SCHEDULING APPLICATIONS ON HETEROGENEOUS PLATFORMS

Quentin Pesquet

Master of Science Thesis
Supervisor: Dr. Sébastien Lafond
Embedded Systems Laboratory
Faculty Of Science And Engineering
Åbo Akademi University
August 2015
ABSTRACT

This thesis proposes to review different methods available to schedule applications on heterogeneous platforms. These platforms are often composed of a host processor and one or more remote processors, which have a specific role such as a digital signal processor. This thesis details the first step of the scheduling process with the representation of the application to be scheduled. Usually represented under the form of a DAG (Directed Acyclic Graph), the tasks which compose the application are mapped and ordered on the available processing units. Finding an efficient schedule helps to meet real-time constraints. Reducing the makespan of a schedule results in a faster execution of the application as well as a lower power consumption. Different scheduling strategies are detailed in order to understand how their algorithms work. In this thesis, we also used a specific platform called the OMAP4460 Pandaboard ES. It is an heterogeneous platform composed of different processing units such as a DSP (Digital Signal Processor) and a dual Cortex A9. We scheduled a simple application onto the Pandaboard to understand its mechanism and how the communications are made between the different processing units.

Keywords: schedule, heterogeneous, algorithm, constraints, makespan, DSP, processing unit, DAG.
# CONTENTS

Abstract i  
Contents ii  
List of Figures iv  
Glossary vi  

1 Introduction 1  
1.1 First section of the introduction . . . . . . . . . . . . . . . . 1  
1.2 Thesis structure . . . . . . . . . . . . . . . . . . . . . . . . . . . 2  

2 Application Scheduling 3  
2.1 Data flow model . . . . . . . . . . . . . . . . . . . . . . . . . . 3  
2.1.1 Directed Acyclic Graph . . . . . . . . . . . . . . . . . . . . 4  
2.1.2 Example of a schedule solution from a DAG . . . . . . . . 6  
2.2 Scheduling strategies . . . . . . . . . . . . . . . . . . . . . . . . 10  
2.2.1 Static scheduling algorithms . . . . . . . . . . . . . . . . . 10  
2.2.2 Dynamic scheduling algorithms . . . . . . . . . . . . . . . 13  
2.3 Partial Expansion Graph: A Novel Approach . . . . . . . . . . . 18  

3 Off-line Optimization tools: Population-based algorithms 21  
3.1 Genetic Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . 21  
3.1.1 Problem Representation . . . . . . . . . . . . . . . . . . . 22  
3.1.2 Algorithm steps . . . . . . . . . . . . . . . . . . . . . . . . . 24  
3.2 Particle Swarm Optimization . . . . . . . . . . . . . . . . . . . . 29  
3.2.1 Problem Representation . . . . . . . . . . . . . . . . . . . . 29  
3.2.2 Algorithm steps . . . . . . . . . . . . . . . . . . . . . . . . . 30  

4 Implementation 37  
4.1 Platform . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37  
4.1.1 Pandaboard ES . . . . . . . . . . . . . . . . . . . . . . . . . 37  
4.1.2 OMAP4460 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40  
4.1.3 Dual Cortex-A9 . . . . . . . . . . . . . . . . . . . . . . . . . 40
## List of Figures

2.1 High level illustration of a model based design approach for signal processing systems (design flow) [1] .......................... 5  
2.2 Example of a DAG [2] .................................................. 7  
2.3 The tasks’ runtime on the four processing units [2] .................. 8  
2.4 Results of the calculated parameters from the DAG of the Figure 2.2 [2] .......................................................... 9  
2.5 Schedule solution from the DAG of the figure 2.3 [2] ............... 10  
2.6 Representation of schedulers based on the EDF strategy on a multi core processors inspired by [3][4] ........................................... 15  
2.7 Representation of the execution of a DSP task on a dual-core processors inspired by [3] .................................................. 16  
2.8 A multi-rate Synchronous Data Flow (SDF) inspired by [5] ......... 19  
2.9 A Partial Expansion of an SDF graph inspired by [5] ................. 20  
3.1 A chromosomal representation of a data flow graph [6] ............ 22  
3.2 Example of a directed acyclic graph inspired by the example made by Wang [7] ........................................................... 23  
3.3 Flowchart illustrating the GA steps inspired by [8] .................... 25  
3.4 A chromosomal representation of a data flow graph inspired by [7] .... 26  
3.5 Example of an overview of the process to define a fitness function for a low-power optimization [6] ................................. 27  
3.6 Mapping and ordering crossover operation inspired by [7] ........... 28  
3.7 Directed Acyclic Graph inspired by [9] ............................... 30  
3.8 Task Resource Assignment Graph inspired from [9] ................. 31  
3.9 Illustration of the velocity and the position update in a two dimensional problem space inspired from [10] .............................. 32  
3.10 Flowchart illustrating the PSO algorithm inspired by [8] ......... 34  
3.13 Representation of the lbest neighborhood of different sizes [12] .... 36  
4.1 Pandaboard ES architectural block diagram [13] ................. 38  
4.2 OMAP4460 block diagram [14] ................. 39  
4.3 Cortex-A9 subsystem top-level architecture [14] .................. 41  
4.4 DSP subsystem top-level architecture [14] .................. 42
4.5 DSP megamodule top-level architecture [14]. . . . . . . . 43
4.6 L3-interconnect overview [14]. . . . . . . . . . . . . . . . . . . 44
4.7 Spinlock module [13]. . . . . . . . . . . . . . . . . . . . . . . . . 45
4.8 MMU overview [13]. . . . . . . . . . . . . . . . . . . . . . . . . 46
4.9 Mailbox system integration [14]. . . . . . . . . . . . . . . . . . 47
4.10 Communication between DSP and ARM Cortex A-9 [14]. . . . 48
4.11 Simple representation of an application made of five tasks inspired by an example [9]. . . . . . . . . . . . . . . . . . . . . . . . . . . 49
4.12 Execution time of the tasks on the processors inspired by [9]. . . 50
4.13 Communication time between the tasks inspired by [9]. . . . . . 50
4.14 Schedule solution. . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
GLOSSARY

DAG
Directed Acyclic Graph. A directed graph with no directed cycles. It is composed of nodes and directed edges.

Makespan
Total length of a schedule.

GPU
Graphical Processing Unit. A processing unit used for rendering graphics.

MMU
Memory Management Unit, provides virtual memory.

GPP
General Purpose Processor. In this thesis, it corresponds to the host processor of the platform.

DSP
Digital Signal Processor. In this thesis, it corresponds to the remote processor of the platform. It permits to process complex tasks.

SDF
Synchronous Data Flow.

ARM
Family of instruction set architectures.

SoC
System on Chip. A chip containing all components in a computer system, thus if using a SoC very few external components are used for building a system capable of computing.
1 Introduction

In the recent years, there has been a growing demand in embedded systems. The emergence of complex applications such as digital signal processing applications has resulted in the need of more powerful devices to process these applications. To be able to run high-performance demanding applications while meeting constraints such as a low power consumption and a low processing latency, the application has to be scheduled accordingly. Mapping and scheduling tasks onto processing units is the critical and most important part during the development process of the application. Hence, a lot of research is done in the embedded systems field to find the best scheduling strategies that make full use of the application properties as well as of the platform architecture.

1.1 First section of the introduction

This thesis depicts the matter of scheduling applications onto heterogeneous multiprocessors systems by reviewing different scheduling methods available as well as an example of a heterogeneous platform. The chosen platform, which is the OMAP4460 Pandaboard ES, features many components integrated together on the single chip. The components of interest are the main processor and its remote processor, the Digital Signal Processor. These processing units are programmable and permit the designer to order and map different tasks on these processors. The tasks ordering and mapping, which are the two steps in the scheduling process, are the results of a scheduling algorithm chosen beforehand by the designer. To produce these algorithms, it is important to know how to represent the application and the architecture of the device on which the tasks are processed.
1.2 Thesis structure

The thesis is divided into three parts. The first part deals with the way of scheduling applications onto heterogeneous platforms detailing the different available methods of scheduling. The second part defines optimization tools that enable us to take care of different scheduling problems that can occur specifically when we use heterogeneous systems. The third part describes the Pandaboard OMAP4460, a well known heterogeneous platform, which permits to make use of programmable processors such as the DSP. The third part also explains how the DSP communicates with the other processing units.

