N° d'ordre : 10/2007-E/IN

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université des Sciences et de la Technologie Houari Boumediene (U.S.T.H.B)
Faculé d'Electronique et d'Informatique

Thesis

Presented in order to get the **DOCTORAT d'Etat** Diploma

In Computer Science

By

Hadda CHERROUN

Theme

Scheduling for High-Level Synthesis

Defended on 10 december 2007

Abstract

Scheduling is one of the important tasks in High-Level Synthesis (HLS). Scheduling a whole program, especially with loops, is hard as too many constraints and objectives interact. We propose to organize scheduling in gradual ways. This thesis focuses on some steps of the designed scheduling approaches. An efficient formalism to express resource constraints, using dis-equations, is presented. In the first part, we examine the problem of Resource-Constrained Scheduling (RCS) tasks whose resource usage is described by reservation tables, while in the second one, we adress the problem of RCS data-dependent tasks. For both problems, several algorithms are proposed. Our main algorithmic contributions are: 1/ an exact branch-and-bound (BAB) algorithm, where each evaluation is accelerated by variants of Floyd's and Dijkstra's algorithms, 2/ a new scheduling method based on graph coloring technique as a tool for a BAB meta-method, where each evaluation is accelerated by maximal and greedy clique computation. The evaluation and comparisons are done on pieces of real-life applications from the PerfectClub and the HLSynth95 benchmarks. The results demonstrate the suitability of these solutions for HLS scheduling.

Keywords: Scheduling, resource constraints, reservation tables, dis-equations, branch-and-bound, Dijkstra, graph coloring, integer linear programming.

Résumé

L'ordonnancement est l'une des tâches les plus importantes dans la synthèse de haut niveau. Vue l'importance des objectifs et des contraintes qui interagissent, il est dur d'ordonnancer, un programme en entier, en particulier lorsqu'il contient des boucles. Pour cela, nous proposons d'hiérarchiser l'ordonnancement en niveaux graduels selon différentes approches. Cette thèse se concentre sur quelques étapes de ces approches conçues. Un formalisme efficace exprimant les contraintes de ressources en utilisant les dis-équations, est présenté. Dans une première partie, nous examinons le problème de l'Ordonnancement sous Contraintes de Ressources (OCR) de tâches dont l'utilisation de ressource est décrite via des tables de réservation, tandis que dans la seconde partie, nous abordons le problème d'OCR de tâches dépendantes. Nos principales contributions sont: 1/ un algorithme exact de type Branch-and-Bound (BAB) associé à une variante de l'algorithme de Dijkstra, 2/ une nouvelle méthode d'ordonnancement basée sur la technique de coloriage de graphe et qui est résolue au moyen d' un BAB associé à un algorithme de calcul de clique (exacte/gloutonne). Les algorithmes proposés ont été implémentés. Le jeu des programmes tests est pris d'applications réelles du PerfectClub et HLSynth95 benchmarks. Les résultats prouvent que les deux méthodes conviennent aux outils HLS.

Mots-clés: Ordonnancement, contraintes de ressources, tables de réservation, dis-equations, branch-and-bound, Dijkstra, coloriage de graphe, programmation linéaire en nombres entiers.

Acknowledgments

I would like to thank my thesis director, Professor Mohamed AHMED NACER, for his ultimate help and advices.

I would like thank also my advisor, Professor Paul FEAUTRIER, giving me the opportunity of working on this thesis in his laboratory with the members of his team, for his support and his directions. This work couldn't have been done without the advices of several other people. First, I would like to acknowledge Alain DARTE, CNRS research scientist at LIP laboratory ENS Lyon, for his ultimate help and participation in the research results.

Specially, I would like to thank Dr. AliMazighi, Professor Nadjib BADACHE first for the fact that they made a difference in my student life and for have accepted to be member of my thesis committee. Dr. Mouloud KOUDIL is also thanked to have accepted to evaluate my thesis.

Also, I would like to thank many of my friends met at the LIP laboratory whose comments helped me at the different stages of this research: Fabrice BARRY, Antoine SCHERRER, Nicolas FOURNEL, Anne BOUILLARD and Eric THIERRY.

A thesis is more than its technical contributions, it is also a personal achievement. For this, I would warmly thank my family and friends. First, my husband Mouloud who joined me in this adventure. Thank you for your loving trust and your support. I would also like to thank my family: my mother for her loving and ultimate help and support, my brothers and sisters whose encouragements throughout my studies have always been the greatest help.

This is also a good opportunity to thank some of the people who really made a difference in my student life. K. REGHI and B. CHATTA professors at Benammer Collége and A. BENGUIT at Hadj Aissa Lycée and most of my teachers at the USTHB university especially N. BENSAOU and N. MADANI.

Finally, my deepest gratitude goes to my close friends Attika, Badra, Hayet, Malika, Mounira, and Zineb. Their friendship and love have helped me a lot and their acknowledgment cannot be expressed by words.

To Allah

To Mouloud
To my Mother

Table of Contents

List of Figures viii

List of Tables ix

1 Introduction 1

1.1 Synthesis and High-Level Synthesis 2

1.1.1 Partitioning 3

1.1.2 Scheduling 3

1.1.3 Allocation -Binding- 4

1.2 Objectives, Constraints & Peculiarities when Scheduling in HLS 4

1.2.1 Constraints 4

1.2.1.1 Precedence/Control Constraints 4

1.2.1.2 Resource Constraints -Sharing Resources- 5

1.2.1.3 Time Constraints 5

1.2.1.4 Special Constraints 5

1.2.2 Datapath Peculiarities 5

1.2.3 Interaction with Allocation 6

1.3 Context 6

1.4 Scheduling Problems Addressed 8

1.5 Objectives 9

1.6 Contributions 9

1.7 Thesis' Overview 10

2 State of the Art 11

2.1 Internal Representation and Constraints Formalisms 11

2.1.1 Internal Representation Formalisms 11

2.1.2 Constraints Formalisms 12

2.2 Scheduling Strategies 13

2.2.1 Basic Algorithms 14

2.2.1.1 ASAP 14

2.2.1.2 ALAP 15

2.2.2 Critical Path Method 15

2.2.3 List-scheduling 15

2.2.4 Force-Directed Scheduling (FDS) 16

2.2.5 Path-Based Scheduling 17
2.2.6 Exact Algorithms 18
2.2.6.1 Integer Linear Programming 18
2.2.6.2 Constraint Programming Technique 20
2.2.7 Miscellaneous Techniques 21
2.2.7.1 Simulated Annealing Based Algorithm 21
2.2.7.2 Genetic Algorithm 22
2.3 High-Level Synthesis Tools 22
2.3.1 GAUT 23
2.3.2 SPARK 23
2.3.3 Ugh 25
2.3.4 MMAlpha 26
2.4 Scheduling using Reservation Tables 27
2.5 Hierarchical basis Scheduling Frameworks 27
2.6 Conclusion 28

3 General Scheduling Approaches **29**
3.1 *Syntol* Project 29
3.1.1 Input Specification: CRP Specification Language 29
3.1.2 Target Architecture: the RTL Formalism 32
3.2 A Schedule the Main Tool to Get an FSMD 33
3.3 Stepwise Scheduling 33
3.3.1 Two-step Approach 34
3.3.2 Three-step Approach 35
3.4 Compilation Flow 37
3.5 *CLooG*: a Code Generator 39
3.6 Conclusion 43

Part I Reservation Tables Scheduling using Dis-equations

4 Formalism and a Greedy Solution **45**
4.1 Introduction 45
4.2 Task and Resource Constraints Formalism 45
4.2.1 Extracting Tasks with Reservation Tables 45
4.2.2 Notations 46
4.2.3 Forbidden Distances 46
4.2.4 Example 1 47
4.3 Scheduling Problem Formulation 48

4.4 How To solve a System of Dis-equation? . . . 48
4.5 A Greedy Heuristic . . . 49
4.5.1 Algorithm . . . 49
4.5.2 Example 1, Continued . . . 50
4.5.3 Experiments . . . 51
4.6 Conclusion . . . 52

5 Integer Linear Programming Approach 53
5.1 General Introduction . . . 53
5.2 Integer Linear Programming Approaches . . . 55
5.2.1 Notations . . . 56
5.2.2 Standard 0/1 Encoding . . . 56
5.2.3 0/1 Simplified Encoding . . . 56
5.2.4 Big-M Encoding . . . 57
5.2.5 Experiments . . . 58
5.3 Conclusion . . . 59

6 Branch-and-Bound-Based Longest-Path Computation Solution 60
6.1 General Introduction . . . 60
6.2 An Exact Branch-and-Bound Solution . . . 61
6.2.1 Finding the Local Bound . . . 62
6.2.1.1 Floyd-based Algorithm . . . 63
6.2.1.2 Dijkstra-based Incremental Algorithm . . . 64
6.2.2 Complexity . . . 66
6.2.3 Speeding up the BAB Algorithm . . . 66
6.2.3.1 Heuristic 1 . . . 68
6.2.3.2 Heuristic 2 . . . 68
6.2.3.3 Heuristic 3 . . . 68
6.2.3.4 Heuristic 4 . . . 69
6.2.4 Experiments . . . 69
6.3 Conclusion . . . 72

7 Comparative Study 73
7.1 Comparative Results and Discussion . . . 73
7.2 Conclusion . . . 76

Part II Resource-Constrained Scheduling using Graph Coloring

8 Scheduling via Branch-and-Bound-Based Graph Coloring 78
8.1 Formalisms . . . 78

8.1.1 Task Model 78
8.1.2 Data Dependences and Resource Constraints Formalism 79
8.2 Scheduling Problem Formulation 80
8.3 How To Color a Graph? 81
8.4 Branch-and-Bound-Based Graph Coloring Solution 83
8.4.1 Branching Rule 83
8.4.2 Evaluation -Bounding- Procedure 83
8.4.3 Algorithm 86
8.4.4 Complexity 87
8.5 Integer Linear Program Solution 87
8.6 Experimental Results and Discussion 88
8.7 Conclusion 88

9 General Conclusion and Future Directions **90**
9.1 Contribution 90
9.2 Future Directions 91

Glossary **94**

Bibliography **97**

A Graph Algorithms **105**
A.1 Depth-First Search Algorithm and its Features 105
A.2 Maximal-weight Paths 107
A.2.1 Relaxation Technique 107
A.2.2 Dijkstra's Algorithm 108
A.2.3 Bellman-Ford's Algorithm 109
A.2.4 Floyd-Warshall's Algorithm 109
A.2.5 Johnson's Algorithm 110
A.3 Roy-Warshall's Algorithm 111

B Benchmark Description **114**
B.1 1995' High-Level Synthesis Design Repository 114
B.2 PerfectClub Benchmarks 114

List of Figures

1.1 The Hierarchical Scheduling Views 8

2.1 (a) Data Flow Graph (b) ASAP Schedule (c) ALAP Schedule 14

3.1 Pipeline System. 32
3.2 Target of HLS - RTL processor- 32
3.3 The Hierarchical Scheduling Views 36
3.4 Design Flow of our HLS System. 37
3.5 Generated FSM for the Pipeline Program. 42

4.1 Forbidden Distance. 47
4.2 Binding for Example 1. 48
4.3 (a) Greedy Solution and (b) Optimal Solution. 51

5.1 Feasible Areas. 54

6.1 Pathological Case css21. 72

8.1 One unit-cycle resource simulation. 79
8.2 Binding for the Example. 82
8.3 The Graphs G, G' and G''. 84

A.1 DFS Properties 107

List of Tables

4.1 Greedy Scheduling Results. 51

5.1 Scheduling Results with the Different ILP Formulations. 59

6.1 Scheduling Results for the Various Tests on the BAB Algorithms. 70
6.2 Improvements on the BAB Algorithm with L_{best} Set to the GS Schedule Latency. 71
6.3 Influence of the Binding Heuristic on Latency and Resolution Time. 71

7.1 Comparative Results. 74
7.2 Comparative Results when the p_i Vary. 74
7.3 Comparative Results with Artificial Data Dependances. 75
7.4 Comparative Results when Splitting a few Macro-tasks. 75

8.1 Scheduling Results for the BAB with both Maximal and Greedy Clique Bounding Algorithms and the ILP. 89

A.1 Summary of Longest-Paths Algorithms. 112

B.1 Features of Designs. 115
B.2 Part of the PerfectClub Benchmarks Suite. 115

Chapter 1

Introduction

In this thesis, the *resource-constrained scheduling problem* in High-Level Synthesis -behavioral synthesis- will be examined, and several algorithms are proposed. The aim of this research is to propose and compare some novel scheduling algorithms, as well from theoretical point of view as experimental one. These algorithms are integrated in *stepwise scheduling approaches.*

With the rise in the complexity of Integrated Circuits (IC's), their design process has become very difficult to manage without any automation or semi-automation. Thus the need of effective and good CAD tools where we can have: shorter design cycle, fewer errors, the ability to search the design space, documenting the design process and availability of digital circuit technology to more people. The challenge of embedded system design is twofold: one must pack compute-intensive algorithms in small platforms; furthermore, the design must be completed as fast as possible, to meet the demands of a highly volatile market. In the long run, this will be possible only if computer-aided design tools are developed far beyond their present status [24].

An artifact such as a cell phone or a digital TV set must behave according to given specifications; however, its hardware parts can only be built from a structural description. The goal of High-Level Synthesis (HLS) is to convert a *behavioral specification* – for the whole or a part of the complete application, to be performed on a dedicated circuit – into a *structural description*, while optimizing several objective functions: performance, size, power consumption among others.

In the past two decades [34] there has been a lot of activity going on in the area of High-Level synthesis and it is becoming an increasingly popular research topic. Currently, several commercial [103, 78, 102] and academic HLS [49, 77, 88, 71, 55, 37] tools exist but the design community don't integrate them into its design flow, because of many reasons: they lack interaction between them and the designers, they can support only limited architectures and the quality of the design they generate is still often worse than that of manual design.

Our contribution focuses on the scheduling problem as it is one key process in HLS. Our aim is to improve those tools by reusing some of methods and models that have been pioneered by the compiler community. Among these powerful methods, operations research techniques have strongly increased the performances of scheduling task.

Scheduling is an important and primary task in HLS. However, scheduling operations under resource constraints to minimize the total duration is $\mathcal{NP}$-complete problem as too many constraints and objectives interact [104, 26]. For efficiently scheduling programs, especially with loops, first, we facilitate the problem by this trick. We propose to organize the scheduling process in **stepwise ways**.

The purpose of these hierarchical decompositions is twofold: one can integrate exact methods into these hierarchical scheduling approaches, that could identify code fragments and schedule them optimally such that it couldn't affect a lot the whole scheduling quality; furthermore, it allows to avoid dealing with a too large system as is well known, applying linear programming to large constraint systems is not cost-effective.

In what follow, we will present briefly the area, the context and the objectives of this research. After what, we report the main contributions. We conclude by giving the outline of the thesis.

1.1 Synthesis and High-Level Synthesis

The High-Level -architectural or behavioral- Synthesis (HLS) process takes a behavioral specification of a system and a set of constraints and goals to be satisfied, and finds a structure that implements the behavior while satisfying the goals and constraints [34]. These goals and constraints can express several objective functions: performance, size, power consumption...

Some times called "hardware compilation", the synthesis of circuits translates a sequential program, into an integrated circuit (hardware). The aim of the synthesis tool is to obtain the physical view of the circuit. This view can be represented by a netlist which is a structural view in the logic-gates level. However, there are several lower-level tools - logic synthesis- which allow the translation to this structural view: Synopsys [103], Cadence [22], Catapult [78], ISE Xilinx tools [113]. Most of them start working on Register Transfer description Level (RTL). Indeed, at the RTL level, the input description is transformed in such way that the register assignation and the functional units assignation are fixed for each cycle of the system [34]. We are interested by the part of the HLS which fills the gap between system " behavioral" level and the RT level by automatically generating an RTL realization from a behavioral description.

Generic High-Level Synthesis System

A typical way of describing behavior is to write a program in an ordinary computer language or in a special hardware description language such as VHDL [29] or Verilog [98].

The first step in HLS is usually the compilation of the behavioral description into an internal representation that is most suitable for HLS tasks. Most approaches use graph-based representations that gather both data flow and control flow implied by the specification. These internal representations are given different names in different synthesis systems (e.g. *value trace*, *data dependency graph* [3], *directed acyclic graph*, *control and data flow graphs*(CDFG) [49]) but are simply different adaptations of similar basic concepts. CDFG is the most popular and in many systems, the control flow graph and the data flow graph are integrated into one structure. Control dependences are derived directly from the explicit order given in the input program and from the compiler's choices of parsing the arithmetic/logic expressions. Data dependences show the essential ordering of operations.

