processing, 1994. ICPP 1994. International Conference on*, vol. 2, pp. 151–154, IEEE, 1994.

[63] K. Ghose and N. Mehdiratta, "A universal approach for task scheduling for distributed memory multiprocessors," in *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, pp. 577–584, IEEE, 1994.

[64] N. Mehdiratta and K. Ghose, "Scheduling task graphs onto distributed memory multiprocessors under realistic constraints," *PARLE'94 Parallel Architectures and Languages Europe*, pp. 589–600, 1994.

[65] O. Sinnen, A. To, and M. Kaur, "Contention-aware scheduling with task duplication," *Journal of Parallel and Distributed Computing*, vol. 71, no. 1, pp. 77–86, 2011.

[66] Y.-K. Kwok and I. Ahmad, "Bubble scheduling: A quasi dynamic algorithm for static allocation of tasks to parallel architectures," in *Parallel and Distributed Processing, 1995. Proceedings. Seventh IEEE Symposium on*, pp. 36–43, IEEE, 1995.

[67] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962.

[68] A. Inc., "Epiphany architecture reference (g3)." http://www.adapteva.com/wp-content/uploads/2012/12/epiphany_arch_reference_3.12.12.18.pdf.