Chapter 2 describes how an application can be analyzed and modeled to be implemented on an heterogeneous platform. Knowing how the application is partitioned is crucial to find a scheduling solution as it exists different type of solutions. Chapter 2 also depicts the different available scheduling strategies with their algorithm and how they can help meet the constraints when running DSP applications. Finally Chapter 2 describes a novel approach to scheduling applications brought recently by Zaki and Al [5].

Chapter 3 describes two different off-line tools available to optimize scheduling solutions. Both are population-based algorithms that are useful when the complexity of either the application or the targeted platform is high and makes it difficult to find optimal scheduling solutions.

Chapter 4 is the implementation part and describes the platform used to run applications. The platform is the Pandaboard OMAP4460, which is a heterogeneous platform composed of two dual cores processors as well as a Digital Signal Processor, which is the component of main importance in this thesis. This Chapter describes how the different processors communicate with each other by using different components the OMAP4460 offers as well as a framework developed by TI in order to be able to program the DSP, which is important when scheduling DSP applications. Finally, an application composed of simple tasks will be used as an example to show a scheduling solution working on the OMAP4460.
2 APPLICATION SCHEDULING

Executing an application on an embedded system requires a broad knowledge of the application itself and of the platform as well. The application, after being divided into different tasks, needs to be scheduled considering the different characteristics of each task. It will also depend on what kind of processor the tasks will be executed on.

In this Chapter, we will see how an application can be represented to help us schedule it. We will then show different kind of scheduling policies. Finally, we will explain a new interesting approach to schedule applications.

2.1 Data flow model

Applications in embedded systems are more and more demanding and have high computing requirements due to different constraints. One of the areas allowing to gain more efficiency and improve performances is the task scheduling. Scheduling an application consists firstly in collecting different data from the application and also from the target platform, where the application is being processed. After profiling the application, this data is organized by the designer in order, for example, to find the different dependencies between the tasks. This can be done by using a Directed Acyclic Graph (DAG). Then the different tasks are mapped onto the available processors to be finally ordered on each processing unit. Efficient scheduling solutions enable us to achieve high performance. Due to the increasing complexity of applications over the years, optimization engines can also be used in order to improve scheduling solutions. These methods are discussed and detailed in Chapter 3.

In order to achieve high performance, it is important to make use of the characteristics of the application and the target architecture. Generally, the performance measurements that have to be optimized are the minimum mean flow time, the makespan
and the iteration period of a schedule [15]:

1. The mean flow time is the sum of the finishing time of all tasks. Different algorithms can be used to calculate the mean flow time [16] [17].

2. The makespan is the total length of the schedule when all the tasks have finished processing. It is important to minimize the makespan as it reduces the execution time of the application and thus helps meet the real-time constraints.

3. The iteration period is the time required to process one cycle of the algorithm.

An effective schedule that permits, for example, to run an application while lowering the energy consumption is a good compromise of the three performance measurements mentioned above.

### 2.1.1 Directed Acyclic Graph

On a heterogeneous platform the main phase when scheduling an application is to map tasks in parallel onto the different available processors (GPP, DSP, ...) and order the tasks so that the real-time constraints are met. Scheduling tasks on a limited set of processors means dealing with each processing unit’s characteristics (its architecture, its processing speed) that can be subject to some constraints as well. A minimum schedule length results in a faster execution time and helps lower power consumption.

The application to be scheduled must be partitioned into sets of tasks (and subtasks if needed considering the complexity of the application). This profiling phase consists generally in generating a directed acyclic graph (DAG) from the application which will be scheduled. It takes into account the different tasks dependencies, their properties, their deadlines so that they will be accordingly executed on the right processor with the right order.

As an input to the scheduler, we define appropriate models to represent the application (Application Description), the platform (Architecture Description) and some objectives and constraints which are represented in the high-level diagram in Figure 2.1. It includes in detail the following:
1. An application description in which the application model (Block Processing DAG) includes a set $T$ of tasks $t$ (also called nodes), and edges $e$ representing the task dependencies. Nodes should be mapped accordingly to the available processors while meeting the dependency requirements and other constraints.

2. An architecture description in which the platform includes a set $P$ of processors.

The application description, the architecture and the different constraints can be represented in a directed acyclic graph for the definition of the schedule. A DAG can be defined by $G=(T,E,P,n,c)$, as seen in [2], where $T$ is the set of task nodes $t$. $E$ is the set of communication edges $e$ with $e_{ij}$ being the arc between the node $t_i$ and $t_j$. $t_i$ is called the parent node and $t_j$ is the child node. $P$ is the set of processors ($p_1, p_2, ..., p_n$).

After simulating the tasks and edges on different processors, we obtain the execution time and the communication cost defined by two functions:

1. $n=t(t \in T, p \in P)$ : it defines the execution time of task $t$ on processor $p$. It is also called the computation cost.
2. $c = c(t_i, t_j)$: it defines the communication cost between $t_i$ and $t_j$. With $c = 0$ if $t_i$ and $t_j$ are executed on the same processor.

The task dependency is conserved in the DAG. Indeed, a task $t_i$ is dependent on $t_j$ if it exists a path which starts at $t_i$ and ends at $t_j$. If there’s none, the tasks are parallel. The same analysis works for the edges.

The DAG gives a scheduling solution that is passed to the code generator with the help of system attributes (scenarios). It then eventually provides a working solution that leads to profiling tasks and tuning relevant parameters that will be iterated over and over again until final specifications are met or if we want to increase the goals and requirements.

2.1.2 Example of a schedule solution from a DAG

In Figure 2.2, a DAG is represented and composed of 11 nodes (tasks) with one entry node (task 1) and one exit node (task 11). There are also 15 communication edges that join tasks between them. It is assumed to be scheduled on a heterogeneous platform composed of 4 cores. Hence the tasks have different runtime on the different processors.

To the notations previously mentioned, a few new equations are used in order to calculate the relevant parameters:

- $PRED(t_i)$: set of tasks that precede $t_i$ and are joined directly to $t_i$.
- $SUCC(t_i)$: set of tasks that follow on from $t_i$. They receive data from $t_i$ and are directly joined to $t_i$.
- $EST(t_i/p_i)$: Earliest Start Time of the task $t_i$ executed on the processor $p_i$.
- $EFT(t_i/p_i)$: Earliest Finish Time of the task $t_i$ executed on the processor $p_i$.
- $FP_1(t_2) = p_1 \in P$: favorite processor for task $t_2$ is $p_1$. $FP_1$ being the best choice processor, $FP_2$ being the next best choice processor, etc.
- $Fpred(t_2) = t_1 \in PRED(t_2)$: favorite predecessor of task $t_2$ is $t_1$.
- $LEVEL(t_i)$: highest value of the whole computation time needed for any path from the node $t_i$ to the exit node.
$LAFT(t_i)$: latest allowable finish time of a task.
$LAST(t_i)$: latest allowable start time of a task.

The run time of the different tasks on the available processors are listed in Figure 2.3.

Bajaj and Al [2] created the algorithm called TANH (Task Duplication-Based scheduling Algorithm for Network of Heterogeneous Systems) which permits to have a scheduling solution from a Directed Acyclic Graph. The TANH has been created based on other algorithms made for homogeneous systems like the Task Duplication-based Scheduling Algorithm (TDS) [18]. To correctly compute the different parameters of the TANH, the graph has to be read downwards, except for the $LAFT$ and $LAST$ where the DAG is read upwards. The formulas that enable us to calculate these parameters can be found in [2]. The calculations can be divided into 5 steps. Step 1 to step 4 are done while reading the DAG in a top-bottom fashion while the 5th step is done by reading the graph from the bottom to the top:

1. Step 1: Calculation of the $EST$ of each task. Computing the earliest start times permits to know the earliest time when a given task has to be started. A task cannot start to be processed on a processor before its $EST$. 

![Figure 2.2: Example of a DAG [2].](image)
2. Step 2: Calculation of $FP$ for each task. The favorite processor is not chosen solely based on the execution time of a node on one processor. It is also chosen based on the communication cost between the node and its successor node.