At this stage and like in the software compilation, some important tasks and some optimizations should be performed by the compiler. They include variable disambiguation, taking care of the scope of variables, converting complex data structures into simple types, type checking, expression simplification, dead code elimination, constant propagation, common subexpression elimination....

The second step of the HLS, which is the core of transforming behavior into structure, includes four major tasks:

- *Partitioning:* deals with the division of the intermediate representation (i.e. the behavioral description or the design) into sub-representations in order to reduce the problem size and exhibit more parallelism.

- *Scheduling:* partitions the intermediate representation into time steps, thereby generating a finite-state machine model.
- *Allocation:* though closely intertwined with scheduling, involves partitioning the intermediate representation with respect to space (hardware resources) which is also known as spatial mapping.
- *Control generation:* Once the schedule and allocation have been computed, it is necessary to synthesis a controller (hardwired or microcoded) that will drive allocated resources as required by the schedule. Finally, the design has to be converted into real hardware. Lower level tools, such as logic synthesis and layout synthesis, complete the design.

1.1.1 Partitioning

Software programming languages have little support for describing hardware efficiently. For example, to model hardware in C/C++, we need additional language features present in Hardware Description Language (HDL) but not present in C/C++:

- Concurrency : hardware is inherently parallel, while C/C++ programs and the like are inherently sequential. The notion of processes (*Always* blocks in Verilog HDL, *Process* in VHDL), which encapsulates programs that execute concurrently, have to be introduced. A system will be described as a network of processes.
- Signals : hardware processes need to use signals (akin to wires or buffered channels) to communicate with one another.
- Reactivity : hardware systems are in continuous interaction with heir environment, i.e. they are reactive. The notion of reactivity is essential to describing hardware systems at all levels of abstraction.
- Data abstraction : C/C++ supports data abstractions that are useful for software programming. However, for hardware, one needs arbitrary precision signed and unsigned integers, bit vectors and fixed point types.

With such a set of features added, an imperative language, like C/C++, can efficiently model hardware/software systems. Thus the aim of partitioning is to transform the input description such that it will be easily and efficiently described in HDL.

1.1.2 Scheduling

A Finite State Machine with Datapath (FSMD) model is the most popular one which is used to describe digital systems at the RT level [44]. It consists of an FSM called the control unit and a datapath. The datapath consists of the storage and functional units necessary for the system. The FSM consists of a set of states, a set of transitions between states, and a set of actions (involving the datapath) associated with each transition.

Scheduling, an important task in HLS, can be described as the process of dividing the intermediate representation into states and control steps, in such a way that can be directly synthesized into an FSMD model. In other words, scheduling does a temporal mapping of the given representation. A behavioral description and hence the intermediate representation consists of a sequence of operations to be performed by the synthesized hardware. The task of scheduling partitions these operations into time steps such that each operation is executed in one time step. Each time step corresponds to one state of the controlling FSM state machine in the FSMD model.

1.1.3 Allocation -Binding-

The binding task assigns the functional operations and memory accesses to available hardware units. A resource such as a functional, storage, or interconnection unit can be shared by different operations, data accesses, or data transfers if they are mutually exclusive. Binding consists of three subtasks based on the unit type:

- *Storage binding* assigns variables to storage units. Storage units can be of many types, including registers, registers files, and memory units. Two variables that are not alive simultaneously in a given state can be assigned to the same register. Two variables that are not accessed simultaneously in a given state can be assigned to the same port or a register file or memory.
- *Functional-unit binding* assigns each operation to a functional unit. A functional unit or a pipeline stage can execute only one operation per clock cycle.
- *Interconnection binding* assigns an interconnection unit such as a multiplexer, a wire or a bus for each data transfer among ports, functional units, and storage units.

1.2 Objectives, Constraints & Peculiarities when Scheduling in HLS

Scheduling, a central task in HLS, involves determining the execution order of operations in a behavioral description. In other words, it is the process of determining the assignment of operations to time slots (control steps) of a synchronous system subject to various constraints.

1.2.1 Constraints

The scheduling problem in HLS must take into account different and heterogeneous constraints, which define requirements imposed on an implementation of a system. There are at least two kinds of constraints. The first group, as in classic scheduling problems, comprises constraints that can be **deduced from a system behavioral description**, such as precedence constraints -*data dependences*-, or conditions for operation execution -*control dependences*-. The other group of constraints defines non-functional **requirements for possible implementations** of the system such as *performance*, *cost*, *timing*, *power consumption* or *memory requirements.*

1.2.1.1 Precedence/Control Constraints

Two types of dependencies exist between the operations from a program specification. Data-flow dependencies impose precedence (execution order) between the operations. For example, operation o_2 has to be executed after operation o_1, if a result computed by o_1 is used by o_2. Control-flow dependencies arise when some portions of the specification are executed conditionally. All data-flow and control-flow dependencies have to be satisfied to ensure a correct execution of the specified behavior.

1.2.1.2 Resource Constraints -Sharing Resources-

Additional constraints arise due to finite hardware resources. Resource constraints impose bounds on a number of functional units available for the task execution. For example, a system implementation may incorporate two adder circuits and, consequently, not more than two additions can be executed simultaneously.

1.2.1.3 Time Constraints

Another set of restrictions comes from the timing constraints. In many critical applications e.g. aircraft engine control, computer hardware has to react to a recognition of a specific event within a strictly prescribed time interval.

1.2.1.4 Special Constraints

The resource-constrained scheduling can consider resources very broadly and therefore power consumption and area can also be defined as a resource. In this thesis we don't explicitly address this kind of constraints.

1.2.2 Datapath Peculiarities

Additionally to this panoply of constraints, scheduling in HLS can not be treated without actually considering realistic design models that would have special resources. Indeed the resources usually have some features like functional units with varying delays and multi-functional units. These issues are not explicitly addressed by our main contribution so just to provide to the reader a global idea, we discuss them briefly:

- **Functional units with varying delays:** each functional unit will have a different delay and therefore it assumes that an operation assigned to a control step would take the same time as another operation. This assumption would lead to a clock cycle that is unusually lengthened by the slowest unit in the design. The following three approaches are used to solve this problem:
  - *Pipelining*: A functional unit may have stages in it. This makes it possible to execute two operations in the same functional unit since they operate in two different stages. As known, pipelining is a simple technique to increase parallelism.
  - *Multi-cycling*: If the clock cycle is shortened to allow fast operation, then the slower operation would take multiple clocks and hence are called multi-cycle operations. However input latches are needed in front of the multi-cycle functional units to hold its operands until its results are available. This would in turn increase the size of the control logic. Multi-cycle operations can be pipelined.
  - *Chaining*: Two or more operations could be allowed to be performed sequentially in a single control step (same clock cycle). Since the output of one functional unit has to be fed to another, they should be directly connected.
- **Multi-functional Units:** it has been assumed that a functional unit can perform only one operation but in practice there are several cost effective multi-functional units being used. For this purpose the scheduling algorithms could be technology based so that it can explore the library of components. For example, an operation in the critical path could be assigned faster functional units than those not in the critical path. Also the scheduling algorithm

could try to use the same multi-functional unit for two data independant operations which are in two different steps.

1.2.3 Interaction with Allocation

In order to know whether two operations can be scheduled in the same control step, one must know whether they use common resources. Moreover, finding the most efficient schedule, for real hardware, requires knowing the delays for the different operations, however those can only be found after that the details of the function units and their interconnections are known. On the other hand, in order to make a good allocation, one must know what operations will be done in parallel, which comes from the schedule. Therefore, scheduling and allocation are **strongly interdependent** tasks.

In the literature, many scenarios have been explored. The most straightforward approach to this problem is to set some limits on the resources cost and then schedule, as it is done in many systems. A more flexible approach is to iterate the whole process changing the resource limits until a satisfactory design has been found. An exact approach, but an expensive one, is to develop the schedule and allocation simultaneously, as in system MAHA [88]. Finally, the allocation can be done first, followed by scheduling as it is done in the BUD system [77].

1.3 Context

The schedulers we describe in this thesis are part of our HLS tool, we currently develop, whose aim is HLS in the field of compute-intensive embedded systems. The input specification is a variant of C (including loops); the output is a hardware description at the RT level. We use the finite state machine with a data path (FSMD) model to describe the hardware at this level.

Scheduling is the basic tool we use for hardware generation: a schedule is a precise description of the operations to be executed at each clock cycle; thus deducing the FSMD from a schedule is considered as a natural task.

Earlier work starts by building the control and data flow graph (CDFG), which is simply the sequential flow diagram of the input description. The nodes of the CDFG are the basic blocks of the original program. Most synthesis tools exploit only parallelism inside basic blocks; the FSMD is usually obtained by scheduling the tasks of each basic block of the CDFG independently. Some parallelism is exploited in loops, but mostly through loop unrolling. Our approach is quite different because we first construct a FSMD from an equivalent parallel code that exhibits the inherent parallelism in the input description and takes into account the imperfect loop nests. Afterwards, according to the resource constraints, we exploit a part or all of this parallelism.

Indeed, to extract parallelism from the loops of the input description, we use a scheduling strategy previously used for automatic loop parallelization [41]. This technique which has already proved it efficiency, assigns a symbolic “date” to each **high-level statement** of the program i.e., each statement in the C program and allows us to rewrite the code into a form with explicit parallelism. This technique will be detailed in Chapter 3. The result of this scheduling pass is the definition of a sequence of *fronts*, i.e., a sequence of logical steps where each step (a front) is a group of macro-tasks to be executed in this logical step. From this result, we build a first coarse FSMD; coarse as the time is measured in logical steps instead of clock cycle. Thus, each state contains a set of data independent macro-tasks and each of them is composed by a sequence of elementary operations. However, this symbolic scheduling technique is quite complex and cannot take into account all the micro-operations (and the architectural resources they need) that are implied in the execution of one macro-task. This fact lead us to design **stepwise scheduling approaches** to schedule programs, with loops, down to RTL.

Many factors and reasons let us to think that such approaches can improve a lot the performance of the scheduling algorithm in term of compromise quality of the schedule and its runtime. Indeed, let us notice that the application size, the diverse nature of constraints, all peculiarities of the datapath - features of the resources- and the requirements of a possible implementation can not be taken into account into one pass to get *an optimal* or *suboptimal* schedule without exploding. In Chapter 3, we will explain in details the motivations that lead us to this choice.

Now, we must refine this schedule by "splitting this logical step into elementary steps such that resource constraints and all peculiarities of the datapath will be respected". Time will be measured in physical time i.e., clock cycle of the target architecture. How to perform this? In this thesis, we have sketched and investigated two hierarchical scheduling approaches.

1. *Two-step Approach:*

   After the previous symbolic scheduling step, one natural possibility is to consider, for scheduling under resource constraints, all the micro-operations of a front simultaneously. In other words, schedule them at the same time while considering resource constraints. We call this last step *simultaneous scheduling.*

2. *Three-step Approach:*

   Again for complexity reasons one can alternatively consider another possibility. Indeed, we consider that, in our context, it will be good enough to decouple the problem in two subproblems. This partitioning can be sketched as:

   - After the symbolic scheduling, we first schedule each macro-task *independently*, taking into account all peculiarities of the data path. The schedule of each macro-task will be summarized with a *reservation table*[1] that states which resources at which cycle (relative to the starting time of the macro-task) are used by this macro-task. We call this second step *micro-scheduling.*
   - Due to our particular construction, the macro-tasks, represented now by reservation tables, in a front are data independent but they may still interfere in their use of resources. The logical step must then be split into as few elementary steps as necessary to satisfy detailed resource constraints. We call this third step *fine-grain scheduling.*

The strengths and weaknesses of each solution, as a scheduling approach, are reported and discussed. For instance, one could emphasize that it would be better to consider, the first approach in which the micro- and fine-grain scheduling are performed at the same time. Even though, we have investigated both approaches, the last decoupling will be good enough. it is difficult to prevent the scheduler to introduce delays between micro-operations, and hence to imply more registers for holding temporary results. Our second approach may be sub-optimal, but we believe that the possible improvements do not warrant the added complexity.

Figure 3.3 summarizes both scheduling decompositions. Dotted lines expresses the area of our main contributions as in this thesis we have focused on some levels of these designed gradual scheduling approaches by suggesting mainly two solutions to the resource-constrained scheduling problems defined by the "fine-grain" and the " simultaneous" scheduling steps. Before that, we propose a new formalism for expressing resource constraints using dis-equations.

[1]A reservation table is a matrix that shows the corresponding assignment between control cycles and resource occupation. The rows of a reservation table correspond to control cycles, the columns to resources.

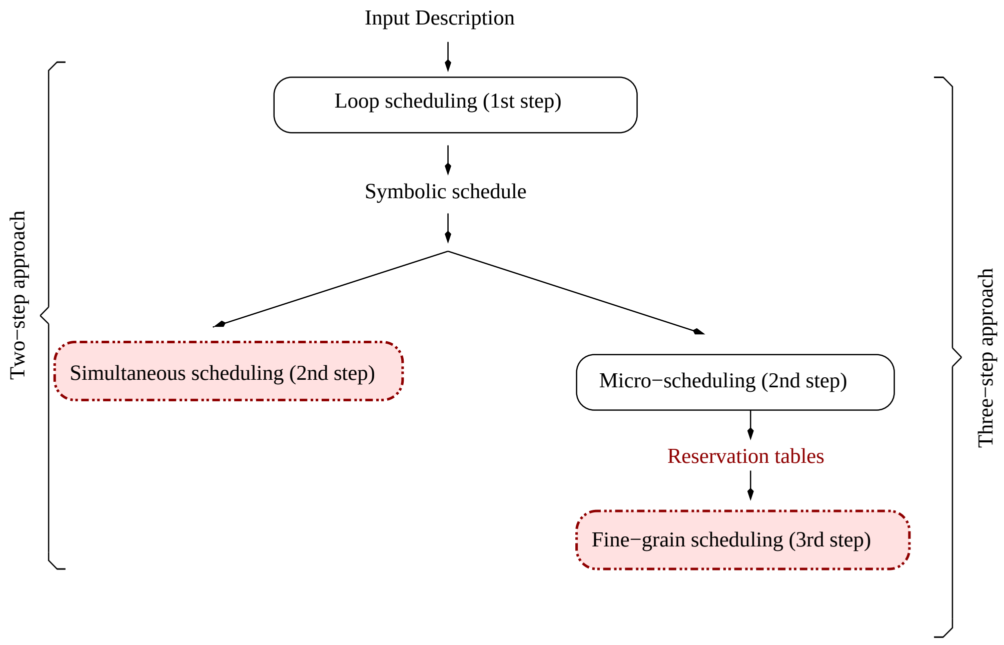

Input Description
Loop scheduling (1st step)
Symbolic schedule
Simultaneous scheduling (2nd step)
Micro−scheduling (2nd step)
Reservation tables
Fine−grain scheduling (3rd step)
Two−step approach
Three−step approach

Figure 1.1: The Hierarchical Scheduling Views

1.4 Scheduling Problems Addressed

Scheduling takes many forms, such as job-shop scheduling, production scheduling, multiprocessor scheduling and so on. To be more precise when defining our scheduling problems, let us recall that scheduling can be of several kinds:

- *Static scheduling*: All information are available to the scheduling algorithm, which runs before any real computation starts. These algorithms are called offline algorithms.
- *Semi-static scheduling:* Information may be known at program startup, or at the beginning of each time step, or at other well-defined points.
- *Dynamic scheduling:* Information aren't known until mid-execution as in real-time systems. These algorithms are called online ones.

In general, solving static scheduling problems under resource constraints is $\mathcal{NP}$-hard. This involves assigning the vertices (tasks) of an acyclic directed graph onto a set of resources, such that the total time to process all the tasks is minimized. The total time to process all the tasks is usually referred to as the makespan or latency.

An additional objective is often to achieve a short latency while minimizing the use of resources. Such multi-objective optimization problems involve complex trade-offs and compromises, and good scheduling strategies are based on a detailed and deep understanding of the specific problem domain.

As defined in our context due to our particular construction, at the fine-grain scheduling step (resp. the simultaneous scheduling), all information about the macro-tasks (resp. operations) in a front are known before scheduling. Thus, we deal with *static* and *acyclic resource-constrained* scheduling problems. They are static as they are done at compilation time and all informations are available, and acyclic as each front is a sequence of data/control independant macro-tasks without loop.