3. Step 3: Calculation of the $EFT$ of each task. A task will be completed at its earliest finished time after being processed on its favorite processor during a specific amount of time called the execution time $n(t,p)$.

4. Step 4: Calculation of $F_{pred}$ for each task. The Favorite predecessor of a node $i$ is the node that precedes node $i$ with the highest value of $EFT$ calculated during step 3. Having the node with the highest Earliest Finished Time as the $F_{pred}$ ensures that all the predecessor tasks are completed.

5. Step 5: Calculation of $LAFT$ and $LAST$ for each task. $LAFT$ and $LAST$ permit to know when at the latest a task can respectively finish processing and start its process. These parameters are calculated by reading the DAG from the enter node to the exit node. The $LAFT$ of a node must be inferior or equal to the $LAST$ of its successor nodes. Two cases can occur: if a node and its successors are planned to be executed on the same processor, then there is no communication cost ($c=0$). If the nodes are on different processors, then there is a communication cost $c>0$ which needs to be subtracted from the $LAFT$ [2]. The method of
calculation of the LAFT and LAST is similar to the method for calculating EST and EFT except the graph is read in the opposite fashion (downwards opposed to upwards).

The results obtained after the making of these steps are listed in Figure 2.4.

Figure 2.4: Results of the calculated parameters from the DAG of the Figure 2.2 [2].

In order to produce a final schedule solution, tasks are clustered and ordered thanks by their LEVEL values (ascending order) to match the best available processor. The LEVEL is computed by calculating all the different computation costs possible for a node to the exit node. The different paths are tested, the highest value of the sum of the computation costs will be the LEVEL value. The algorithm used to generate the clusters can also be found in [2]. Four clusters are generated for four processors available. The clusters executed on Processor 1, 2, 3 and 4 are respectively 11;9;5;2;1, 8;4;1, 6;3;1 and 10;7;4;1. It eventually leads to one possible schedule with a minimal makespan of 25 time unit, from the DAG previously presented, shown in Figure 2.5. We can notice five arrows. One arrow indicates that two successive nodes of the DAG are processed on two different processors, which is meaning of a communication cost: between the task 2 (executed on P1) and task 6 (executed on P3), there is indeed a communication cost of 1 unit of time (as also seen on the DAG).
2.2 Scheduling strategies

There are basically two main scheduling strategies possible for real-time systems called static scheduling and dynamic scheduling policies. They both have their own characteristics and useful properties considering the type of application to be scheduled and the platform architecture. It is important to define the "right" method as it will determine when each task (or subtask) is executed and on which processing unit the tasks will be processed.

2.2.1 Static scheduling algorithms

The scheduling decisions of a static scheduler are made off-line. Hence it doesn’t require much CPU power at run-time. The tasks are checked before the process begins, which gives a pre-determined schedule. When the task is considered as ready, it is immediately executed. The static scheduling techniques require us to know precisely what is going to be scheduled before it is scheduled. Having an accurate estimation of the different tasks’ execution time is therefore very important.

Two examples of static DAG scheduling based on list-scheduling heuristics have been proposed by Topcuoglu and Al [19] with the Heterogeneous-Earliest-Finish-Time (HETF) algorithm and the Critical-Path-On-a-Processor (CPOP) algorithm. In these algorithms, the tasks are ordered with a double-rank system called upward rank and downward rank.
HEFT algorithm

The HEFT is a list heuristic that permits to schedule a set of tasks represented by a Directed Acyclic Graph (DAG) on heterogeneous platforms [19]. It is one of the most frequently used algorithm to reduce the makespan of a schedule [20].

This algorithm consists of three phases. Firstly, the tasks are prioritized and ordered according to their rank, also called upward rank. The upward rank noted rank\textsubscript{u}(t\textsubscript{i}) of a node t\textsubscript{i} is equivalent to the length of the critical path from the said task t\textsubscript{i} to the exit node of the DAG. The graph is then read in an upward fashion in order to compute each rank\textsubscript{u}. Assigning a rank to a task consists in assigning a priority to the task. After calculating the different rank\textsubscript{u} of each node, the tasks are ordered by decreasing order of rank\textsubscript{u} value. The task with the highest priority is selected waiting for the third phase which consists in selecting the processor. The tasks that have the same rank (same priority) are randomly picked between them. The processor that minimizes the most the execution time of the task is selected. The selected processor permits the task to be executed at the earliest time possible. The simplified algorithm is shown in Algorithm 2.1. The detailed calculation of the parameters can be found in [19].

Algorithm 2.1 HEFT Algorithm [19]

read graph upward from exit node;
compute rank\textsubscript{u};
order the nodes by decreasing rank\textsubscript{u} values order;
while unscheduled nodes are present do
    remove 1st task t\textsubscript{i};
    p\textsubscript{i} ← t\textsubscript{i} (assign task t\textsubscript{i} to processor p\textsubscript{i} with minimization of EFT(t\textsubscript{i}));
end while

CPOP algorithm

The CPOP (Critical-Path-On-a-Processor) is yet another list-heuristic, which schedules tasks from a DAG on heterogeneous systems.
Algorithm 2.2 CPOP Algorithm [19]

read graph upward from exit node;
compute \( rank_u \);
read graph downward from enter node;
compute \( rank_d \);
\[ |CP| = rank_u(t_1) \] where CP = critical path and \( t_1 \) = enter node;
\textbf{for} (i=1, i<=n, i++) \textbf{do}
\hspace{1em} \textbf{if} (rank_d(t_i) + rank_u(t_i)) == |CP| \textbf{then}
\hspace{2em} \( t_i \) is critical path node CPN;
\hspace{1em} \textbf{end if}
\textbf{end for}
select critical-path-processor with minimization of CPN;
initialize priority queue with the entry nodes;
\textbf{while} unscheduled nodes are present in priority queue \textbf{do}
\hspace{1em} select node with highest priority in the queue with maximization of \( rank_d(t_i) + rank_u(t_i) \);
\hspace{1em} \textbf{if} \( t_i \) is CPN \textbf{then}
\hspace{2em} schedule \( t_i \) to critical-path-processor;
\hspace{1em} \textbf{else}
\hspace{2em} \( p_i \leftarrow t_i \) (assign task \( t_i \) to processor \( p_i \) with minimization of EFT(\( t_i \)))
\hspace{1em} \textbf{end if}
\hspace{1em} \textbf{if} \( n_i \) is ready-node \textbf{then}
\hspace{2em} update priority queue with successor(s) of \( t_i \);
\hspace{1em} \textbf{end if}
\textbf{end while}
This algorithm follows the same logic as the HEFT. It computes a rank value to a task in order to prioritize the tasks. However, in addition to the upward rank, the CPOP also uses another rank called downward rank. The downward rank noted $rank_d(t_i)$ of a node $t_i$ is equivalent to the length of the critical path from the entry node to the said task $t_i$. The DAG has to be read downward to compute the downward rank of each task.

The algorithm assigns to each task its priority equal to $rank_u(t_i) + rank_d(t_i)$. The tasks are grouped into two categories: critical-path-node and non-critical-path-node. A critical-path-node is executed on the processor that will minimize its critical path the most among all the processors available. The selected processor is called critical-path-processor. A non-critical-path-node is executed on the processor that will minimize its execution time the most among all the processors available (task with the earliest finish time possible). The complete algorithm is represented in Algorithm 2.2 and the detailed computation parameters can be found in [19].

As we can see, the HEFT, the CPOP and also the TANH (detailed in section 2.1.2) are three static scheduling techniques that only work under certain conditions, as stated before: it is crucial to know the different execution times of each task, the different processors and their parameters. These conditions are restrictive and the applications don’t always fulfill these requirements. Some problems can occur such as the lack of information on either the tasks or the architecture of the platform. In such cases, using dynamic scheduling algorithms can be a good solution, as it doesn’t require a complete knowledge of all the parameters.