Now, we summarize in these definitions both scheduling problems addressed in this thesis:

Definition 1. *Fine-grain scheduling: is the process of determining the optimal assignment of macro-tasks, defined by reservation tables, to time steps on a synchronous system, subject to resource constraints.*

Definition 2. *Simultaneous scheduling: is the process of determining the optimal assignment of operations to time steps on a synchronous system, subject to data dependences and resource constraints.*

1.5 Objectives

In this thesis, the scheduling approaches, that we have designed and their corresponding techniques which are developed to solve part of scheduling levels have multiple objectives:

- Exploit efficiently more parallelism in the whole program particularly within nested loops; by efficiently partitioning the input description and applying some techniques that have proven their efficiency in automatic parallelization;
- Deal with the constraints into higher levels in the HLS design process where a global view is more clear than in lower levels;
- Formally and uniformly express constraints (data dependence and resource constraints);
- Bring some guarantee on the quality of the computed schedules;
- As the defined problems are $\mathcal{NP}$-hard, try to integrate exact methods into hierarchical scheduling approaches, that could identify code fragments and schedule them optimally. Consequently deal with manageable code size that don't reach these exact methods limits.

1.6 Contributions

This thesis presents some solutions to the resource-constrained scheduling problems for HLS. Indeed, for efficiently scheduling programs, especially with loops, first, we make easy the problem by organizing the scheduling process in stepwise ways. We propose two approaches: Two-step and Three-step approaches.

First, a formalism to accurately express resource constraints for complex tasks represented as reservation tables is proposed. Indeed, the resource constraints are modeled by "dis-equations" and finding an optimal schedule entails resolving a system of dis-equations. The proposed formalism can be generalized to support problems of resource-constrained scheduling even when tasks are data-dependent.

For both approaches, we have proposed some solutions to certain scheduling steps. Indeed, for the Three-step approach, we have focused on the fine-grain scheduling step. We have proposed several solutions for scheduling macro-tasks -defined by reservation tables-: 1) a greedy heuristic similar to list-scheduling and 2) two exact algorithms, the first one uses ILP techniques while the second one is based on a branch-and-bound meta-algorithm using a variant of Dijkstra's algorithm and Floyd's algorithm which compute a maximal weight path.

Within the Two-step scheduling approach, we have proposed some solutions to perform the simultaneous scheduling step. First, we use “ dis-equations” as mathematical way to express uniformly both constraints: resource constraints and data dependences. Afterward, we propose a novel scheduling algorithm that finds an optimal schedule by properly coloring the conflict graph. In order to optimally coloring a graph and conversely to classic graph coloring algorithms, we designed a new method so that coloring is done by means of a branch-and-bound that is accelerated by a clique computation algorithm. The clique computation is computed exactly as well as greedly.

The greedy heuristic and the ILP based algorithms are used as yardsticks for measuring the efficiency and robustness of our main algorithmic contributions.

1.7 Thesis' Overview

The outline of the rest of the thesis is:

- In Chapter 2, we present some related work both for HLS constrained scheduling in general and for scheduling with reservation tables in particular.
- We detail our general scheduling approaches in Chapter 3. Indeed, in this chapter we give a detailed overview of our HLS tool and we explain and discuss the reasons that have led us to design such hierarchical scheduling strategies. However, the rest of the thesis focuses just on the fine-grain and the simultaneous scheduling problems.
- In Part I, our problem of scheduling tasks defined by reservation tables is treated and many solutions are proposed. First, the problem is formulated in Chapter 4, where we present, also, a simple greedy heuristic which will be compared with the proposed exact algorithms. The first outcome of this research, presented in Chapter 6, is an exact branch-and-bound algorithm, where the evaluation of each potential solution is accelerated thanks to variants of Floyd's and Dijkstra's algorithm. Chapter 5 presents several Integer Linear Program formulations of the problem. In each of these Chapters, we report at the end some experimental results to highlight the benefits of each solution. Lastly, in Chapter 7, we compare and analyze these experimental results and demonstrate the effectiveness of the proposed methods. Furthermore, we give some guidelines for selecting the most effective one according to the context.
- The second contribution, described in Part II, is a solution to the simultaneous scheduling problem. In this solution, the dis-equations -representing data dependences and resource constraints- are modeled by an interference graph and the scheduling problem is resolved using a novel graph coloring technique. Some experiments and results are reported;
- Finally, in Chapter 9, general conclusion is drawn as well as the questions to be addressed in future.
- Abbreviations and some definitions from digital systems and scheduling literature are given in the Glossary (page 94). A background on some used elementary graph algorithms is provided in Appendix A. Finally, Appendix B describes the benchmarks kit that is used to evaluate the performances of the designed algorithms.

Chapter 2

State of the Art

Much work has been done in the area of HLS. The three major HLS tasks –allocation, scheduling and binding– have been widely researched in the last two decades. To get an exact survey on what has been done on scheduling is a difficult task. This is due to the large number of target implementations and specificity of contexts. This chapter tries only to highlight some works related to our contributions. Indeed, in the following section, some works related to several internal representations used in HLS tools, are reported. This is followed by how constraints are formalized to efficiently carry out the scheduling task. In Section 2.2 some major search techniques instrumented in scheduling algorithms are exposed and discussed. Some recent HLS tools are described in Section 2.3. During the description we concentrate on their scheduling algorithms. In Sections 2.4 and 2.5 some special researches are underlined: the usage of reservation table in scheduling and the stepwise scheduling approaches.

2.1 Internal Representation and Constraints Formalisms

The behavioral description and constraints are the main inputs of the constrained scheduling problem. The formalism of the internal representation of the description and the formalism used to express the constraints must be efficiently designed to simplify the schedule computation.

2.1.1 Internal Representation Formalisms

Before all the activities of an HLS system, the behavioral description has to be represented into an internal format. It is commonly agreed that the intermediate design representation is closely linked to the quality of the scheduling results. The design of an internal representation is also important for the simplification of the engineering of the HLS techniques used to efficiently carry out the HLS design activities.

The most popular internal representation of the input behavioral description is the Control and Data Flow Graph CDFG [49]. For this, most scheduling techniques are graph-based models. In these models, basic tasks are gathered into a set of basic blocks linked by flow-of-controls. A flow-of-control can be forward or backward. A forward edge represents a move from a basic block to a successor block, while a backward edge represents a loop.

However, there are some frameworks which investigated more efficient internal representations to get easily the HLS activities specially the scheduling task [16, 60, 114].

For instance, the intermediate representation used in the *SPARK* tool [55] consists of basic blocks encapsulated in *Hierarchical Task Graphs* (HTG) [50]. It is an intermediate representation designed by the parallelization community to ease the automation of transformations.

An HTG is a directed acyclic graph that has three node types: single nodes (non-hierarchical nodes), compound nodes (nodes that have sub-nodes), and loop nodes. Operations that they are executed concurrently are aggregated together into single nodes called statements. Statements that have no control flow between them are aggregated together into basic blocks. These laters are encapsulated into compound HTG nodes to form hierarchical structures such as if-then-else blocks, switch-case blocks, loop nodes or a series of HTG nodes. Expressions are stored as abstract syntax trees [80] and each operation expression is initially encapsulated in a statement node of its own.

An important feature of HTG is that they are strongly connected components i.e., for each couple of nodes there are at least one path. Furthermore, each component has a single entry and a single exit point. This property enables HTG to be used to encapsulate complex loops and irregular code regions of code, to regularize code motion techniques and to reduce the amount of patch-up code.

For control-dominated systems, Bergamaschi et al. developed a *path-based* representation [16]. They extract from the CDFG a new representation where all possible paths are exhibited. Using this representation, they optimize the control-step number by path (see Section 2.2.5 for more details on path-based scheduling).

Huang et al. extended the path-based representation to a *tree-based* representation [60]. They aimed at removing the restriction on the execution order of operations before scheduling imposed by the path-based representation. Instead of treating each path individually as in path-based representation, all paths are kept in a tree.

Recently, more symbolic representation are used [94, 114], especially when systems are control dominated. For instance, Radivojevic et al. [94] present an exact conditional resource sharing analysis using a symbolic representation formalism. Indeed, in this formulation, all of the scheduling constraints (precedence, control and resources) are represented as boolean equations. An *Ordered Binary-Decision Diagram* (OBDD) corresponding to their intersection is built. Each variable in the OBDD describes a particular operation occuring at a particular time step, over a finite set of time steps. A variable is true if the corresponding operation is scheduled during the corresponding time step in particular solution. To allow control-dependent scheduling a set of 'guard' variables is introduced –true for one branch and false for the other–. Thus, every path through the CDFG is encoded by a product of corresponding guards. Their formalism allows the generation of a set of valid schedules via a compressed representation based on the OBDD representation.

Yang et al. introduce another symbolic representation based on *automata representation* [114]. In this model, with a uniform formalism; both the design and constraints (timing, resource, synchronization . . .) are represented by a unique automaton based also on Binary Decision Diagrams.

Despite these attempts to find suitable internal representations, there is some agreement that design representation is not a mature topic. In our framework, we have avoided using the CDFG representation extracted from the sequential input description. Indeed, we start our HLS activities on an automatic generated internal representation in which we exhibit all inherent parallelism, we will return to this fact with more details in Chapter 3.

2.1.2 Constraints Formalisms

The constraints that a hardware system has to respect come from different domains: classic precedence/control constraints, constraints defining non-functional requirements for possible implementations of the system such as performance, cost, timing, power consumption, area or memory requirements. The formalism used to express these various constraints directly acts upon the technique used to compute a schedule.

Attempts have been made to define these constraints in a formal way using constraint description languages, such as the *Design Constraints Description Language* DCDL [1]. However, during the design process these constraints are usually treated informally and most today's tools do not unify different requirements in a single formalism.

Recently, a more general formalism has been proposed by Kuchcinski [68]. He designed a model for solving both scheduling and resource assignment problems by using constraint handling methods provided by the Constraint Programming paradigm CP. In this approach, a system is modeled by a set of constraints over variables. Each variable contains several integer values in its domain and therefore it is called *Finite Domain Variable* (FDV). The FDV eventually can obtain an integer value that specifies a solution. The constraints are given as arithmetic expressions, equalities, inequalities and specialized combinatorial constraints. They define application's constraints (i.e., operation precedence constraints), resource constraints, specialized implementation constraints (e.g. pipelined resources) as well as general requirements such as performance or cost. The model is solved using constraint satisfaction/consistency techniques [14].

In Chapter 4 we have tried to express all kind of resource constraints by using dis-equations. In addition, in another contribution we treat all constraints uniformly and formally (see Chapter 8).

2.2 Scheduling Strategies

Scheduling data-path operations into control steps -time slots- is an important task. For obtaining an efficient design, a complete strategy of scheduling must consider both timing and resources constraints as well as storage and interconnection costs. Furthermore, for a specific systems it also must consider power consumption constraints.

As defined in the HLS design flow, there are three dimensions along which scheduling algorithms may differ:

1. the *objective function* and *constraints* that algorithms consider;
2. the *interaction* between scheduling and allocation;
3. the *resolution technique* that the scheduling algorithm used.

According to this features many tentative taxonomy of scheduling algorithms can be imagined. Scheduling algorithms can be broadly classified into *time-constrained* and *resources-constrained* scheduling, based on the goal of the scheduling problem as done by Govindarajan [52]. In time-constrained scheduling –also called as fixed-control-step approach– the functional units (resources) number is minimized for a fixed number of control steps. The video processing and the realtime applications are the main area for such algorithms. While in resource-constrained scheduling the control-steps number is minimized for a given design cost (number of functional and storage units).

Šilc [107] defines another taxonomy in which he classifies such algorithms into *transformational* or *iterative/constructive* algorithms.

A transformational algorithms start with an initial schedule (e.g. maximally serial or maximally parallel) and applies transformations to it in order to obtain other schedules. These

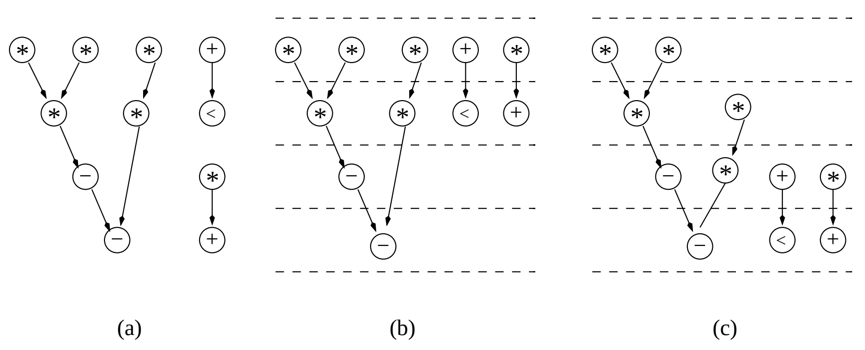


Figure 2.1: (a) Data Flow Graph (b) ASAP Schedule (c) ALAP Schedule

algorithms differ in how transformations are done, they can use *exhaustive* search, *branch-and-bound* technique or some *heuristics*. The other type of algorithms kind, the iterative/constructive ones, build up a schedule by adding operations one at a time till all the operations have been scheduled. These algorithms differ in how the next operation to be scheduled is chosen and into which control step it is put.

Several techniques have been experimented then integrated in both academic and commercial HLS systems, in the following survey of the constrained scheduling algorithms, we try to classify the scheduling algorithms using the **type of technique** that they use, starting from the most basic scheduling techniques as list scheduling and its variants [25, 32], to exact scheduling techniques from optimization area like integer linear programming [99] while reporting some miscellaneous techniques such as genetic [51] and simulated annealing techniques [92].

First, let us give some common definitions. Let $G = (V, A)$ be a data flow graph, where V is the set of operations to be scheduled, and A is the set of dependences. Let $n = |V|$ and $m = |A|$. Each operation is labeled by o_i, $1 \leq i \leq n$. A precedence relation between operations o_i and o_j is denoted by $o_i \prec o_j$, where o_i is immediate predecessor of o_j. There are q types of resources available. A function unit of type r is denoted by F_r. A relation between operation o_i and a resource F_r is denoted by $o_i \in F_r$, if F_r can perform o_i.

2.2.1 Basic Algorithms

In what follows, we will briefly describe the principle of some known scheduling algorithms, which take into account only data dependences constraints but no resource constraints. The schedules computed by these techniques represent the earliest and the latest bounds within operations in the DFG. Most constrained-scheduling algorithms that will be described later require these bounds.

2.2.1.1 ASAP

A simple scheme is to schedule operations "As Soon As Possible". The ASAP algorithm starts with the highest nodes (that have no parents) in the DFG and assigns time steps in increasing order as it proceeds downwards. It follows the simple rule that a successor node can be execute only after its parents has executed. This algorithm clearly gives the fastest schedule possible. In other words, it schedules in least number of control steps but never takes into account the resources constraints. Figure-2.1(b) shows ASAP schedule for the DFG in Figure-2.1(a).

2.2.1.2 ALAP

This approach is a refinement of the ASAP scheduling concept conditional postponement. The postponement occurs whenever the operations concurrency is higher than the number of available functional units. The ALAP algorithm “As Late As Possible” works exactly in the same way as ASAP algorithm except that it starts at the bottom of the DFG and proceeds upwards. This algorithm gives the slowest possible schedule for tasks. However, this doesn't necessarily reduce the number of functional units used. Figure-2.1(c) shows ASAP schedule for the DFG in figure-2.1(a).

The problem with ASAP and ALAP schedulers is that when there are limits on resource usage no priority is given to operations on critical paths. Hence, less critical operations can be scheduled first and thus block critical ones.

2.2.2 Critical Path Method

The minimum amount of cycles needed to schedule a basic block corresponds to the maximum depth of its data dependence graph. This longest-path is called the *critical path*. The ordering of the operations in the critical path is implied by their data dependencies.

Using ASAP and ALAP values, the critical path consists of those operations that are mapped to the same cycle in the early as well as in the late scheduling. The method takes three steps. First the critical path is computed. The operations from this critical path are scheduled. They represent a frame for adding the remaining operations. In the last step, the remaining operations are inserted into the critical path. This is done for each operation by testing the cycles between the early and late dates with respect to data dependencies and resources conflicts. This may lengthen the schedule.

The critical path method is not always able to create an optimal schedule. This is due to the fact that subsequent subframes cannot be merged together although this would be permitted by the data dependencies and the resources use.

2.2.3 List-scheduling

Continuing along the scale of increasing complexity, there are algorithms that use *list-scheduling* principle. List based scheduling is a generalization of the ASAP algorithm with the inclusion of resources constraints. Due to both its simple principle and low complexity, it is the most used approach. Indeed, it is used in many old HLS systems [19, 74, 88] as well in recently developed ones [37, 55].

The algorithm builds a constructive schedule, for each step, the operations available to be scheduled into that step are kept in a *list* - hence the name of list-scheduling - which is ordered by some *priority function*. Each operation on the list is scheduled if the resources it needs are still free in that step; otherwise it is differed to the next step. Scheduling an operation to a control step makes other successor operations ready, which will be added to the priority list.

There are many priority functions used. In the Slicer system [19] the priority function is based on increasing operation *mobility* [86]. The mobility of an operation is defined as the difference between ASAP and ALAP values of an operation. This would ensure that operations with large mobility are differed to later control steps because the number of control steps into which they could be is large.

Other systems like, ELF [48], use the operation *urgency* as priority function. The urgency of an operation is defined as the minimum number of control steps from the bottom that this operation can be scheduled before a timing constraint is violated.

More complex priority functions are developed, for instance, in the MAHA system [88], they use the information from the critical path. Indeed, the operations on critical paths are scheduled first (and a resource binding is done) after what the other operations are scheduled one at a time according to the least mobility. This is the same approach as in the critical path method. Indeed, the critical path method is a list scheduling method in which the priority function relies on belonging to the critical path.

The time and space complexity for this approach is slightly more because several lists have to be maintained dynamically. For this reason, Jain et al. [63] have designed a static list scheduling. This approach starts by creating a single large list before starting. It uses the ASAP and ALAP algorithms to obtain the Least and the Greatest possible Control Step assignments (LCS and GCS) for each operation. Then, the algorithm sorts all the operations in ascending order using the GCS labels as the primary key then sorts each operation set with the same GCS labels, in descending order with the LCS labels as the secondary key.

As it can be seen, the list-scheduler success depends mainly on the priority function used.

2.2.4 Force-Directed Scheduling (FDS)

Based on probability distribution technique, it is a very popular scheduling heuristic [89]. It is originally designed for time-constrained scheduling. Thus, the main goal is to reduce the total number of resources used. This algorithm achieves its goal by uniformly distributing the operations of the same type over the available control steps. By balancing the concurrency of operations, it ensures that each structural unit has high utilization which in turn decreases the total number of units required.

A simple outline of this algorithm :

1. **Determine time frame:** this step consists of determining the time frames $[S_i, L_i]$ of each operation $o_i \in V$, where S_i and L_i are obtained by the ASAP and ALAP algorithms. Let $p_{i\tau}$ denote the probability that o_i will be scheduled into control step $\tau \in [S_i, L_i]$. A useful heuristic is to assume a uniform probability distribution i.e., $p_{i\tau} = \frac{1}{1+L_i-S_i}$;

2. **Create distribution graph:** The next step is to create a distribution graph, by adding the probabilities of each type of operation r for each control step τ. The resulting distribution graphs indicate the concurrency of similar operation. $P(r,\tau) = \sum_{o_i \in F_r} p_{i\tau}$, where the sum is over all operations of a given type;

3. **Force calculation:** The final step is to calculate the force $\mathcal{F}$ associated with every feasible step τ assignment of each operation o_i. It temporarily reduces the operation's time frame to the selected step τ:

$$\mathcal{F}(o_i, \tau) = P(r,\tau) - \sum_{t=S_i}^{L_i} \frac{p_{r,t}}{1 + L_i - S_i}$$

   where r is the type of the operation o_i. In other words, the force associated with the tentative assignment of an operation o_i to step τ is equal to the difference between the distribution value in that step and the average of the distribution values for the steps bounded by the operation's initial time frame. The forces for all predecessors and successors (indirect forces) of the current operation must be calculated. The total force is the sum of the direct and indirect forces.

4. **Schedule** once all the forces are calculated, the operation-control step pair with largest negative force (or least positive force) is scheduled. Then P and $\mathcal{F}$ values are updated and the entire process is reitered until all operations are scheduled.

The complexity of the FDS algorithm is $O(c.n^2)$ where c is the control-steps number and n the operations number.

Later the authors generalized the approach [90] to treat many other problems. Among these problems: scheduling under resource constraints, minimizing storage and interconnection cost. Indeed, the force-directed list scheduling (FDLS) algorithm that they designed combines the characteristics and strengths of the list scheduling algorithm where force is used as a priority function. Another improvement has been done by Verhaegh et al. [105], they have changed the FDS strategy by pruning one control step from its mobility range and postponing the decision to a later stage.

The force-directed scheduling algorithm never backtracks on its decisions and hence is classified under constructive algorithms.

2.2.5 Path-Based Scheduling

Another commonly used algorithm is the path-based scheduling (PBS) designed by Camponaso [23] and used in different context by many HLS systems [16, 15]. The PBS is a control-flow-based algorithm which focuses on exploiting the control dependencies among operations. It analyzes all paths in the CFG and schedules each path independently, thus minimizing the number of control steps in each path. Paths in the CFG arise from conditional operations and loops.

The scheduling is based on solving constraints on the paths. These constraints are general restrictions on the schedule, and may be due to resources, delays, or any other cost measure. Note that the PBS algorithm is not concerned about parallelism or chaining. The only elements in the algorithm formulation are the control-flow paths and the constraints.

Constraints are represented as intervals in the control-flow paths and the problem of finding the minimum number of control steps in each path is the same as finding the minimum number of cuts crossing all constraint intervals in all paths which is done using exact clique-covering techniques. The minimum number of control steps is obtained for each path, not just the critical path as in Data-flow-based schedulers. Minimizing the number of control steps per path may not result in the overall minimum number of states.

In this approach, constraints formulation is completely general. In fact, it can model resource limitations, delay targets, or any other restrictions that should cause two operations to be scheduled in different control steps. These restrictions are mapped onto the CFG as constraint intervals and treated in exactly the same way by the scheduler.

Furthermore, the path-based approach can determine mutual exclusion (whether two operations can share the same hardware resource) in a general way by looking at the control paths and at the actual conditions controlling the execution of the operations.

However, the PBS has some problems: Firstly, its complexity is proportional to the number of control paths, which can grow exponentially with the number of conditional operations and this complexity became clearly unacceptable for large designs. Secondly, fixing the execution order limits parallelism; another limitation of PBS is that it does not change the order of the operations in a path. The constraint intervals are created for a fixed ordering of operations (usually the same order as in the input language description), which the scheduler is not allowed to change. In addition, optimizing each path independently may increase hardware; indeed, in order to optimize the number of control steps in each path, path-based scheduling (and subsequent allocation) may

have to create a more complex FSM and possibly more multiplexers and control signals which may result in larger area.
To avoid some of these disadvantages, Bergamaschi et al. have designed an adaptative scheduling algorithm in which they combine data-flow and control-flow techniques by instrumenting the path-based algorithm [17]. Their improvements consist in 1) reordering operations in the CFG in order to increase parallelism and maximize constraint overlapping 2) reducing the number of control paths by collapsing all conditional branches when it is possible 3) If the number of paths is still too large (after collapsing), it applies a control partitioning algorithm which reduces the number of paths by partitioning the CFG.

All the above schemes are constructive algorithms in the sense where the selection and fixing of operations in the control step occurs one by one until all the operations are fixed. Due to lack of a look-ahead scheme and the lack of compromises between early and late decisions, these constructive algorithms do not guarantee the solution quality. One can cope with this weakness by iteratively rescheduling some of the operations in a given schedule. For example, Park and Kyung [87] proposed an approach which is based on the paradigm originally proposed for the graph-bisection problem. In this near optimal approach, an initial schedule is obtained using any scheduling algorithm. At each iteration, a new schedule is obtained by rescheduling a sequence of operations that maximally reduces the schedule cost. If no improvement is attainable, the process halts.

2.2.6 Exact Algorithms

Previous algorithms are heuristic methods, to get a globally optimal solution one can rely on some exact search techniques as Integer Linear Programming [99] or techniques based on constraint programming paradigm [14].

2.2.6.1 Integer Linear Programming

Some of the best known exact scheduling techniques are based on Integer Linear Programming (ILP) models. According to HLS system contexts and the aimed objectives, many ILP formulations are proposed in the literature [72, 61, 47, 71, 27, 112, 65].

In a first attempt, Lee et al. [72], in their ILPS system, using an ILP formulation, tried to find an optimal schedule via a branch-and-bound search algorithm. This algorithm involves some amount of backtracking. They aimed a time-constrained algorithm. First the algorithm calculates the *mobility range* for each operation $Mob_{o_i} = \{Step_j | \ S_i \leq j \leq L_i\}$, where S_i and L_i are the ASAP and ALAP values respectively.

Let C_r be the cost of a resource of type r and M_r be an integer variable denoting the number of resources of type r. Finally, let $x_{i\tau}$ be 0/1 binary variable where $x_{i\tau} = 1$ if o_i is scheduled into control step τ; otherwise, $x_{i\tau} = 0$. Assuming a one-cycle propagation delay for each operation and a non-pipelined execution, the feasible scheduling problem can be stated as follows:

$$\sum_{o_i \in F_r} x_{i\tau} \leq M_r \qquad 1 \leq \tau \leq s, 1 \leq r \leq q; \tag{2.1}$$

$$\sum_{\tau=S_i}^{L_i} x_{i\tau} = 1 \qquad 1 \leq i \leq n; \tag{2.2}$$

$$(\sum_{\tau=S_i}^{L_i} \tau * x_{i\tau} - \sum_{\tau=S_j}^{L_j} \tau * x_{j\tau}) \leq -1 \qquad \forall o_i \prec o_j. \tag{2.3}$$

The first constraints state that any schedule should respect the resource constraints i.e., no schedule should have a control step containing more than M_r resources of type r. The second constraints guarantee that each operation o_i is executed once between S_i and L_i. The third constraint ensures that precedence constraints of the data flow graph are preserved. In this formulation M_r are unknown and the whole number of control steps C_{step} is fixed. Thus, the only objective function is $min \sum_{r=1}^{q} C_r * M_r$.

Another feasible scheduling problem using ILP has been formulated by Hwang et al. [61]. They aimed both *resource* and *time* constrained algorithm.

They use the same formulation as the above one but the objective function that they defined is a combination of time-constraint objective function $min \sum_{r=1}^{q} c_r * M_r$ and resource-constraint objective function $min\ C_{step}$. This approach allows user to control the resource-time trade-off. The ILP approach has made HLS problems better understood because the strength of ILP formalism is that it can express in a uniform way any kind of constraints and system requirements. Indeed more structured and detailed, but more complex, ILP-formulations are proposed by Gebotys et al. [47] and Zhang [117] in which they resolved the combined resource-constrained scheduling and register allocation problem. However, the algorithm execution time grows exponentially with the problem size represented by both the number of variables and the number of inequalities.

In practice, the ILP approach is applicable only to small problems. In addition, general ILPs may be difficult to solve due to weekness of bounds and speed of the algorithm of resolution. Many facts make a general ILP problem as an $\mathcal{NP}$-complete one [104]. In fact, the simple formulation above increases rapidly, in term of number of unknown, with the number of control steps. Indeed for unit increase in the number of control steps we will have n additional binary variables. Furthermore the usage of the precedence constraints 2.3 entails dealing with large coefficients –values of τ– which generaly makes the ILP resolution process $\mathcal{NP}$-complete in the broad sense [58]. Thus, optimal solutions can be found - albeit at the cost of high compilation times. Nevertheless ILP-formulation can be tightened by more understanding the polyhedra theory [99]. This can be done via many facts:

- designing a structured formulation;
- identifying of redundant serial constraints;
- insertion of valid inequalities.

These facts are studied and integrated in SILP[2] tool designed by Zhang [117]. More improvements in execution times has been observed but this computation time remains high.

[2]SILP: Scheduling and Allocation with Integer Linear Programming.

Recently Kästner et al. [65] have investigated approximations based on relaxation of the integrality constraint principle [83]. Indeed for large input programs, they do the relaxation in hierarchical way to guarantee the sub-optimality of the solution.

2.2.6.2 Constraint Programming Technique

A newer technique than ILP, the Constraint Logic Programming (CLP) model offers exact mechanisms –constraints handling methods– to resolve general optimization problems [14].

The CLP has already been applied in the hardware design automation area, in particular in resources assignment and scheduling as well as in verification and test.

For instance, Kuchcinski [68] designs a model for solving both scheduling and resource assignment problems by using constraint handling methods provided by this paradigm. In this approach, a system is modeled by a set of *finite domain constraints* over variables. Finite domain constraints are used to specify different properties and restrictions imposed on the specified design. The model is solved by using constraint satisfaction/consistency techniques.

First, Let us introduce what is a constraint satisfaction problem (CSP) for finite domain constraints and then present the formulation of the digital system modeling in terms of these constraints. A CSP is a 3-tuple $S = (V, D, C)$ where:

$V = \{x_1, x_2, \ldots, x_n\}$ is a finite set of variables, also called Finite Domain Variables (FDV),

$D = \{D_1, D_2, \ldots, D_n\}$ is a finite set of domains, and

C is a set of constraints restricting the values that the variables can simultaneously take.

For each variable x_i , a finite set $D_i \in \mathcal{P}(\mathbb{Z})$ of possible values constitutes its domain, called a finite domain (FD). For example, the specification $T :: \{1..10\}$ defines FDV T, which can have values $1, 2, \ldots$ and 10 while the specification $R :: \{23, 56\}$ defines FDV R, which can have a value of either 23 or 56.

A constraint $c(x_1, x_2, \ldots, x_n) \in C$ between variables of V is a subset of the cartesian product $D_1 \times D_2 \times \ldots \times D_n$ that specifies which values of the variables are compatible with each other. In practice, the constraints are defined by equations, inequalities, global constraints, or programs. For example, an inequality $T_1 + D_1 \leq T2$ defines a constraint on three FDVs T_1, D_1 and T_2.

Each constraint can be in one of three states: *satisfied, not satisfied* or in a state that cannot yet determine whether the constraint is satisfied or not *"don't know state"*. If the constraint is in the "don't know state" the consistency enforcement for this constraint can be applied. A particular program that implements a consistency method is called a *propagator* since it propagates changes in FDVs to domains of all FDVs, involved in a given constraint, by narrowing their domains. Combinatorial constraints are usually implemented using several propagators that consider different aspects of their consistency. Baptiste [9] has involved the CLP by describing and evaluating new resource constraint propagation algorithms for several classes of scheduling problem.

A solution s to a CSP S, denoted by $S \models s$, is an assignment to all variables V, such that it satisfies all the constraints. There usually exists many solutions that satisfy the defined constraints. They have different qualities which are defined by related cost functions. In most design problems, we are interested in optimal solutions that minimize or maximize this cost function. An optimal solution s to a CSP S is a solution $S \models s$ that minimizes or maximizes a value v assigned to a selected variable x_i . The standard method to find a solution to a CSP is to systematically assign FDVs with values from their domains. After each assignment the consistency of all constraints that contain changed FDVs is carried out. The process finishes when each variable has a value. If during the assignment an empty domain for a FDV is detected the process fails and backtracking is initiated. This is usually implemented as a depth-first-search method and the optimization uses some kind of branch-and-bound algorithm.

In the case of constraint-driven scheduling, the CLP provides a tool to model uniformly any kind of constraints. First, all required FD variables have to be defined. Kuchcinski, for each operation, o_i, defines three FDVs, T_i , D_i and R_i which represent the operation start time, the operation's delay and the resource used for its execution, respectively. For instance the following constraints are modeled as:

- precedence constraints: for each $o_i \prec o_j$: **impose** the constraint $T_i + D_i \leq T_j$;
- resource sharing: As R_i specifies possible implementation resources for a given operation. The resource constraint prohibits simultaneous use of resources and can be specified using disjunctive constraints like:

  for each o_i and o_j using the same resource: **impose** $T_i + D_i \leq T_j \vee T_j + D_j \leq T_i \vee R_i \neq R_j$

In this way, all kind of constraints are modeled. Scheduling can try to optimize a given cost function such as the time steps number, resource cost, power consumption, register/memory usage or a combination of these. The cost function is defined by FDV that is constrained to represent a given cost. For example, a cost function can be defined as:

For each o_i: **impose** $E_i = T_i + D_i$ **impose** $\max(EndTime, [E_1, \ldots, E_n])$.

Minimization of the domain variable *EndTime* produces the shortest schedule satisfying given resource constraints. Optimal solutions are found using a method similar to a branch-and-bound algorithm. This method interactively applies the depth-first-search algorithm.

As in the ILP technique, for large problems the CLP requires usually a large amount of computation time to find the optimal solution since the search space becomes huge. Therefore, partial search methods are proposed which unlike depth-first-search do not search systematically through all possible solutions but backtrack earlier if certain conditions are fulfilled.

2.2.7 Miscellaneous Techniques

To cope with the gap between the very expensive techniques in term of computation time and the graph-based methods which are enable to give guarantees on the computed schedule quality, some scholars have proposed heuristics that use modern search techniques such as *simulated annealing, tabu* and *genetic* based algorithms.