### 2.2.2 Dynamic scheduling algorithms

In a dynamic scheduling algorithm, the decisions are made without any jobs’ predictable execution time. It is then more flexible than the static policy, as it will execute tasks without any predictable knowledge about them. Thus, if a task’s behavior is unstable or unpredictable, it can be scheduled accordingly by taking into account all the current active jobs and their priorities [4]. The tasks are dynamically chosen based on their priorities. Furthermore, we can notice two kinds of dynamic scheduling algorithm, as the priority of each task can be either statically or dynamically assigned.
Static priority algorithms are used under different constraints, especially on the execution of the tasks, as stated by Balarin and Al [21]. Indeed, each task generates jobs of the same priority. The tasks must be executed periodically. Tasks’ priority which is computed at run-time is fixed and the execution of a task is independent from another. In addition to the periodicity of each task, the tasks also must have a constant runtime. An example of static priority algorithm that manages to be working under these conditions is the Rate Monotonic Scheduling (RMS), which was first introduced by Liu and Layland [22]. It is known to be an optimal algorithm for uniprocessor systems. The Rate Monotonic Scheduling consists in assigning each task a rate-monotonic priority: the task whose period is the shortest will be assigned the highest priority [21]. However, this algorithm can be adapted to multiprocessor platforms using the same logic [23]. The migration of the RMS on heterogeneous systems has been done by Baruah and Goossens [24].

The dynamic priority policy is a result of an extension of the static one also presented by Liu and Layland [22]. The main difference with the dynamic priority policy compared to the static one resides in the fact that a same task can generate jobs of different priorities. Earliest-Deadline-First (EDF) algorithms are based on this strategy [21]. In the EDF algorithm, a global scheduler is used to divide the tasks into subtasks (stream of subjobs) and map them to the available processing units. A representation of such scheduler is shown in Figure 2.6 in which a global scheduler maps tasks onto three processing units: two General Purpose Processors (GPP) and one Digital Signal Processor (DSP) that compose the heterogeneous system. Before the tasks are executed on the right processor, they are ordered by prority with the use of local schedulers.

An algorithm based on the EDF strategy has been proposed by Kuo and Hai [3] called the Online EDF-based Scheduling Algorithm (OESA). In this algorithm, the task (divided into subtasks) with the earliest deadlines (in other words, with absolute deadline nearest to the current time) is assigned the highest priority. For each processor, a local scheduler takes care of ordering the subtasks (also called subjobs) on the processor assigned to the said local scheduler. The subtasks are assigned deadlines that will permit to know which subtask to map in which order to which processing unit available. The tasks to be scheduled can be divided into a set of subtasks with different
Figure 2.6: Representation of schedulers based on the EDF strategy on a multicore processors inspired by [3][4]

execution time, deadlines and can be executed on different processor. For example, a DSP (Digital Signal Processing) task can be executed on different processors. Divided in three subjobs, the DSP task can begin its execution on a GPP and then be assigned to complete its process on a DSP. It depends on its different subjobs and their deadlines.

An example of the execution of a DSP task is given in Figure 2.7, which shows the simplified execution of a decoding task; it is split up in three subjobs: GPP reads the video data, then the DSP decodes the data and finally the GPP shows the changed data. The three subjobs are represented in Figure 2.7 as $t_{i,j,pre}$, $t_{i,j,dsp}$ and $t_{i,j,post}$, which compose the task $t_{i,j}$ when the task starts its execution at the time $r_{i,j}$. Their execution time are respectively noted $c_{i,pre}$, $c_{i,dsp}$ and $c_{i,post}$. Each of the subjobs, which are of equal period (as they compose the same task), are assigned an absolute deadline that will
permit to prioritize the subjobs. A subjob can only be executed when its predecessor’s
deadline is past. The calculations of the absolute deadlines, which are based on the
period and execution time of the subtasks can be found in [3]. $t_{i,j,\text{pre}}$ and $t_{i,j,\text{post}}$ will be
executed on the GPP. On the other hand, $t_{i,j,\text{dsp}}$ will be executed on the DSP. Considering
the number of subtasks, the subtasks are scheduled on their corresponding processor
queue (GPPqueue for the GPP and DSPqueue for the DSP). The subtasks are ordered
in a queue based on their deadlines waiting to be executed. The final schedule is said
to be feasible if all the subtasks of the application meet their deadlines [25].

![Figure 2.7: Representation of the execution of a DSP task on a dual-core pro-
cessors inspired by [3].](image)

The Algorithm 2.3, which represents an EDF strategy inspired by the work of Kuo
and Hai [3], shows the different steps of a task’s scheduling. This strategy is basically
composed of two main parts:

1. Step 1: The scheduler checks the status of the task. There are three different
   status assigned to a task that permit to differentiate the task: pre_status and
   post_status indicate that the task has to be executed on the GPP. dsp_status in-
   dicates that the task has be to executed on the DSP. When the task is scheduled
   on its appropriate processor’s queue (GPPqueue or DSPqueue), the scheduler
   assigns the task another status following a specific fashion, which is pre_status
   then dsp_status and finally post_status periodically.

2. Step 2: The scheduler checks the priority of the jobs by comparing the differ-
   ent (currently running or arriving) subjobs’ deadlines with the current time. The
Algorithm 2.3 Earliest Deadline First Algorithm [3]

if subjob of job $t_{i,j}$ finishes then
    if $t_{i,j} \_status == pre\_status$ then
        if $c_{i,dsp} \neq 0$ then
            $t_{i,j} \_status = dsp\_status$;
            insert nowGPP into DSPqueue;
        end if
        nowGPP = Null;
    else
        if $t_{i,j} \_status == dsp\_status$ then
            if $c_{i,post} \neq 0$ then
                $t_{i,j} \_status = post\_status$;
                insert nowDSP into GPPqueue;
            end if
            nowDSP = Null;
        else
            if $t_{i,j} \_status == post\_status$ then
                nowGPP = Null;
            end if
        end if
    end if
end if

if job $t_{i,j}$ arrives then
    insert job $t_{i,j}$ into GPPqueue;
end if

if 1st element’s deadline of GPPqueue is closer to the currentime than nowGPP’s deadline then
    insert nowGPP into DSPqueue;
    nowGPP = 1st element of GPPqueue;
end if

if nowDSP == Null then
    nowDSP = 1st element of DSPqueue;
end if

Execute operations of nowGPP and nowDSP;
scheduler also checks the availability of the DSP processor. The subjobs are then assigned to the appropriate processing unit in the appropriate order.

2.3 Partial Expansion Graph: A Novel Approach

An another method of scheduling applications onto heterogeneous systems composed of Digital Signal Processors (DSP) consists in representing the application to be scheduled differently from a Directed Acyclic Graph (DAG). Instead of a DAG, the application is represented by a Partial Expansion Graph (PEG). This novel approach proposed by Zaki and Al [5] is motivated by the increasing complexity of nowadays DSP applications and by the complexity of the architecture of heterogeneous multiprocessors platforms. Indeed, using a PEG will help the designer utilize the different characteristics that the representation of an application can offer such as:

1. task parallelism.
2. data parallelism.
3. pipeline parallelism.

As its name suggests, the PEG is an expansion of the synchronous dataflow graph (SDF). A representation of a SDF can be seen in Figure 2.8. A Synchronous Data Flow graph is generally defined by actors (also called nodes) that represent the application’s tasks. These actors fire tokens, which represent the data communicating between the nodes. Each node has an input and an output. When an actor is triggered, the number of tokens this actor produces is equal to the number of tokens consumed by the same actor. The example seen in Figure 2.8 shows an SDF composed of multi-rate arcs and three actors with for example q(A)=1, q(B)=N and q(C)=N². q(X)=Y being the equation that gives the rate Y to the actor X. Such multi-rate SDF could lead to exponential graphs to be scheduled, thus leading to time consuming solutions [26][5]. It is in these conditions that Zaki and Al [5] proposed to apply a Partial Expansion Graph transformation to the SDF.
A PEG is defined as an undirected graph composed of three main elements:

1. One or more sets of *Instances*. One set of *instances* corresponds to one *actor* in the SDF.

2. A buffer manager which composes with the *instances* the vertex set of the PEG.

3. A set of undirected edges which permits to connect each instance to the buffer manager.

Figure 2.9 shows an example of a PEG after it was expanded from the SDF in Figure 2.8. This transformation gives a PEG composed of three sets of instances (noted \( N_A = 1 \), \( N_B = 4 \) and \( N_C = 2 \)) for a total of seven instances, seven undirected edges and the buffer manager.