2.2.7.1 Simulated Annealing Based Algorithm

Another type of transformational feasible scheduler can use the simulated annealing technique. Kirkpatrick et al. [66] give some idea about how to instrument this technique in optimization. Indeed, the simulated annealing technique can be used for combinatorial optimization problems specified by a finite set of configurations and a cost function defined on all the configurations. The algorithm randomly generates a new configuration which is then accepted or rejected according to a random acceptance rule governed by the parameter analogous to temperature in the physical annealing process [92].

For instance the algorithm of Badia et al. [7] starts on an initial configuration obtained by applying ASAP strategy. The *Cost* function evaluates how good a configuration is. It is defined as:

$$Cost(X) = \alpha\ Area(X) + \beta\ Time(X)$$

where $Area(X)$ is the estimated total area of the used resources and $Time(X)$ is the total execution time corresponding to the given configuration X. The tuning of the algorithm is performed by taking different values for α and β. For example, if $\alpha \ll \beta$ the algorithm is closer to resource-constrained scheduler (since solutions efficient in speed become more important) while $\alpha \gg \beta$ makes the algorithm more time-constrained.

At the beginning, a high temperature $T_{initial}$ is given in order to accept most new configurations even if they increase the cost. Given a configuration X, a new configuration Y is generated either by insertion or removal of a register, scheduling an operation to next or previous control step or by shrinking/expanding a control step. Although simulated annealing is robust, it requires long execution time.

2.2.7.2 Genetic Algorithm

Genetic algorithms, as optimization ones, have been early and widely used in HLS. Wehn and al. [82] and Heijligers et al. [57] have instrumented the genetic paradigm (GP) to resolve both scheduling and allocation problems. Dhodhi et al. [35] use also the GP to resolve the circuit area optimization problem. Yang et al. in [115] use a genetic algorithm and draw Pareto diagrams for scheduling under power constraints, the algorithm has been extended to address run-time scheduling on System-on-chips (SoCs) [116].

For instance, Heijligers et al. [57], given a data-flow graph (precedence constraints) and a resource constraints, instrument a list-scheduler into a genetic algorithm. Their idea is based on the following fact: the advantage of a list scheduler is that the schedules constructed always satisfy the precedence constraints and the resource constraints however, the disadvantage is the influence of the priority function on the quality of its results. To overcome this disadvantage a genetic algorithm can be used to search for a good priority function to direct a list scheduler. Inside their genetic algorithm, the encoding of a schedule consists of a permutation of operations which can be used as a priority list for the list scheduler. The completion time of the resulting schedule is used to calculate the fitness -function used to distinguish between better and worse individuals - of the individual.

Initially a population with individuals is constructed, each containing a random permutation. Schedules are constructed by decoding permutations into priority list and applying a list scheduling algorithm. The genetic algorithm selects individuals for re-combination using stochastic sampling with replacement. Using this strategy, fit individuals have higher probability to be selected than non-fit individuals. A lower bound of the completion time using the precedence relations and resources constraints information is calculated using a prediction method. The genetic algorithm stops if it meets the lower time bound or if the number of iteration is 100. More special attentions have been paid to improve the quality of the population for more details see [57].

Genetic algorithms are probabilistic search algorithms, thus the computed schedules are without guarantee that they reach the optimum.

2.3 High-Level Synthesis Tools

There are two types of HLS tool: data-flow oriented tools and control-flow oriented ones. Though the later tools are most general because they are able to deal with data-flows too.

Most existing HLS tools are specialized for a restricted class of applications. In the domain of signal processing (DSP), numerous architectural synthesis tools are described in a variety of publications. Among them, we quote MAHA [88], HAL [89], EASY [101], MIMOLA [75], CADDY [56], OSYS [62] and Phideo [106]. For more generic and flexible architectures, other tools are developed: Amical [67] and SPARK [55]. For more specialized architectures, like systolic architecture we quote MMAlpha [97]. In the field of compute-intensive embedded systems, we quote LooPo [38] and Syntol[3].

Many tools are developed in general for purely data flow designs, here, we just mention some recent ones while focusing on the structure of their schedulers.

2.3.1 GAUT

GAUT is a pipeline architectural synthesis tool dedicated to signal processing applications developed jointly by LESTER and LASTI laboratories at Lannion and Rennes universities respectively [74].

GAUT generates a structural and functional VHDL description of a dedicated architecture. The designer provides as input: a behavioral description of the circuit, a description of the library of operators, a maximal latency and a cycle frequency. The library models must have been assigned with their physical characteristics. GAUT uses these characteristics to compute the cost of the operator assignment. A generic library can be parametered by time and cost to become a technology driven library.

In addition, GAUT may consider synchronization constraints; hence the scheduler takes into account the order in which data are exchanged with the system and at what time this is possible.

GAUT's scheduler is a list scheduling algorithm with mobility as a priority function which also depends upon the availability of allocated operators. The operations are scheduled as soon as the operator is available. The optimal assignment of a candidate operation on an available operator responds to the minimization of interconnections between operators. The pipeline control of each operator is managed by a complementary priority on assignment. When an operator is allocated, but as yet not used, its use is primarily inferior to that of an operator already utilized. Furthermore, if a candidate operation has a positive mobility, then the scheduling is delayed. Finally, if an operator allocated from the beginning of the period is never used during the entire period, its allocation interval is delayed for one clock period.

GAUT is also a user guided synthesis tool and it instruments memory optimization techniques but it requires predefined timing characteristics and it uses Loop flattening.

2.3.2 SPARK

SPARK[4] is an HLS research tool developed at the university of California at Irvine. It takes a behavioral description in *ANSI-C* as input and produces synthetizable RTL VHDL.

Gupta et al. [55] use parallelizing compiler technology, as we do, developed previously to enhance instruction-level parallelism and re-instrument it for HLS by incorporating ideas of mutual exclusivity of operations, resource sharing and hardware cost models.

The intermediate representation used in *SPARK* consists of basic blocks encapsulated in Hierarchical Task Graphs (HTG). To exhibit more parallelism and allow finding a good schedule, they use a tool transformations toolbox. The techniques implemented are:

- *code motion (CM):* Two code motion techniques are used: *percolation* scheduling and *trail-blazing*. Percolation Scheduling (PS) was developed as a technique to target code to parallel

[3]An HLS research tool, currently develop by CompSys, an INRIA project, France, team at LIP laboratory `http://www.ens-lyon.fr/LIP/COMPSYS/`

[4]Available at `http://www.cecs.ici.edu/~spark`

architectures such as VLIW and vector processors. It compiles programs into parallel code by systematically applying semantic preserving transformations. These transformations have been proven to be complete with respect to the set of all possible local, dependency-preserving transformations on program trees. However, to move an operation from a node A to node B, percolation requires a visit to each node on every control path from A to B. The incremental nature of these linear operation moves causes a code explosion by unnecessarily duplicating operations and inserting copy operations. Trailblazing was proposed to circumvent these problems. Trailblazing is a code motion technique that exploits the hierarchical structuring of the input description's operations and global information in the HTG to make non-incremental operation moves without visiting every operation that is bypassed. At the lowest level, trailblazing is able to perform the same fine-grained transformations as percolation. However, at higher levels, trailblazing is able to move operations across large blocks of code.

- *dynamic renaming:* As it is well known there are four types of data dependencies: flow dependence(variable read after write), anti-dependence (write after read), output-dependence (write after write) and input-dependence (read after read). Only flow dependencies are important for the semantic preserving transformation since the other ones express memory reuse and can be discarded. Thus, non-flow dependencies that prevent code motion can often be resolved by dynamic renaming.
- *common sub-expression elimination (CSE):* this transformation attempts to detect repeating subexpressions in a piece of code, stores them in variables and reuses the variable whenever the sub-expression occurs subsequently.
- *speculative code motion:* Operations may be moved out of conditionals and executed speculatively, or operations before conditionals may be moved into subsequent conditional blocks and executed conditionally by reverse speculation, or an operation from after the conditional block may be duplicated up into preceding conditional branches and executed conditionally by conditional speculation. Operations can also be moved across entire hierarchical blocks, such as if-then-else blocks or loops.

SPARK's scheduler is also a priority-based list scheduling heuristic: the inputs to this heuristic are the unscheduled HTGs of the design and the resource constraints list. Additionally, the designer may specify a list of allowed code motions: speculation, conditional speculation, whether dynamic variable renaming is allowed, and the code motion technique (percolation or trailblazing) for moving the operations.

The heuristic starts by assigning a priority to each operation in the input description based on the length of the dependency chain of operations that depend on it. Scheduling is done one scheduling step at a time while traversing the basic blocks in the design's HTG. Within a basic block, each scheduling step corresponds to a statement HTG node. At each scheduling step in the basic block, for each resource in the resource list, a list of *available operations* is collected.

Available operations is a list of operations that can be scheduled on the given resource at the current scheduling step. Initially, all unscheduled operations that can be scheduled on the current resource type are added to the available operations list. Subsequently, operations whose data dependencies are not satisfied and cannot be satisfied by dynamic variable renaming, and operations that cannot be moved in the HTG to schedule them onto the current scheduling step using the allowed code motions, are removed from the available list. The remaining operations are assigned a cost based on the length of the dependency chain leading up to the operation.

The scheduling heuristic then picks the operation with the lowest cost from the available operations list. The trailblazing is then instructed to schedule this operation at the current scheduling step. This is repeated for all resources in each scheduling step in the HTG. Once the chosen operation has been scheduled, the dynamic CSE heuristic finds and eliminates common subexpressions in the operations in the available list, if the new position of the scheduled operation permits.

It is reported that in effect, this scheduler improves the performances of the final netlist and reduces by up to 50% the total delay through the circuit. However, despite all these sophisticated heuristics, no guarantee can be given on the quality of the computed schedule. This is mainly due to the list scheduling approach which performs only local choices.

The SPARK tool exploits the parallelism into and through basic blocks, but it doesn't consider the inherent parallelism through perfectly or imperfectly nested loops while such parallelism is widely present in many high-throughput digital signal processor applications.

2.3.3 Ugh

User Guided HLS (*Ugh*) is another HLS research tool. It is a part of the Disydent project[5]. This framework [6] is dedicated to SoC platform based design for shared memory Multiple Instructions Multiple Data (MIMD) architectures.

Ugh's designers introduced more interactions between the tool and the user. They search for a best solution in a space of solutions obtained by repeating a list-scheduling heuristic.

The first transformation done by Disydent aims at increasing the performance: the application must be parallelized and/or pipelined. Indeed, the Disydent approach advocates the MIMD solution using Kahn Process Networks modeling [64].

Ugh's inputs are a restricted C program, a Draft Data-Path (DDP) and a clock frequency. It produces both a synthetizable VHDL RTL model and a cycle accurate simulation model. Indeed, to guide the tool the designer must define a DDP which is a simplified structural description of the target data-path which respects coarsely the resource constraints. The DDP is a directed graph whose nodes are functional or memorization operators and whose arcs indicate the authorized dataflow among the nodes. Each C variable has an associated register in the DDP.

The synthesis process is split into 3 main steps: first, the coarse grain scheduling (CGS) is run, resulting in allocation and translation of C statements into RTL instructions, then the mapping is performed to get the physical data-path and temporal characteristics. Finally a fine grain scheduling (FGS) is run, resulting in the *rescheduling* of the RTL instructions taking as constraints the annotated timing delays of the previous data-path:

1. In CGS, coarse means that the operations are only partially ordered. The algorithm used in CGS must choose a DDP sub-graph for each C statement and then coarsely order them. These choices and this ordering are done by maximizing the intrinsic parallelism while trying to reduce the data-path area. The degrees of freedom for reducing the area are the minimization of the input numbers of the added multiplexers and the binding of operations of the same type. Its temporal constraints are: multipliers need 2 cycles, adders and subtracters need 1 cycle, and all other functional cells have negligible propagation times. This algorithm is a list scheduling algorithm [37]. It produces a coarse finite state machine.

[5]A Digital System Design Environment, an open source framework developed at the university Pierre et Marie Curie, Paris, France. Available at `http://www-asim.lip6.fr/recherche/disydent/`

2. After mapping, placing and routing, the generated circuit will probably not run at the expected frequency. The main reasons are that the FSM has been constructed with estimated operator and connection delays. Furthermore, it is also possible that the circuit does not run at all if it mixes short and long paths. This happens frequently in circuits having both registers and register files.

3. Afterwards, the user decides, if the synthesized cycle didn't respects all constraints, to perform the FGS by introducing some directives and resume the process (rescheduling) until an acceptable solution is found. The FGS adapts the coarse FSM to the characterized datapath to ensure that the circuit will run at the given frequency. FGS extracts the register transfer instructions from the coarse FSM and then reschedules them taking into account the propagation delays, the setup and hold times of the cells and the intrinsic parallelism supported by the data-path. This algorithm is once again based on list-scheduling.

Ugh is dedicated to control oriented applications, it allows multi-cycle operations, operator chaining and multi-functional operators. However, it requires very low-levels information given by the user and it is highly dependent on a commercial tool (Synopsys).

Same as in SPARK tool, Ugh does not take into account the parallelism in loop nests. It is satisfied to exploit the parallelism in the innermost loop by its unrolling. Another problem, is that no register optimization is performed.

In addition, the process (CGS - mapping - FGS) can take time to be resumed before an acceptable solution is found. Indeed the tool, in the rescheduling pass, uses the electrical characteristics of an old data-path to improve the features of the design; the new generated data-path after placing and routing may need more cycles, and thus the circuit is less efficient and need other passes.

2.3.4 MMAlpha

MMAlpha [97] is another open source research tool developed at Irisa laboratory of Rennes, France[6]. It is dedicated to highly pipelined accelerators applications. This tool does not handle resource constraints. We quote it only because it is an instance of tools which efficiently handle parallelism in loops without unrolling them.

Its kernel technology rely on polyhedral model to increase the input performance by exploiting the parallelism in loops. In the polyhedron model a loop nest is abstracted by the polyhedron described by the loop indices during execution of the loop. It can be used for any index-based structure : memory (arrays), communications (accesses).

MMAlpha uses systolic design methodology. Its input is a functional specification in Alpha language and its output is an RTL description of systolic-like architecture in Alpha or VHDL.

Alpha is a functional language for expressing regular algorithms, synthesizing regular architectures or compiling to sequential or parallel machines from a high level specification. An algorithm is described by equations involving variables defined on multi-dimensional domains (which are extracted from the polyhedral formulation). By successive transformations (uniformization, parallelization for instance), the description is refined until it may be interpreted as an architecture. Then, this description can be translated towards logic synthesis tools in order to generate a VLSI architecture. Alternatively, different analysis (scheduling, lifetime, etc.) may guide the transformations towards imperative loop code for general purpose (sequential or parallel) processors.

[6]`http://www.irisa.fr/cosi/ALPHA/`

However, MMAlpha by using the Alpha language and the systolic design methodology are not widely accepted as these concepts are very different from designer's habits.

2.4 Scheduling using Reservation Tables

The reservation table were originally used in scheduling by Lam [69] in software pipelining which is a scheduling technique for VLIW (Very Long Instruction Word) processors.

In our scheduling approach, we have instrumented this concept both for decreasing the complexity of the scheduling problem and for taking into account both peculiarities of the data path and special resource features, like pipelined units, bypassing, and other communication constraints.

This concept is also instrumented by Ly et al. [73] but with another terminology; they proposed the idea of " behavioral templates ". The term *template* was used in [96] to describe structural patterns to exploit regularity. Formally they defined a behavioral templates T, as a CDFG object which specifies a set of tuples, (n_i, o_i), where n_i is a CDFG node and o_i is an integer cycle *offset*. The semantic is that T imposes the constraints:

$$schedule(n_i) = schedule(T) + o_i \qquad \forall (n_i, o_i) \in T$$

where $schedule(n_i)$ and $schedule(T)$ denotes the schedules for n_i and T respectively.
Extracting templates is done by pattern matching on the CDFG nodes. A template locks a number of operations into a relative schedule with respect to one another. This fact allows easly 1) handling time constraints, 2) sequential operation modeling, 3) pre-chaining of certain operations, and 4) hierarchical scheduling. They organize CDFG nodes into super nodes (templates) and schedule them. Their hierarchical scheduler is based on a list-scheduling algorithm in which task priorities are deduced from resource-free ASAP and ALAP schedules.

2.5 Hierarchical basis Scheduling Frameworks

In the literature most scheduling algorithms, designed for HLS, compute a schedule into *one pass*. The sample of algorithms chosen and presented in this chapter are among them. Many factors let us think that this approach is not the best one. It is due to the large number of constraints and objectives to be satisfied. Indeed, the application size, the diverse nature of constraints, all peculiarities of the data path, the requirements of the possible implementation could not be taken into account in one pass to get an optimal schedule without combinatorial explosion.

In order to cope with these problems and reduce them to manageable sizes one can propose stepwise scheduling algorithms. Such approaches can be justified by another reason. Indeed, in practice and contrary to a massive parallel system, embedded systems (known as very constrained systems) do not always require the generation of an optimal solution if the solution obtained is close to an optimal one.

In the HLS literature there are few frameworks that have used stepwise scheduling. Verhaegh et al. [106], for high-throughput applications, has designed a HLS tool –PHIDEO– which uses a two-stages scheduling algorithm. In fact, to exploit the parallelism through nested loops that contain operations using multi-dimensional arrays, they introduce a model of multidimensional periodic operations. In this model, operations are executed repeatedly with several dimensions of repetition, each of which corresponds to one loop. A specific execution of an operation can be identified by the corresponding values of the loop iterators. The time at which such an execution takes place is explicitly given in the model by means of the operation's period vector, whose components denote the time between two consecutive iterations in each dimension of repetition, and its start time, which denotes the time of the first execution of the operation.

In their multi-dimensional periodic scheduling problem, they have to determine the operation's period vectors and start times and they have to assign the operations to resources on which they are executed.

Due to the high throughput, severe timing constraints, and high memory requirements, it is of utmost importance to choose the period vectors and start times such that a highly parallel implementation is obtained in which the original loops are executed concurrently.

They take into account three sets of scheduling constraints: 1) timing constraints, which bound the period vectors and start times of the operations, 2) resource constraints and 3) precedence constraints, The scheduling objective they consider is to minimize the area occupied by the hardware. In video applications, area is not only determined by resources, but also by the memories that are used. So, a tradeoff has to be made between resources and memory.

In the first stage, they designed an algorithm to assign periods such that storage costs are minimized. To this end, they use a branch-and-bound approach based on linear programming and constraint-generation techniques and using an approximate cost function. For the second stage, they use an iterative algorithm to assign start times to operations and to assign operations to resources based on graph coloring. For both stages, they use integer linear programming techniques. It is reported that this hierarchical scheduling approach has considerably reduced the complexity of the scheduling problem.

2.6 Conclusion

In the literature, many HLS scheduling algorithms are developed. Most approaches belong to the family of priority-list scheduling algorithms, differentiated by the way in which task priorities are assigned, they can be sufficient for less constraints embedded systems. At the moment, these graph-based methods are the only way to schedule large programs within an acceptable time. Nevertheless, they don't give any guarantee on the quality of the solution.

On another hand, there are exact methods which are generaly based on ILP techniques. By construction, they guarantee the optimality of the solution, but in practice these techniques are applicable only to small problems.

To fill the gap between these two approaches, we believe that the quality of the schedule could be improved by *integrating exact methods* into *hierarchical scheduling approaches*, that could identify code fragments –with reasonable sizes– and schedule them.

In addition, there are other reasons for which the actual HLS scheduling algorithms are not mature. Indeed, very quiet interest to exploit the parallelism in nested loops; despite that many embedded application, specially in high-throughput DSP and compute-intensive embedded systems, contain nested loops and multidimensional arrays, describing repetitive executions of operations and repetitive production and consumption of data.

Chapter 3

General Scheduling Approaches

As has been shown in the introductory chapter, we have investigated on hierarchical scheduling approaches. Many factors have influenced these choices. In this chapter, we give an overview of our HLS tool and we explain both scheduling strategies that we have designed and discuss our general motivations to design such stepwise schedulers.

3.1 *Syntol* Project

The schedulers we describe in this thesis are part of the *Syntol* tool. It is an HLS research tool, that we currently develop with the CompSys[7] team at LIP laboratory[8]. Its aim is HLS in the field of compute-intensive embedded systems. The starting specification is a variant of C, including loops, the output is a hardware description at the RT level.

3.1.1 Input Specification: CRP Specification Language

Different languages have been used as input to HLS. Hardware Description Languages (HDL), such as Verilog-HDL and VHDL, are the most commonly used. However, designers often write system-level models using imperative programming languages, such as C/C++, to estimate the system performance and verify the functional correctness of the design. Using such languages offers higher level of abstraction, fast simulation as well as the possibility of leveraging a vast amount of legacy code and libraries, which make easy the task of system modeling. In addition, imperative languages contain many features which are not present in adapt subsets of HDLs for example data abstraction, dynamic use of memory.... However, the use of all or a subset of an imperative language to describe hardware is a less mature topic. Indeed, to easily map input description to hardware we need some language features present in HDL but not present in software languages. *Concurrency* is the most important one. For instance, it is allowed by means "*Always blocks*" in Verilog HDL and *"Process"* in VHDL. Indeed, hardware is inherently parallel, while imperative programs are inherently sequential.

Despite, the recent initiative SystemC [10] which is an attempt to standardize the C/C++ based language for both hardware and software, describing hardware with the present status of imperative languages, remains weak. Thus the notion of processes which encapsulate programs that execute concurrently, have to be introduced. This notion allows to describe a system as a network of processes. This is done at the partitioning step, which is usually manually done

[7] An INRIA project, France, `http://www.ens-lyon.fr/LIP/COMPSYS/`

[8] `http://www.ens-lyon.fr/LIP/COMPSYS/`

by the designer. By the way, in *Syntol* tool, we do the same. Indeed we use a special model *CRP* (Communicating Regular Process) to describe systems [41]. This model is inspired from the Kahn Process Network KPN model [64].

CRP model has been designed mainly to allow the decomposition of large application into small modules. The aim of this formalism is to express parallelism in an easy way, allowing more modularity and scalability of scheduling and promising reuse and readability. These modules - in fact processes- communicate through channels; the resulting system allows a visual representation and looks familiar to electronic designers (see Figure 3.1. CRP is not a programming language but a specification language. It can be seen as a variant of C augmented with "process" and "channel" constructors with the following semantic:

Process

A process is a sequential program which can communicate with other processes through channels. With the exception of channels, all variables are local to one and only one process and are not visible from other processes. The code of a process can be written in any convenient algorithmic language. We use C here, but other choices are possible: Pascal, Fortran and others.

The code of a process is regular, or has static control in the following sense:

- Statements are assignment statements and bounded loop statements. All variables are considered part of some array, scalars being zero-dimensional arrays.
- Loops are of the arithmetics progression variety (exactly the for loops of Pascal), and the loop upper and lower bounds are affine forms in numerical or symbolic constants and surrounding loop counters.
- The only method of address calculation is subscripting into arrays of arbitrary dimension. The subscripts must be affine forms in constants and surrounding loop counters.

Some of these restrictions are quite natural when one is designing compute-intensive embedded systems with real time constraints. It is difficult, for instance, to predict the execution time of a while loop or of the traversal of a truly dynamic data structure. In addition, other restrictions can be lifted by preprocessing (goto removal, inductive variable detection, subscript-like pointer detection, function inlining).

The iteration vector of a statement is a list of its surrounding loop counters, from outside inward. An iteration vector for S cannot take arbitrary values. It must belong to the iteration domain of S, which is obtained by stating that each counter is within the bounds of the corresponding loop. Under the assumption that the program is regular; iterations domains are sets of integral points inside polyhedron. Let D_S be the iteration domain of statement S. An iteration of S or operation is written $\langle S, x\rangle$, $x \in D_S$ where x is the iteration vector. The set of operations of a process P is the disjoint union:

$$E_P = \bigcup_{S \in P} \{\langle S, x\rangle |\ x \in D_S\}$$

and the set of operation of a process system is $\cup pEp$. In what follow, we may simply write $u \in E$ for an arbitrary operation.

Channels

A channel is an array of arbitrary dimension which is used as a communication medium from one process to another. Channels are unidirectional. One process is declared as the writer to a channel. Considered as an array, each cell of the channel must be written only once by its writer: this is the single assignment property. Writing to a channel is non-blocking.

A channel may have any number of readers. Reading is not destructive: a value remains in a channel at least as long as some process may have some use for it. If a process reads a cell which has not yet been defined, it blocks until a definition happens.

$W(A)$ denotes the set of operations that write into channel A with subscript function ω_A , and $R(A)$ denotes the set of operations that read from A with subscript function ρ_A . Clearly, $W(A) \subseteq E$ and $R(A) \subseteq E$. The set:

$$F(A) = \{\omega_A(u) | u \in W(A)\}$$

is the footprint of A. If the following constraint:

$$G(A) = \{\rho_A(u) | u \in R(A)\} \subseteq \ F(A)$$

is not satisfied, it is clear that some process will block for ever when accessing a memory cell in $G(A) - F(A)$.

An Example

To illustrate this model, let's see the following trivial example. It specifies a system where the process `producer` generates values which are consumed by the process `consumer`. We call this illustrative system a `Pipeline`.

```
int n;
process producer(outport int x[]){    process consumer(inport int y[]){
 int i;                                 int i;
 int t;                                 int z;

 lb1:  t = 1;                                for(i=0; i<n; i++){
      for(i=0; i<n; i++){               R:    z = y[i];
 K:       t = (t + i) >> 1;                       }
 W:       x[i] = t;                    }
      }
}
                 /* the glue code  */

                    void main(){
                         channel int a[];

                    lb2:      n = 100;
                    P:        producer(a);
                    Q:        consumer(a);

                    }
```

As we have mentioned above, Figure 3.1 which diagrams this example, is well familiar to electronic designers.

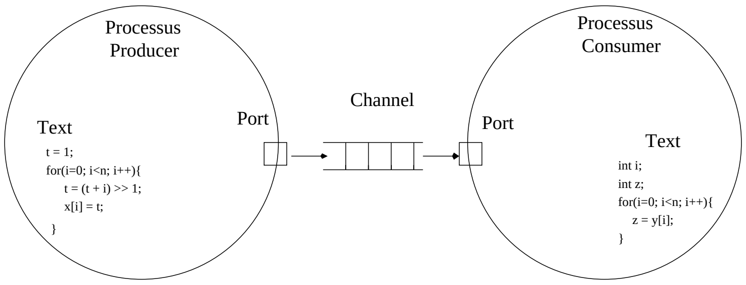


Figure 3.1: Pipeline System.

The new keywords *process, inport, outport and channel* are self-explanatory. Technically, they appear as new storage specifiers in the C grammar. In the glue code, one starts a process with the same syntax as for a function invocation. However, the process call returns immediately.

3.1.2 Target Architecture: the RTL Formalism

In order to define our HLS tool we first define the processor model which is used to express the target structural description of the behavioral input program. In fact, we use a description at *Register Transfer Level* [8] (RTL); the most popular and the most standardized model to describe embedded systems [45]. Such model consists of a controller -typically described by a finite state machine- and a Datapath.

As shown in Figure 3.2, the model has two types of I/O ports: 1/data ports, which are used by the outside environment to send and receive data to and from the model, 2/control ports, which are used by the outside environment to receive the information about the status of the system and to send the information about the status of the environment

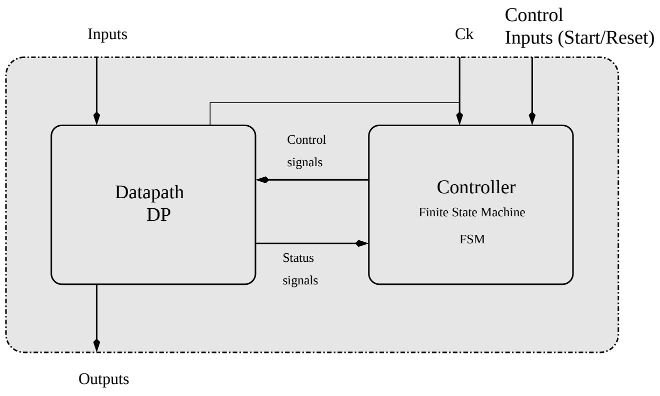


Figure 3.2: Target of HLS - RTL processor-

The datapath consists of storage units such as registers, register files, memories, combinatorial units such as ALUs, multipliers, shifters and comparators. These units and their I/O ports are connected by wires and buses. The datapath takes the operand from storage units or input ports, performs the computation in the combinatorial units, and returns the results to storage units or output ports during each state, which is usually equal to one clock cycle.

The selection of operands, operations, and the destination of the result are controlled by the control unit by setting proper values of the datapath control signals. The datapath also indicates, through the status signals, when a particular value is stored in a particular storage unit or when a particular relation between two data values stored in the datapath is satisfied.

A controller consists of a state register and next-state and output logic. Next-state logic generates the value of the state register in the next step -typically the next clock cycle- while output logic generates the value of control and external signals.

3.2 A Schedule the Main Tool to Get an FSMD

In our HLS tool, we get the FSMD of a given application specified as a system of communicating processes, by means of a *valid schedule* of its operations. In Contrast with earlier HLS systems where an FSMD is synthesized by scheduling the CDFG which is simply the sequential flow diagram of the input description [45]. Indeed, it is a quite different way, as we regenerate an equivalent program which exhibits more parallelism; So the first step of this conversion is the construction of a *schedule*, which gives the epoch at which each operation/instruction in the program is executed. The problem of regenerating an equivalent program from a valid schedule has been first studied by Irigoin [5] and a very efficient solutions, with associated software, are available today.

The main purpose of regenerating an equivalent code is to extract parallelism through the loops of the input description. In fact, to perform this, we use a scheduling strategy previously used for automatic loop parallelization [40, 41]. This technique which has already proved its efficiency:

- assigns a *logic "date"* – so that the schedule is considered as just a way of specifying an execution order – to each statement in the C program. The result of this pass is the definition of a sequence a sequence of logical steps (fronts) where each step is a group of operations to be executed in this logical step. Typically, a front is a pool of a few data-independent (i.e., parallel) loop iterations, each iteration consisting of several statements (in general parallel too, but not necessarily). Classical loop parallelization algorithms [40, 32] generate maximal parallelism expressed as parallel loops (i.e., large parallel fronts with no resource constraints); our algorithm is a variant that can generate -currently, in a heuristic way- bounded fronts if limited parallelism is desired. This is a form of symbolic loop unrolling or tiling. This technique will be detailed in the following section.
- when a schedule is computed, it allows us to rewrite the code into a form -parallel code- with explicit parallelism (see details in Section 3.5). From this equivalent parallel code one can build a finite state machine in which the time is measured in logical steps.

A short review of the generating method will be given below.

3.3 Stepwise Scheduling

In the symbolic scheduling, we take into account all data dependencies. Other constraints also can be treated such as delays of the elementary operations. However, the delays of operations are first approximation to the real delays as we consider, for instance, that a multiplication takes twice the time of an addition. It is important to emphasize that in this first scheduling step, all the operations in the nested loops are taken into account.

However, this symbolic scheduling technique is quite complex and cannot take into account all the micro-operations - and the architectural resources they need- that are implied in the execution of one high-level statement. For instance a C statement as:

```
        R:                         y = a[i+2] * t
```

in the symbolic scheduling technique is considered as an *atomic statement* so the schedule gives it one logical date. However in hardware R is considered as a macro-task because it is composed at least by a set of three actual elementary operations: 1) a subscript/address calculation, 2) one multiplication and 3) an access to the block memory assigned to the array a if we assume that it is mapped to a block memory. So this is still a too *coarse* description for hardware generation and we must provide separate micro-operations for subscript calculations, memory management, and functional units use. Thus, for a given logical step, each statement is a complex sequence of micro-instructions that we call micro-operations (for the sake of simplicity we use operations). Furthermore, into a statement, the operations may be linked by data dependences constraints.

Due to this particular construction, the macro-tasks in a front are data independent but they may still interfere in their use of resources. So, the front -logical step- must then be “split” into as few elementary steps as necessary to satisfy resource constraints. In other words, we need at least another step to schedule locally all operations of the macro-tasks belonging to the same logical step to satisfy the resource constraints and data dependencies between the elementary operations of the same macro-task.

> So these considerations lead us to the idea of designing **gradual approach** to scheduling programs with loops down to RTL.

Many factors and reasons let us to think that such approach can improve a lot the performance of the scheduling algorithm. First, let us notice that the application size, the diverse nature of constraints, peculiarities of the datapath - features of the resources- and the requirements of the possible implementation can not be taken into account into one pass to get an optimal schedule without exploding.

In addition, let us mentioned that our symbolic scheduling technique is based on integer linear programs. For a large application, finding legal schedules entails solving large linear programs. Thus including the amount of all elementary operations at the first scheduling level, which aim is extraction of parallelism in loops, would greatly increase its complexity, and would not improve the result significantly.