The buffer manager handles communications between each set of instances (each set corresponds to one actor in the SDF). The buffer manager permits to pass on information between the different instances mapped onto different processors [5]. The buffer manager sends and receives back messages in order to activate the instances (just like the actor firings in the SDF) using a system of four different states as explained in Zaki and Al’s work [5]. Those messages contain the information about the number of tokens each actor needs to produce and consume in the corresponding SDF. In other words, the buffer manager takes care of the scheduling as it decides when to activate...
the instances on which processor. The buffer manager checks the availability of the current state of the instances as well as the availability of the processor the instances are mapped onto.

The two types of scheduling strategies presented earlier in this thesis can be used. Static and dynamic algorithms can be implemented within the buffer manager to assign the instances to the processors [5]. The way the instances are ordered and prioritized will then depend on which scheduling technique has been chosen and implemented.
3 Off-line Optimization Tools: Population-based Algorithms

Although the scheduling methods mentioned in the previous Chapter are optimal on uniprocessor architectures, it becomes more complicated to correctly map and order tasks on multiprocessors systems, especially on heterogeneous platforms depending on whether a static or a dynamic algorithm strategy is used. As heterogeneous architecture are composed of different processing units and each processor has its very own characteristics and properties, it is known to be a NP-complete problem to schedule tasks on these platforms [28]. Furthermore, the nowadays applications demand more and more high performance from the device, better reliability, robustness, flexibility and low-power consumption [27]. However, additional methods used alongside the scheduling algorithms mentioned previously permit to find near optimal solutions while meeting multi-objective goals. These methods are called optimization engines and are more and more used nowadays due to the high complexity of the applications and target devices.

In this Chapter, we will explain two optimized algorithm used on heterogeneous platforms. Firstly, the Genetic Algorithm will be detailed. And finally, we will explain the Particle Swarm Optimization.

3.1 Genetic Algorithm

With the increasing complexity of embedded systems, other solutions can be used in order to improve the performance of an application by reducing its execution time and thus by minimizing the makespan of the schedule. It leads to a faster and low-power execution. Off-line optimization engines permits to find schedule solutions at compile-time in heterogeneous platforms as it is known that creating a schedule is NP-complete.
The Genetic Algorithm (GA) is capable of solving complex optimization problems as well as reducing schedule’s makespan and increasing the schedule reliability by copying the biological evolution process. The goal is to encode all the essential information for code generation. Multiple codings for GA can be applied depending on the type of application or the problem to be solved. For example, the individuals (also called chromosomes), which represents the different scheduling solutions can be represented by dataflow graphs, DAG, vectors of integers, trees, etc... Depending on each coding used to represent the chromosomes, its characteristics will vary as well.

### 3.1.1 Problem Representation

![Data Flow Graph](image)

Figure 3.1: A chromosomal representation of a data flow graph [6].

Figure 3.1 shows a chromosomal representation of a data flow graph. The chromosome is subdivided into genes (representation of an operation like an addition, a multiplier, etc…) used to encode the variables of the optimization problem. The values of a gene (alleles) express the information needed (used registers, processor instructions, execution cycle, speed, etc…) for a code generation. A good combination
of alleles results in a good solution to optimization problem. The structure of this chromosomal representation, which is of variable length to provide flexibility, is made here of three execution elements (ADD for adder, MUL for multiplier and DEL for delay), primary inputs stored in the functional element INPUT and four connection nets (n1, n2, n3, n4). Finally, the primary outputs are saved independently in a lookup table Y since they are only required to decode the chromosome before the genetic process finishes. They are not used during the evaluation step whereas the primary inputs are (for fitness calculation). The execution elements are the representation of the genes of the individual as mentioned above. There are different pieces of information encoded in each gene called alleles. For example [6], the alleles can correspond to the characteristics of each gene such as its execution speed (of a task on a processor), its function, its dependence with other genes, its input and output (input and output being the connectivity data of the gene).

Another way to represent a chromosome is to transform a DAG (representing an application to be executed, as seen in Chapter 2) into its chromosomal form, which is more intuitive than the previous example. A simple example of a DAG composed of five tasks and five edges is shown in Figure 3.2. This example will be used as the application DAG in order to explain the different steps of the genetic algorithm.

![Figure 3.2: Example of a directed acyclic graph inspired by the example made by Wang [7].]
3.1.2 Algorithm steps

The Genetic Algorithm (see Figure 3.3) is made of six main steps, which are common for every chromosomal representation:

1. Initialization of all individuals, which constitute the population (graph nodes to be scheduled in the example seen in Figure 3.2).
2. Evaluation of these individuals.
3. Selection of the individuals that inherit their genes to the next generation of individuals.
4. Crossover: two individuals are chosen and we swap their genes with each other.
5. Mutation: Change alleles to create new genes material.
6. Re-evaluation of the individuals.

Iteration of this process until finished and defined conditions are met.

A Genetic Algorithm needs a varied initial population composed of diverse individuals in order to find the solution space from different locations. This initialization step permits to encode the application DAG into chromosomes, which represent possible scheduling solutions. The solutions are randomly generated so that tasks are assigned randomly to the processors available on the platform targeted. The tasks must keep their order. In other words, they have to obey to their precedence constraints they had in the corresponding DAG. It is called "performing a topological sort" of the DAG [28]. The different chromosomes generated represent the initial population. Two examples of chromosomes generated from the DAG in Figure 3.2 can be seen in Figure 3.4. Each chromosome is composed of a mapping part and an ordering part. The mapping part generated randomly gives the information about which processor is assigned to which task. The ordering part informs about the task ordering (the chromosome is read from top to bottom).

Once the population is composed of a specific amount of individuals defined by the designer, each and everyone of the individuals are evaluated by applying a fitness
Figure 3.3: Flowchart illustrating the GA steps inspired by [8].
Figure 3.4: A chromosomal representation of a data flow graph inspired by [7].

function. This function depends on what kind of optimization we want to operate and the objectives of the schedule. For example minimizing the makespan of the schedule [28]. It is also possible to get a low-energy optimization by estimating its supply voltage and its capacitance (see Figure 3.5). More information on the calculation of these estimations is detailed in [6].

After applying the fitness function, the low-quality solutions that don’t meet the requirements are eliminated. The fittest chromosomes can be duplicated [7] in order to keep a population as large as the first generated population but of better quality.

The crossover is a reproduction technique that generates two offsprings, which are a combination of the characteristics of the two parents. The crossover is randomly done by applying a one-point crossover on each parent resulting in dividing each of them into two parts called the head and the tail. It is called the ordering crossover. The tasks of the tail or the head are swapped leading to a re-order of the tasks. Another
kind of crossover can be done by exchanging a part of the mapping part of the parents by applying randomly another one-point crossover. Figure 3.6 shows how an ordering crossover and mapping crossover are realized based on the chromosomes generated in Figure 3.4. The two new solutions, called the two children, guarantee to give valid and feasible schedules [7] as they inherited characteristics from their parents.

The next step is the mutation of the population. The mutations are comparable to the crossovers as there are two kinds of mutations: the ordering mutation and the mapping mutation. The ordering mutation consists in randomly choosing a task and its definite "valid range" (above and under the task to be mutated) [28] in which the said task can be transported. The range is said valid because within this range, the new position of the mutated task must not compromise the topological order (data dependency constraints [7]) of the corresponding DAG. The mapping mutation consists in applying the same method that the ordering mutation except there are not data dependency problems. For a same chromosome, the processor assignments of the randomly chosen tasks are swapped with each other. A mapping mutation can be seen as a local crossover involving only one chromosome. A more detailed representation of the mutation

Figure 3.5: Example of an overview of the process to define a fitness function for a low-power optimization [6].
Once the mutation is done, the process then moves on to another evaluation step. Every chromosome is evaluated by checking their ordering and mapping components. After testing and checking the execution time of the different tasks on the processors they are assigned (execution of the schedule solution), the fitness can then be measured and evaluated. There can be as many fitness functions to be checked as there are scheduling objectives set by the designer. If the solution which is left meets the different requirements (fitness functions, number of iterations), the simulation terminates and the best solution is chosen as the final schedule. Otherwise, the algorithm will reiterate until the conditions set by the designer are fulfilled. In other words, the best parents are selected and will be part of the new population generated alongside with new randomly created chromosomes [28].
3.2 Particule Swarm Optimization

Particle swarm optimization is yet another population based algorithm similar in some ways to the genetic algorithm mentioned above, but with no direct combination between the individuals. Mapping and ordering tasks onto heterogeneous systems is an NP-complete problem [29] and the PSO has been used in order to find near optimal solutions [30]. It is inspired by the behavior of the flocks of birds and their collective intelligence. It is a fairly recent heuristic research method as it was invented in 1995 by Kennedy and Eberhart [31].

3.2.1 Problem Representation

Like most of the application representation when it comes to scheduling, a Directed Acyclic Graph (DAG) is used in order to represent the different tasks as nodes, and edges as tasks dependencies. A DAG can be slightly transformed into his Task-resource assignment graph form (T-RAG) as explained by Chen and Wang [9]. A T-RAG is basically a DAG with a few more features. Indeed, the nodes are directly assigned to the processors with the execution time and the communication cost. It gives then an already working schedule made "randomly". It is said "randomly" because the processor assignment is made randomly although the topological order of the tasks from the DAG is respected.

Let the Figure 3.7 be the example DAG and Figure 3.8 be one possible T-RAG transformation. Figure 3.8 shows the tasks $T_i$, the processors $P_j$ assigned to the tasks, the communication cost $C(T_i; T_{i+1})$ on each edge and the execution time $E(T_i; P_j)$ for each node. As it is made off-line, the execution time and communication cost are supposed to be known in advance. The operations to calculate these parameters can be found in [9]. It is from the T-RAG that the problem representation can be encoded for the PSO algorithm. Indeed, the execution time and communication cost are transformed into two position vectors. Both of them combined will form only one position vector. The different computation techniques to vectorize the communication cost and execution time are present in [9].
Figure 3.7: Directed Acyclic Graph inspired by [9].

The generated vector position, which will represent a particle (explained in the next section) in the PSO, is one possible scheduling solution to the application. It is also important to have multiple "random" DAG from the application in order to generate multiple particles that will be part of the algorithm.

### 3.2.2 Algorithm steps

The Particle Swarm Optimization (PSO) is initialized with a population of randomly generated candidate solutions called particles. Each particle has specific coordinates assigned in the $N$-dimensional problem space. The position of a particle is noted $X_i(k) = \{x_{i;1}(k); x_{i;2}(k); \ldots; x_{i;N}(k)\}$ with $k$ being the current iteration value. The dimension of the problem space $N$ is defined by the number of tasks to be scheduled present in the application. The particles move through the problem space at each iteration. It means that each time it iterates, each particle is assigned a new position composed of $N$ elements ($N$ tasks = $N$-dimension) represented by $X_i(k) = X_i(k) + V_i(k+1)$ with $V_i(k)$ representing the velocity parameter. The velocity is the rate of the position change between the current position and the next one [10] and is represented by $V_i(k) = \{v_{i;1}(k); v_{i;2}(k); \ldots; v_{i;N}(k)\}$ where $i$ represents the $i$-th particle and $i = \{1; 2; \ldots; N\}$ is the
size of the swarm at iteration $k$. The particles keep track of both their own personal best position, called local best represented as $p_{i,\text{best}}$ and the best position ever found among all the particles in the swarm, called global best and represented as $g_{\text{best}}$. For each particle, its fitness function is evaluated: The $p_{i,\text{best}}$ of every particle is both compared to its previous value and to the $g_{\text{best}}$ of the swarm. After finding the two best values, the particles move in the problem space by updating, if needed, both $p_{i,\text{best}}$ and $g_{\text{best}}$ and consequently their velocities and positions. Here is the equation of the velocity for each individual $i$ at iteration $k$:

$$V_i(k+1) = \omega \cdot V_i(k) + c_1 \cdot \text{rand}_1 \cdot (p_{i,\text{best}}(k) - X_i(k)) + c_2 \cdot \text{rand}_2 \cdot (g_{\text{best}}(k) - X_i(k))$$

where $\text{rand}_1$ and $\text{rand}_2$ are two randomly generated numbers $\in [0,1]$. $\omega$, $c_1$ and $c_2$ are the weights set respectively to 1, 2 and 2 in the original PSO algorithm [31]. Different values can be given to these weights between 0 and 2 depending on the application. The motion in the search space is represented in the Figure 3.9 in which the three weights have been given random values.
Figure 3.9: Illustration of the velocity and the position update in a two dimensional problem space inspired from [10].
In the equation of the velocity mentioned above, we can differentiate three components:

1. The *inertia* of the previous velocity where the weight $\omega$ controls the balance between exploration and exploitation of the problem space [32]. This constant permits to keep the particle’s direction of movement as it is proportional to the previous velocity.

2. The *cognitive* component represents the attraction of the particle towards its *local best* scaled by the product of the constant $c_1$ and the random number $rand_1$. This is the particle memory influence.

3. The *social* component represents the attraction of the particle towards the *global best* scaled by the product of the constant $c_2$ and the random number $rand_2$. This is the swarm influence.

The algorithm then iterates until the number of iterations set by the designer is reached or until the conditions on the fitness function are met, as presented in Figure 3.10, which shows a flow diagram that represents the process of the PSO. The global best solution will be the best individual thus the best schedule for our application.

Some problems can occur during the process of the PSO. Indeed, it can be trapped into a local minima and get stuck with no possible further improvements. Preventing the genesis of premature convergence is crucial especially because of real-time constraints. In order to avoid such situations, the PSO can process by gathering two or more particles to form *neighborhoods* of different size and different topology [11]. Instead of having one particle (the one that has the best position) that communicates with all the particles of the swarm, each particle can only share its information with the other particles of its *neighborhood*. The way these neighborhoods are used will then greatly influence the behavior of the algorithm.

The basic PSO can be represented as one neighborhood where each particle is connected to each other. It is called the *gbest* neighborhood and is illustrated in Figure 3.11. The 6th particle in red is the global best. This has been proved to be a very fast algorithm but often leads to local minima [12]. Depending on the application, this
Figure 3.10: Flowchart illustrating the PSO algorithm inspired by [8].
topology can be useful but lacks in its robustness.

Figure 3.11: Representation of the \textit{gbest} neighborhood [11].

One another widely used topology is represented in Figure 3.12. It is called the \textit{lbest} neighborhood and has a shape of a ring. It is made of sub swarms composed of three particles. Here, particle 2, 4 and 8 in blue are the local best of their own neighborhood. The particle 6 in red is the global best. This topology provides a better robustness but takes more time in its process. The main advantage of this neighborhood is that it does not get stuck in a local optima [12].

The size of these neighborhoods is also very important as it can lead to more solutions thanks to the interconnection of multiple sub-neighborhoods. Some examples of different sized ring topology neighborhoods are illustrated in Figure 3.13.
Figure 3.12: Representation of the \textit{lbest} neighborhood [11].

Figure 3.13: Representation of the \textit{lbest} neighborhood of different sizes [12].
4 IMPLEMENTATION

This Chapter will describe the hardware used and how a scheduling strategy could be implemented theoretically on this hardware. The used board, which is a Pandaboard, will be detailed in this Chapter as well as its most important components. Their description will permit us to know how they compose an heterogeneous system and how they play an important part in the scheduling method on this board.

4.1 Platform

4.1.1 Pandaboard ES

The used hardware is an OMAP4460 Pandaboard ES [13]. It is composed, as its name indicates, of an OMAP4460 Processor [14] distributed by TI that will be discussed and detailed later in this section. The Pandaboard ES also has major components such as: TWL6030 Power Management Companion Device that permits to power other components on the board, TPS62631 switching power supply to power up the ARM cores, JTAG, UART, LAN9514 USB hub and Ethernet controller, POP Mobile LP-DDR2 SDRAM Memory. Two HDMI Connectors of type A. One of which is a HDMI transmitter output and the other one is a DVI-D output sourced via OMAP4 parallel display output. Audio connectors, SD/SDIO/MMC Media Card Cage.

The overview of its architecture is shown in Figure 4.1.
Figure 4.1: Pandaboard ES architectural block diagram [13].
Figure 4.2: OMAP4460 block diagram [14].