Besides, even when it is possible to consider all details of micro-operations in the symbolic scheduling step, a tight packing of these micro-operations also has the desirable result of minimizing the number of intermediate values to be stored in registers and such a property is hard to ensure with loop parallelization techniques.

Let us recall that from the result of the first scheduling step, we build an FSMD in which time is measured in logical steps. Each state contains a set of data independent high-level statements (macro-tasks) and each of them is composed by a sequence of operations. Now, we must refine this logical step to satisfy detailed resource constraints and taking into account all peculiarities of the datapath. Time will be measured in physical time i.e., clock cycle of the target architecture. How to perform this? In this thesis, we have sketched two hierarchical scheduling approaches.

3.3.1 Two-step Approach

After the first scheduling step, one possibility is to consider all the micro-instructions of a front and schedule them simultaneously while considering resource constraints. This is more natural possibility; if we choose an exact method we reach an actual optimal solution.

3.3.2 Three-step Approach

However, again for complexity reasons one can alternatively consider another possibility. Indeed we consider that, in our context, it will be good enough to decouple the problem into two subproblems. This partitioning can be sketched as:

- After the symbolic scheduling, we first schedule each macro-task *independently*, taking into account all peculiarities of the data path and resources, like pipelined units, bypassing, and other communication constraints. The schedule of each macro-task is summarized by a *reservation table* that states which resources at which cycle (relative to the starting time of the macro-task) are used by this macro-task. We call this second step *micro-scheduling.*
- Due this particular construction, the macro-tasks in a front are data independent but they may still interfere in their use of resources. So we need a another scheduling step to satisfy resource constraints. We call this third step *fine-grain scheduling.*

Let us consider the following fragment:

```
 for(i=0;;i++){
Z: s[0] = 0;
   for(j=1;j<7;j++)
M:   s[j]=s[j-1]+e[i+j-1]*w[j];
W: o[i] = s[6];
 }
```

This program represents the application of a six-taps FIR filter to input `e` giving output `o`. Dependence analysis and symbolic scheduling show that this program has the following causal schedule:

$$\theta(Z,i) = 2i \tag{3.1}$$
$$\theta(M,i,j) = 2i+j \tag{3.2}$$
$$\theta(W,i) = 2i+7 \tag{3.3}$$

For HLS, one has first to infer from these results exactly what happens at each clock cycle, i.e., to solve equations of the form $\theta(U,\vec{i}) = t$, where t is a time variable. It is clear that the cases t even and t odd are to be treated separately. The result is that for t even, one will execute statement Z for $i = t/2$, statement M for $t/2-3 \leq i \leq t/2-1$ and $j = t-2i$, and that statement W is not executed. The step at logical time t is thus made of one instance of statement Z and three instances of statement M. A similar result holds when t is odd.

The detailed timing of statement M depends on the amount of hardware we intend to devote to its execution. For the sake of definiteness, let us assume circular buffers for `e` and `o`, a register file for `s` and a ROM for `w`, and a floating point adder and multiplier, both implemented as three stage pipelines. It is easy to see that for each instance of M, the value of j is fixed ($j = 2,4,6$ for the even steps) hence the only address calculation is that of `e[i+j-1]` which can be simplified by strength reduction. A reservation table for an instance of M is:

step	1	2	3	4	5	6	7	8	9
address	x								
e buffer		x							
s file					x				x
ROM		x							
mult			x	–	–				
add						x	–	–	

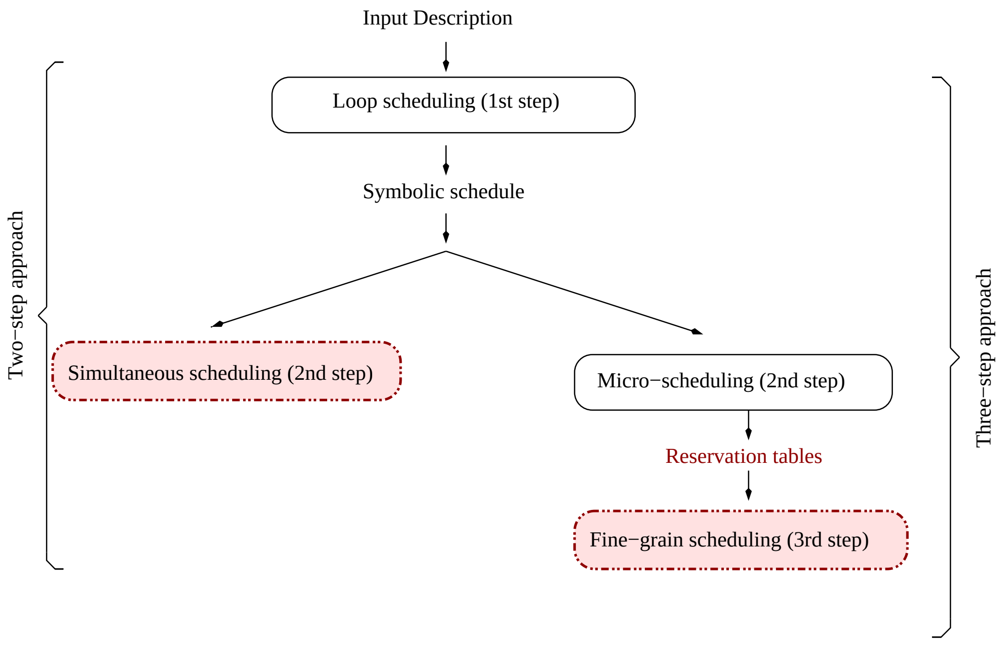

Input Description
Loop scheduling (1st step)
Symbolic schedule
Simultaneous scheduling (2nd step)
Micro−scheduling (2nd step)
Reservation tables
Fine−grain scheduling (3rd step)
Two−step approach
Three−step approach

Figure 3.3: The Hierarchical Scheduling Views

where a "–" indicates that the corresponding pipelined operator is busy but may accept further operands. This reservation table has been compacted as much as possible. Firstly, in this way M is executed as fast as possible; secondly, any other implementation would implies additional registers to hold intermediate results.

We now have to schedule three copies of this table plus the simpler table for Z on the available hardware (floating point and fixpoint operators, register files, ROM) to convert logical time into physical time, and this is the central point of this thesis. In the present case, the problem is of small size (three independent tasks). However, it is easy to imagine bulkier examples: we may need more than six taps, or several filters working in parallel for a multiband equalizer.

As mentioned, one could argue that it would be better to consider, at the symbolic scheduling stage, all the micro-operations generated by each statement. This is of course true in theory. In practice, the size of the problem would increase dramatically – by a factor of 7 in the preceding example. Besides, it is difficult to prevent the scheduler to introduce delays between micro-operations, and hence to imply more registers for holding temporary results. Our approach may be sub-optimal, but we believe that the possible improvements do not warrant the added complexity.

Though we have investigated both approaches, the last decoupling will be good enough. Our fronts are indeed in general chosen such that the number of operations they contain offer enough parallelism to saturate the critical resources. So in the micro-scheduling step, macro-tasks are scheduled independently, but when *assigning* resources, we try to distribute parallel resources among macro-tasks. In other words, after the first two steps, the directed acyclic graph of micro-scheduled macro-tasks (tasks with reservation tables) should still contain a sufficient degree of parallelism. The final resource constraints are then taken into account with the fine-grain scheduler.

Figure 3.3 summarizes both scheduling decompositions. Dotted lines expresses the area of our main contributions as in this thesis we have addressed only some parts of these scheduling

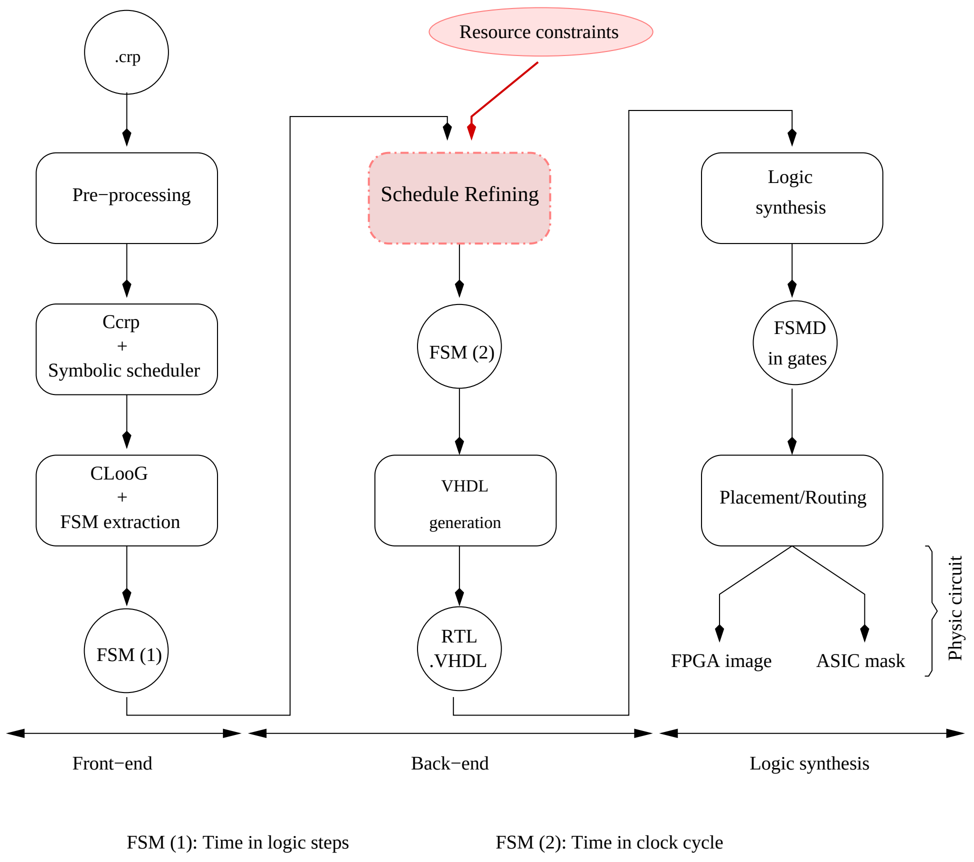


Figure 3.4: Design Flow of our HLS System.

steps.

3.4 Compilation Flow

In order to get good ideas about our scheduling approaches, let us describe their usage in our context. Let us see the design flow of our HLS framework which is diagramed by Figure 3.4.

A brief description of all steps of this HLS system going from an input description written in CRP to a physical view of the target circuit (which will be described via an FPGA image or ASIC mask) through an RTL description is:

- In a pre-processing step, by hand for the moment, each statement of the program is split – if necessary – until it fits the target datapath in the number of simultaneous operations, memory, and register accesses. For example, a high-level statement that reads three different memory locations while the target architecture can only perform two reads simultaneously is decomposed into intermediate operations. For example

```
R:                            b = 3 * a[i+2] *  a[j]
```

  If the array `a` is mapped to a one-port memory or we can't have more than one multiplier in the circuit, this high-level statement should be split into two statements such:

```
R1:                          b1 =  3 * a[i+2]
R2:                          b  = b1 * a[j]
```

This amount of node splitting we do before scheduling is clearly one of the adjustable parameters of our methods, and it is clear that more investigations are needed in order to find the best decomposition, which probably depends on details of the application and the architecture. This issue -how to do this splitting?- is clearly a matter for research however it isn't specially addressed in this thesis;

- After, the behavioral input description is compiled by the *Ccrp* compiler which does a syntactic and semantic analysis, same as a classical compiler. In addition to this, It performs a dataflow analysis [39]. Its aim is to exhibit all the dependencies between array and scalar references of the system. It extracts both kinds of dependencies data dependences and communicating dependences;

- Using the results of this analysis, a *first-level schedule* - a multidimensional schedule- is computed by modeling the problem as an integer linear program [41]. This scheduler is a scalable and modular version of a multidimensional scheduling algorithm [40]. In this thesis, we will ignore the explanation of such construction.

  The output of this scheduling module is a scheduling function θ:

$$\theta : E \longmapsto T$$

  where E is the set of all operations[9] of the program and T is a set of time values. Each time value is an integral vector, totally ordered by lexicographic order. So that it gives for each operation an unique execution date. Indeed for each statement S of the program, the function $\theta(< S, x >)$ is an affine form of the iteration vector x:

$$\theta(< S, x >) = h_S.x + k_S \tag{3.4}$$

  Where h_S is the time vector of S and k_S is a scalar.

  The multidimensional time notion can be easily compared to the decomposition of a date into many dimensions (like year/month/day/hour...) where the first dimensions are the most significants. For the case of the Pipeline program of Section 3.1.1, this scheduling algorithm gives the following schedules:

$$\theta(\langle lb1, i\rangle) = 0, \qquad \theta(\langle lb2, i\rangle) = 0$$
$$\theta(\langle K, i\rangle) = 2i + 1, \qquad \theta(\langle W, i\rangle) = 2i + 2, \qquad \theta(\langle R, i\rangle) = 2i + 3.$$

  This solution means that both statements $lb1$ and $lb2$ start at time 0, after in a cyclic way, -at each 3 consecutive clock tops- a sequential execution of an instance of K, W and R respectively holds.

- Given this schedule, now we regenerate an efficient parallel code. With present day tools [93, 11], it is quite easy to automatically do this and many competitive algorithms try to find a compromise between the code efficiency and its simplicity [5, 20, 93]. In our framework

[9]an operation is an instance of statement defined by the name of the statement and the iteration vector

we have used *ClooG*[10] to generate such code. In Section 3.5 we will explain in detail how the chosen tool works.

From this equivalent parallel code, we build a finite state machine -control automaton- in which the time is measured in logical steps. In this automaton, each state executes a set of data and control independent operations.

- According to the scheduling approach used, we refine the resulting FSM by splitting each logical step to guarantee that resource constraints and peculiarities of the data path are respected .
- Starting from the previous constructed FSM, in the last step of the front-end part we generate an RTL description written in synthesizable VHDL form according to the standard *IEEE 1076-1987* [2].

Now, it remains to submit this RTL description to any logic synthesis tool. In our context we use the ISE Xilinx 6.x[11] tool kit. This synthesis environment uses mainly: 1/ *XST*[12] for the synthesis of gates and 2/ *ModelSim* to perform at several levels temporal simulations.

3.5 *CLooG*: a Code Generator

The problem of automatic code generation is solved thanks to reasoning in the polyhedral model. This model is based on a linear-algebraic representation of programs[13] Indeed, using this representation, the code generation problem entails scanning the $\mathbb{Z}$-polyhedron[14], defined by the iteration domains, in the lexicographic order. In other words, it entails visiting each integral point of a polyhedron in the lexicographic order.

At first, the automatic code generation problem was solved by Ancourt and Irigoin [5]. For more complex situations, the best solution is the Quilleré et al.'s algorithm [93]. Both methods generate code with loops. Boulet and Feautrier [20] proposed another solution in which, they directly generate low level code without loops.

Let a program be represented by its $\mathbb{Z}$-polyhedron -defined by the iteration domains- and a legal schedule. In our framework, we use the *ClooG* tool to generate the control automaton. Indeed *ClooG* [11] gives an efficient parallel code with less control. The heart of the generation process is the Quilleré et al. algorithm. Their technique is simple and can be summarized in three steps:

- It generates loop levels by projecting the polyhedra onto the corresponding dimension.
- Next, it splits the projection into disjoint polyhedra and it sorts the resulting polyhedra according to the lexicographic order.
- Lastly, it recursively generates loop nests that scan each projection.

[10] *CLooG*: for Chunky LOOp Generator. This soft is available at `http://www.cloog.org/`.

[11] http://support.xilinx.com/support/sw_manuels/xilinx6/

[12] Xilinx Synthesis Technology

[13] Also called the polytope model, it became very popular because of its rich mathematical theory and its intuitive geometric interpretation. Moreover it addresses a class of codes with very regular control that includes a large range of real-life program parts [12, 100].

[14] A convex set of points in a lattice (also called lattice-polyhedron), i.e., a set of points in a $\mathbb{Z}$ vector space bounded by affine inequalities [99].

In order to force scanning to respect some rules in addition to the lexicographical order, CLooG allows using some scattering functions, such as the scheduling functions that we have previously computed. Scattering is a shortcut for scheduling and allocation functions and the like. Indeed in order to exhibit parallelism, CLooG applies some transformations on the $\mathbb{Z}$-polyhedron while respecting the scheduling functions.