4.1.2 OMAP4460

The OMAP4460 Processor is the main part of the board [14]. It supports different OS such as Windows, Linux, Android and is composed of a large variety of subsystems such as: a dual-core ARM Cortex-A9 cores 1.2Ghz frequency, Digital Signal Processor (DSP) subsystem that will be detailed later in this part. Image and video accelerators IVA, SGX 540 2D/3D graphics accelerator, Cortex-M3 MPU subsystem including two ARM Cortex-M3 microprocessor. Display subsystem, Image Signal Processor (ISP) and other minor parts that can be found in the TI data-sheet [14].

Figure 4.2 shows a detailed block diagram of the OMAP4460 in which we can see all the different subsystems and their interconnections. Most of the subsystems and especially the DSP and the dual-core ARM cortex-A9 cores are linked and can communicate with each other by sharing interconnects (see L3 Interconnect on the Figure 4.2).

4.1.3 Dual Cortex-A9

The Dual Cortex-A9 is the multi core processor unit which is composed, as its name indicates, of two processing units which have the same characteristics. The Cortex-A9 subsystem also includes, as the Figure 4.3 shows, means of communication with two interconnects which will be detailed later (L3 and L4). Each CPU is composed of a cache memory that is shared and a memory management unit (MMU). The Dual Cortex-A9 is basically the main processing unit in the device and will be the one that will handle the communication between the other subsystems and especially with the DSP.

4.1.4 DSP

The digital signal processor subsystem is composed of the following components: DSP megamodule [14] composed of a high-performance TI DSP (TMS230DMC64x). Two levels shared cache and memory controller (L1 and L2). A subsystem counter time module SCTM, local interconnect network, power management and clock generation modules, a dedicated enhanced date memory access (eDMA) for downloading and uploading data, a dedicated memory management unit (MMU) for accessing L3 in-
Figure 4.3: Cortex-A9 subsystem top-level architecture [14].
Figure 4.4: DSP subsystem top-level architecture [14].

The communication between the DSP and the dual Cortex A-9 cores through the L3 and L4 interconnect will be explained later in this part as it is what makes, in a way, this platform heterogeneous (The fact that different types of processors can communicate with each other).
Figure 4.5: DSP megamodule top-level architecture [14].

DSP Megamodule

The DSP megamodule shown on Figure 4.5 is composed of the C64x+ core, Program Memory Controller (PMC), Data Memory Controller (DMC) and two unified cache L1, L2. These modules are controlled and configured by the local DSP interconnect. This interconnect is directly linked to the L2 cache controller, itself linked to the L1 cache.

4.1.5 Interconnects

There are two types of interconnect that are of interest for the communication between the Dual Cortex A-9 and its remote processor DSP:
1. L3-Interconnect

2. L4-Interconnect

The L3-Interconnect works as a sort of shared bus between all the processing units. It can be used to transfer different type of data with a maximum width of 128 bits from the sender to the receiver [14]. Figure 4.6 shows all the subsystems linked to the L3-interconnect as well as the other interconnect L4-CFG. Each module connected to the L3-interconnect has an ID (called ConnID) that will permit to identify the sender and the receiver during data transfer. The L4-CFG-Interconnect handles the configuration of several modules in the board such as the Digital Signal Processor and the Mailbox module that will be detailed later in this part. The complete list of the target modules that are configured through the L4-CFG interconnect can be found in the OMAP4460 datasheet [14]. Just like for the L3 interconnect, each module which has an access to the L4 interconnect is assigned an ID.

Figure 4.6: L3-interconnect overview [14].
4.2 Communication between the GPP (Cortex A-9) and the DSP

There are several means of communication possible in the Pandaboard ES between the processing units: Spinlock, Mailbox and Shared Memory. These means permit to send messages and receive messages back between two processors as well as sharing information. Firstly, the spinlock module will be detailed. Secondly, an explanation of the Memory Management Unit will be given. Finally, the mailbox will be detailed as well.

4.2.1 Spinlock

Spinlock or hardware spinlock is a module which provides hardware assistance for the synchronization of processes running between the processing units in an heterogeneous platform. For the OMAP4460, it allows the DSP and the Cortex-A9 to synchronize tasks between them as well as performing mutual exclusion. Figure 4.7 represents the structure of the spinlock device. It is composed of 32 hardware semaphores to service tasks between the heterogeneous processors. Each spinlock accepts
and runs one request at a time [14]. The spinlock device is configured with the L4-CFG-interconnect. The PRCM mini-module handles the power of the spinlock device as well its reset and its clock management.

### 4.2.2 MMU (Memory Management Units)

![MMU overview](image)

The OMAP4460 is composed of three Memory Management Unit, one for each processing unit (Cortex-A9; Cortex-M3 and the DSP). A MMU is a hardware device that provides the virtual memory for the subsystem. The memory of each subsystem is then shared with each other. Figure 4.8 shows how each MMU is connected with each other, as they share the same interconnect L3 that will permit to configure the MMUs.

### 4.2.3 Mailbox

The Mailbox facilitates the communications between the different processors, and especially between the ARM Cortex A-9 and the C64x DSP. The sender sends data to a receiver by writing a message to the mailbox registers "monitored" by interrupt signals in order to notify the receiver that a message is in the mailbox queue. The mailbox module includes eight mailbox message queues, four FIFO and three interrupts.
Figure 4.9: Mailbox system integration [14].
Figure 4.9 shows the integration of the mailbox system. There are the three interrupts for the three users: the Cortex A-9, the DSP and the Cortex A-3. When an interrupt request is enabled, the subsystem (A-9, A-3 or DSP) is notified. It means that one of two events has occurred: a message has been received by a subsystem which enables its interrupt, or the message-queue is not full. In both case, the subsystem of the mailbox is notified. Figure 4.9 also shows that the configuration of the mailbox messages is done through the L4-CFG-Interconnect. This interconnect permits to query the interrupt status register of each user [14].

Figure 4.10: Communication between DSP and ARM Cortex A-9 [14].
An example of communication between the A-9 and the DSP is illustrated in Figure 4.10. It shows the two users 0 and 1 respectively the Cortex A-9 and the DSP. The Cortex A-9 writes a message in the mailbox (MAILBOX_MESSAGE_0) through the L4-CFG interconnect. This message will trigger the mailbox interrupt of the DSP. Once the message has been read by the DSP, the DSP (user 1) sends a message back (MAILBOX_MESSAGE_1) to the Cortex A-9 (user 0). It will trigger the interrupt (new message in the queue) so the Cortex A-9 is notified of the arrival of a new message and is ready to read it.

4.3 Example of an implementation

In this implementation part, we will see how a simple application made of five tasks can be scheduled on the OMAP4460 described earlier in this Chapter. We will see how the different tasks can be divided and how the communications between the processing units play their part in the scheduling process.

4.3.1 Problem Definition

![Diagram of five tasks](image)

Figure 4.11: Simple representation of an application made of five tasks inspired by an example [9].

49
For this problem, a simple application made of five tasks will be used as the example. We will take into consideration two processing units the OMAP4460 has to offer: the DSP and the Cortex-A9, which make the platform heterogeneous. Figure 4.11 shows a representation of the application in its directed acyclic graph form. Each task is given its execution time on the two proposed processors. The processors are of different nature, hence each task has a different execution time considering on which unit the task is processed. The execution times, for this example, have been chosen randomly and can be seen in Figure 4.12. The communication times are represented in Figure 4.13. The communication time between two successive tasks is considered to be 0 if the two successive tasks are executed on the same processor.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Processing units</th>
<th>Execution times (units of time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>DSP, Cortex A9</td>
<td>10, 5</td>
</tr>
<tr>
<td>Task 2</td>
<td>DSP, Cortex A9</td>
<td>5, 9</td>
</tr>
<tr>
<td>Task 3</td>
<td>DSP, Cortex A9</td>
<td>3, 7</td>
</tr>
<tr>
<td>Task 4</td>
<td>DSP, Cortex A9</td>
<td>4, 10</td>
</tr>
<tr>
<td>Task 5</td>
<td>DSP, Cortex A9</td>
<td>20, 40</td>
</tr>
</tbody>
</table>

Figure 4.12: Execution time of the tasks on the processors inspired by [9].

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Task successors</th>
<th>Communication times (units of time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>Task 3, Task 4</td>
<td>2, 4</td>
</tr>
<tr>
<td>Task 2</td>
<td>Task 5</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 4.13: Communication time between the tasks inspired by [9].
4.3.2 Problem Solution

Many different schedules could result from the graph represented in Figure 4.11 leading to different solutions. Only one will be used as an example.

We will choose to execute task 1 and task 3 on the cortex A9 while tasks 2, 4 and 5 will be processed on the DSP. For each processor, a task processing order is chosen. On the cortex A9, task 1 must finish processing before the task 3 starts. On the DSP, two different orders are conceivable. First order would be to process the task 2, then task 5 and finally task 4. The second order would be the processing of task 2, then task 4 and finally task 5. For this example, the latter order will be taken into account (task 2 then task 4 then task 5). Since the different communication time and execution time are known in advance, the schedule can be done offline and its makespan can be calculated. The Cortex A9 and the DSP must then communicate with each other so the task processing and ordering can be done effectively.

To facilitate the communications between processing units on heterogeneous platforms such as the OMAP4460, TI has developed a framework called RPMsg [33]. RPMsg is the evolution of the IPC (Inter Processors Communication) Syslink that was used on the previous OMAP platforms. The communications between the Cortex-A9 and the DSP can then be done with this IPC. RPMsg makes use of the shared memory, the mailbox and the spinlock hardware together to facilitate the data transfer between processors. It permits to help offload the most intensive tasks of the main CPU to the DSP [34]. RPMsg is the tool on the OMAP4460 that permits to make a task schedule work on the heterogeneous platform. RPMsg framework is composed of two main elements: remoteproc and rpmsg.

The remoteproc is the component which permits the user to have an access to the remote processors of the board, which is the DSP in this case. The remoteproc works like a kernel and is used to, for example, boot up the DSP. It also takes care of the power management and has access to the MMU of the DSP [33] discussed previously in this Chapter. The remoteproc is programmable from the main processing unit of the OMAP4460, which is the Cortex A9.

The rpmsg part is the remote processor messaging bus, which enables client drivers to communicate with the DSP. It will permit the user to send requests from the GPP to the DSP as well as receiving messages back from the DSP to the GPP. The framework makes use of the architecture of the board and especially of the mailbox module to
enable the communications between the two processing units.

The use of RPMsg framework on the OMAP will then enable the cortex A9 to send the different requests to the DSP so the remote processor executes the right tasks in the right order.

As previously mentioned, the schedule chosen in the example consists in executing the task 1 then the task 3 on the main processor while the task 2 then task 4 then task 5 are respectively processed on the DSP. According to the execution time, it takes $5 + 7 = 12$ unit of time to process task 1 and task 3. There is no communication time between task 1 and task 3, as they are processed on the same processing unit. It is also assumed that a request is sent from the GPP to the DSP via RPMsg in order to execute task 2 onto the DSP (the communication time of the first message sent to the DSP from the GPP is not taking into account. In other words, task 1 and task 2 start their process at the same time). It requires 5 unit of time to execute task 2. The following task to be executed on the DSP is task 4, which also appears to be a successive task of task 1. After task 1 has finished, a request is sent by the GPP to the DSP, using the RPMsg framework. That request is a message permitting the DSP to process task 4. The communication time for this request will be of 4 unit of time, as mentioned in Figure 4.13. Considering the execution times of task 1 and task 2 are both 5 unit of time, the task 4 will be able to start its process at $5 + 4 = 9$ unit of time and will be processed during 4 unit of time. Task 4 finishes its execution on the DSP at 13 unit of time. The last task remaining is the task 5, which must be executed after the task 4 on the DSP (as the order was defined previously). The execution of task 5 takes 20 unit of time. It means that task 5 will finish its process at $13 + 20 = 33$ unit of time. We can now calculate the makespan of our schedule as all the tasks have finished their execution. To process the tasks on GPP, it takes $5 + 7 = 12$ unit of time. To process the tasks on the DSP, it takes $5 + 4 + 4 + 20 = 33$ unit of time. Thus, the critical path in unit of time [9], also called the makespan of the schedule is 33 unit of time long. In this example, it corresponds to the sum of the execution time of task 2, task 4 and task 5 on the DSP as well as the communication time, made possible by RPMsg, between the GPP and the DSP (between task 1 and task 4). Figure 4.14 displays the schedule of the tasks on the processors of the OMAP4460. Many others schedule solutions can be found given the example that has been chosen.
This simple example illustrates the mechanisms of scheduling an application on the OMAP4460 when the host processor and its remote processor are in charge of executing the different tasks of the application. The communication between the two processors, which is fundamental, is facilitated by the RPMsg Framework.

Figure 4.14: Schedule solution.
5 Future Work and Conclusions

5.1 Conclusions

This thesis depicted the matter of scheduling applications onto heterogeneous systems. Scheduling an application accurately is one of the most important areas in embedded systems. Finding good scheduling solutions permits to meet different real-time constraints as well as having the lowest power consumption possible, which is another important feature nowadays in embedded systems.

Before scheduling an application, it is important to know how the said application can be represented. An accurate knowledge of the architecture of the targeted platform is also important in order to produce the most suitable schedule. In this thesis, the applications are partitioned into different tasks, which are dependent on each other. One way to represent these tasks is to use a directed acyclic graph, which enables us to make use of the tasks dependencies and their different properties.

When it comes to scheduling, it has been shown that different strategies can be used: static scheduling strategies and dynamic scheduling strategies. For each scheduling policy, a few algorithms have been detailed to explain the mechanisms of scheduling tasks onto multiprocessor platforms. The HEFT (Heterogeneous Earliest Finish Time) and the CPOP (Critical-Path-On-a-Processor) algorithms, two static scheduling strategies, permit to schedule a set of tasks on multiprocessors platforms. They are said to be static because the scheduling decisions are made off-line considering the different task properties (execution time, communication time, task dependencies). As for dynamic scheduling strategy, the EDF (Earliest-Deadline-First) algorithm was chosen in order to depict the mechanisms of the dynamic strategy, which differs from the static one. Indeed, as its name indicates, the dynamic strategy takes into account tasks that can change of priority during their process. It is an interesting scheduling strategy when the tasks are not predictable in advance.

Mapping and ordering tasks onto heterogeneous platforms become more advanced
nowadays as the applications are more complex and more demanding in performance. It also implies more powerful and more complex platforms. To be able to schedule such applications on such platforms, some optimization tools have been created. In this thesis, two off-line tools and their mechanisms have been described. First of all, the GA (Genetic Algorithm), which is based on chromosomes’ behavior. Each chromosome, which represents a scheduling solutions from a DAG, swaps its genes with other chromosomes in order to create new individuals with hopefully better characteristics. Successive crossovers and mutations will result in a better scheduling solution.

Another off-line optimization tool described in this thesis is the PSO (Particle Swarm Optimization), which is a population based algorithm inspired by the behavior of the swarm of birds. Several scheduling solutions compose the population, which moves through the problem space. Each individual has its own coordinates in the problem space. At each iteration, the individuals composing the population move towards the best solution possible. The process is finished when the best schedule meets the different requirements set by the user.

Finally, this thesis detailed a commonly used board, which has interesting characteristics for the topic of this thesis. The board chosen is the OMAP4460 Pandaboard ES, which is composed of a few different processing units. Indeed, this Pandaboard features a host processor (Cortex A9) and a remote processor (DSP). The heterogeneity of this platform gives the board interesting and useful properties, especially when it comes to scheduling applications. The example used in this thesis showed how tasks composing an application can be mapped and ordered on the GPP and on the DSP considering different factors of both the tasks and the processors. In order to schedule tasks, the processors need to communicate with each other. These communications are possible thanks to the architecture of the OMAP4460 as well as a useful framework called RMPsg facilitating the requests and messages between the GPP and the DSP.

5.2 Future work

A novel approach to schedule applications on embedded systems thought by Zaki has been described in this thesis. New ways of representing complicated DSP application such as a PEG (Partial Expansion Graph) enables us the find better schedules by making the most of the application characteristics. Indeed, by taking advantage of the
different task parallelism, data parallelism as well as the pipeline parallelism that an application can provide, we can schedule accurately an application on a heterogeneous platform.

As scheduling tasks is one of the most research areas in embedded systems, it is important to find new approaches in the representation of applications to be able to map and order correctly tasks onto the more and more complex platforms that are built nowadays.
BIBLIOGRAPHY


57


59