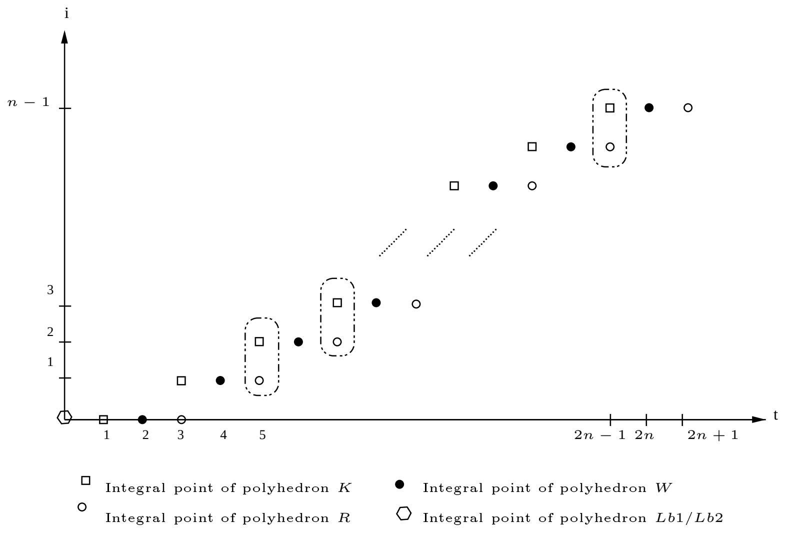
To illustrate the behavior of this algorithm, we unroll it on the Pipeline example. Let us recall that for the case of the Pipeline program, the multidimensional scheduling algorithm gives the following schedules:

$$\theta(\langle lb1, i\rangle) = 0, \qquad \theta(\langle lb2, i\rangle) = 0$$

$$\theta(\langle K, i\rangle) = 2i+1, \qquad \theta(\langle W, i\rangle) = 2i+2, \qquad \theta(\langle R, i\rangle) = 2i+3.$$

These scheduling functions can be represented by the following parameterized 5 polyhedra (it represents the CLooG input):

Polyhedron $lb1$:
$$\begin{cases} t = 0 \\ i = 0 \end{cases}$$
Polyhedron $lb2$:
$$\begin{cases} t = 0 \\ i = 0 \end{cases}$$
Polyhedron K
$$\begin{cases} t = 2i+1 \\ 0 \le i \le n-1 \end{cases}$$
Polyhedron W
$$\begin{cases} t = 2i+2 \\ 0 \le i \le n-1 \end{cases}$$
Polyhedron R
$$\begin{cases} t = 2i+3 \\ 0 \le i \le n-1 \end{cases}$$


i
n − 1
3
2
1
t
1
2
3
4
5
2n − 1
2n
2n + 1
□ Integral point of polyhedron K
● Integral point of polyhedron W
○ Integral point of polyhedron R
⬡ Integral point of polyhedron Lb1/Lb2

In the context of $n > 1$, let us generate the code which scan this 5 polyhedra:

- Firstly, we project the 5 polyhedra on the first dimension t and we split them into disjoint polyhedra. Then, we sort these polyhedra so that the textual order of the loops, scanning the first dimension, complies with the lexicographic order. We get this first pseudo code:

```
for (t=0;t<=0;t++){
  Polyhedron $lb1$: $\{i = 0\}$
  Polyhedron $lb2$: $\{i = 0\}$
}
for (t=1;t<=1;t++){
  Polyhedron $K$: $\{i = 0\}$
}
for (t=3;t<=2n-1;t++){
    Polyhedron $W$:
        $\{i = (t-2)/2\}$
    Polyhedron $R$:
        $\{i = (t-3)/2\}$
    Polyhedron $K$:
        $\{i = (t-1)/2\}$
 }
for (t=2n;t<=2n;t++){
   Polyhedron $W$: $\{i = (t-2)/2\}$
 }
for (t=2n+1;t<=2n+1;t++){
   Polyhedron $R$: $\{i = (t-3)/2\}$
}
```

- We recurse on the resulting disjoint polyhedra: so we project them on the next dimension i and we separate them into disjoint polyhedra. Then we sort these polyhedra so that the textual order of the loops scanning the second dimension complies to the lexicographic order.
- After the elimination of the loops with one iteration, the final code which scan the 5 polyhedron is:

```
lb1(i = 0);
lb2(i = 0) ;
K(i = 0) ;
W(i = 0) ;
for (t=3;t<=2*n-1;t++) {
 if ((t-3)%2 == 0) {
   R(i = (t-3)/2) ;
 }
 if ((t-2)%2 == 0) {
   W(i = (t-2)/2) ;
 }
 if ((t-1)%2 == 0) {
   K(i = (t-1)/2) ;
 }
}
W(i = n-1) ;
R(i = n-1) ;
```

The unknown t expresses the time vector. Conditional statements are generated to guarantee the integrality of subscripts. To avoid such complex subscript functions and guards, we can change this temporal basis thus, moving from a one dimension basis to two dimensions basis. Thus, we obtained the new computed schedule:

$$\theta(\langle lb1, i\rangle) = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \ \theta(\langle lb2, i\rangle) = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

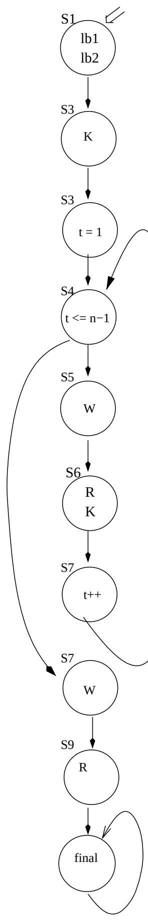


Figure 3.5: Generated FSM for the Pipeline Program.

$$\theta(\langle K,i\rangle) = \begin{pmatrix} i \\ 0 \end{pmatrix}, \theta(\langle W,i\rangle) = \begin{pmatrix} i+1 \\ 0 \end{pmatrix}, \theta(\langle R,x\rangle) = \begin{pmatrix} i+1 \\ 1 \end{pmatrix}.$$

The generated code becomes simpler:

```
lb1 ;
lb2 ;
K(i = 0) ;
for (t=1; t<=n-1; t++) {
   W(i = t-1) ;
   R(i = t-1) ;
   K(i = t) ;
}
W(i = n-1) ;
R(i = n-1) ;
```

As the above code shows, *GLooG* don't consider the body of the statements; it recognizes the statement just by a label which is automatically generated according to the textual order of the apparition of their iteration domain in the GLooG input file. In our framework we generate directly from the *ClooG* internal representation the control automaton while the body of statement is inserted from an external input file. For example, the synthesized FSM for the pipeline program is diagramed by Figure 3.5.

3.6 Conclusion

HLS scheduling is a very constrained task. Indeed, to get an acceptable schedule, that allows exploiting enough parallelism inherent to the input description, we have used some techniques which have been pioneered in automatic loop parallelization. However, these techniques are quite complex and can't take into account all datapath peculiarities and detailed resource constraints. For this we have sketched some scheduling approaches, in which scheduling is performed in gradual ways.

In this chapter, we have explained our general stepwise scheduling approaches and how they are inserted in our framework. Let us recall that, in this thesis, we have addressed only some parts of these scheduling approaches. Indeed, considering the three-step approach, in the following part (Part I), we focus on the third scheduling step problem: *"how to schedule tasks whose resource usage is described by reservation tables"*. In addition, we propose some solutions to perform the last scheduling step defined in the two-step approach in Part II.

Part I

Reservation Tables Scheduling using Dis-equations

Chapter 4

Formalism and a Greedy Solution

This Chapter is divided into two parts. In the first part, we will explain how tasks are represented by mean of reservation tables. According to this task model, a new formulation of resource constraints is developed. In this model, constraints are represented using dis-equations. Then a formal definition of the scheduling problem is deduced.

In the second part, a greedy heuristic is proposed and some related experimental results are reported and commented.

4.1 Introduction

Let us recall that in general, the HLS scheduling problem is an $\mathcal{NP}$-Hard problem and especially when one wishes exploiting parallelism inherent in the input description. For this main reason, among others, we have sketched the three-step scheduling approach. In what follow, we propose to formalize the problem defined in the third scheduling step i.e. the fine-grain scheduling problem.

4.2 Task and Resource Constraints Formalism

In this section, we explain what is our task model – basically a set of (possibly dependent) tasks, each being a complex sequence of elementary pre-scheduled operations – and how we represent resource constraints for such tasks.

4.2.1 Extracting Tasks with Reservation Tables

Basically, a macro-task is a statement in some high-level language (C in our case). As shown in Section 3.3 at the hardware generation level, each macro-task must be split into simpler operations like address calculations, memory accesses and arithmetic operations.

According to the result of the second scheduling step the elementary operations in each macro-task are *mapped to resources* and *pre-scheduled*; indeed, independently the schedule of each macro-task is described by a *reservation table* that states which resources at which cycle are used by this macro-task, relative to its starting time. In other words, this pre-scheduling leads to a reservation table in which the start time of each elementary operation is fixed, once and for all, relative to the start time of the macro-task. From now, each statement can be viewed as a macro-task whose resource usage is fixed.

The resource assignation is done in a simple manner. Indeed, we chose the binding that greedily allocates all the available resources to tasks, i.e., we assign the whole resources to the functional operations of each task. In other words, the binding rule assigns the available resources to the competing tasks in a round-robin way.

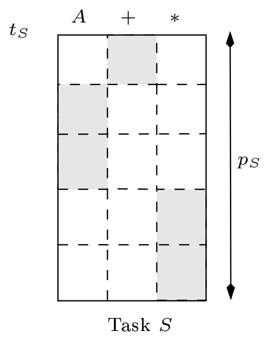
Without loop, the data dependences between operations, into a macro-task, can be represented by a directed acyclic graph. Hence, for this step –micro-scheduling–, we use classical task graph scheduling techniques(Chap. 1 of [32]). Here also, the problem is $\mathcal{NP}$-hard, but there are some heuristics which give approximative solutions and which can be guaranteed to achieve at most twice the optimal execution time. In addition, the node splitting performed at the preprocessing step insure that the size of a macro-task is relatively small hence in most of time the micro-scheduling heuristic reaches the optimal solution.

Now, as macro-tasks are represented by reservation tables and they are data independent, getting a whole schedule entails just fixing the relative starting dates of macro-tasks, while respecting resource constraints and minimizing the total execution time.

4.2.2 Notations

We denote by T the set of n macro-tasks, R the set of resources, t_i the starting date of the macro-task i, and $p_i \geq 0$ the latency of task i (the unit is the clock cycle), i.e., the difference between the ending time of the last elementary operation it contains and the starting time of the first one. The reservation table of task i is thus of size $p_i \times |R|$. Let us see the following macro-task S. Assuming that we have one adder, one multiplier, and that the array a is mapped to one-port memory block A. Assume also that a memory access and the multiplier take 2 cycles and that both can be pipelined. One possible binding is diagramed by the following figure:

```
S:    y = a[i+2] * b
```



A reservation table for S.

In what follows, we will use "task" (resp. "operation") instead of "macro-task/high-level statement" (resp. "micro-instruction/micro-task") for brevity.

4.2.3 Forbidden Distances

To get an optimal schedule we must express efficiently the constraints of the scheduling. Resource constraints are the main ones.

Consider a couple of tasks i and j, with respective starting dates t_i and t_j. In a legal schedule, if the tasks i and j are data independent, they can start at any dates except those which put them into resource conflict. So the intuitive idea is to express the resource constraints as a set of *forbidden distances* $(t_j - t_i)$. Let us explain, assume that a resource $r \in R$ is used at micro-step d_i in the reservation table of task i and at micro-step d_j in the reservation table of task j. (A given

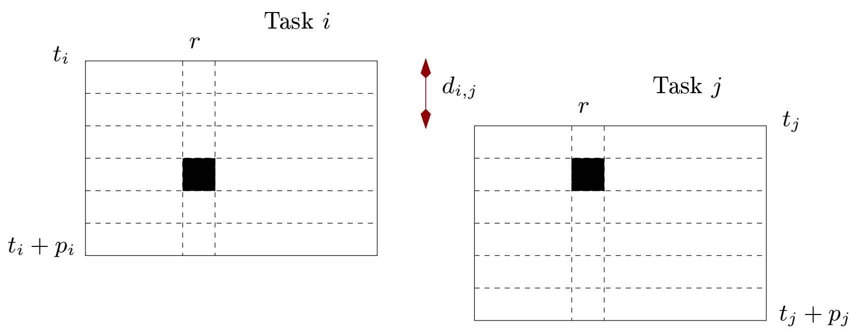


Figure 4.1: Forbidden Distance.

resource r can be used more than once in a given reservation table, so we should use a notation such as $d_{i,r,k}$, but we dropped the indices r and k for clarity.) This means that in a schedule the resource r is used at time step $t_i + d_i$ by task i and at step $t_j + d_j$ by task j. To satisfy the resource constraint for r it is necessary that:

$$t_i + d_i \neq t_j + d_j, \text{ i.e., } t_i - t_j \neq (d_{i,j} = d_j - d_i). \tag{4.1}$$

Note that the values d_i and d_j are problem inputs as the reservation tables are given, whereas t_i and t_j are unknowns. In fact, dis-equation (4.1) eliminates, from the solution space of t_i and t_j, only the forbidden distance $d_{i,j}$. In this way, all resource constraints for a pair of tasks (i, j) can be expressed as dis-equations by a systematic examination of their respective reservation table. It follows that, for the set T of tasks in a logical state, all the resource constraints can be obtained by defining for each couple of tasks (i, j) all the dis-equations expressing the resource constraints. Figure 4.1 illustrates the notion of forbidden distance.

When there are more than one copy of each resource, this simple formulation of the constraints is no longer possible, unless we bind resources to tasks a priori. As mentioned below, our binding rule assigns the available resources to the competing tasks in a round-robin way. We have done some experiments that show that this heuristic has no great influence on the final latency (§ 6.2.4).

4.2.4 Example 1

Consider the following excerpt from the PerfectClub Benchmark, The SPICE[15] program, from line 16 to 19. This example illustrates what a logical step may contain after symbolic scheduling.

```
Task 1:   GSPR = VALUE(LOCM+2)*AREA
Task 2:   GEQ  = VALUE(LOCT+2)
Task 3:   XCEQ = VALUE(LOCT+4)*OMEGA
Task 4:   LOCY = LYNL+NODPLC(LOC+13)
```

The four tasks are data independent. Assume that the available resources are one adder, one multiplier, and two memory blocks: `Val` (where the `VALUE` array is mapped) and `Ndp` (where the `NODPLC` array is mapped). Assume also that a memory access and the multiplication take 2 cycles and that both can be pipelined. Scalar variables like `AREA` or `LOCY` are assumed to be allocated to registers, where they can be accessed in no time. Figure 4.2 diagrams one possible binding, where the label `RM Val` (resp. `RM Ndp`) means to read the memory block `Val` (resp. `Ndp`).

[15]SPICE is a widely used circuit simulation program developed at UC Berkeley.

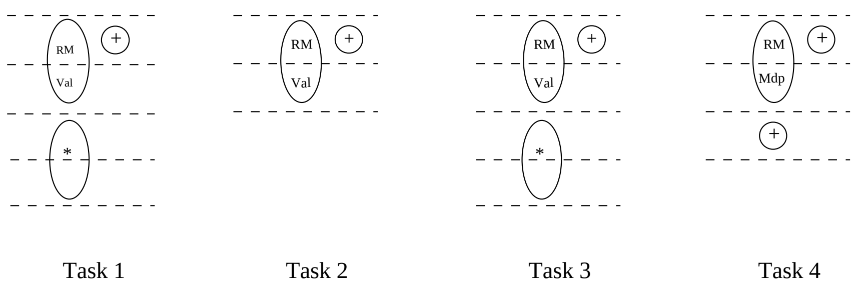


Figure 4.2: Binding for Example 1.

For this example, the system of resource constraints is composed of 9 constraints defined as follows:

$$\begin{cases} t_1 - t_2 \neq 0 & t_1 - t_3 \neq 0 & t_1 - t_4 \neq 0 \\ t_3 - t_4 \neq 0 & t_2 - t_3 \neq 0 & t_2 - t_4 \neq 0 \\ t_4 - t_2 \neq -2 & t_4 - t_1 \neq -2 & t_4 - t_3 \neq -2 \end{cases}$$

For instance, the constraint $t_1 - t_3 \neq 0$ expresses the fact that tasks 1 and 3 cannot start at the same time because (among other reasons) both use the adder in their first step.

4.3 Scheduling Problem Formulation

Now, if there are no data dependences between tasks, finding a *valid schedule* for T entails solving the following system of dis-equations in integer values:

$$\begin{cases} t_i - t_j \neq d^k_{i,j} & i, j \in T \\ t_i \geq 0 \end{cases} \tag{4.2}$$

For a given pair of tasks i and j, there can be several forbidden distances $d_{i,j}$, hence the index k. The set of inequalities $t_i \geq 0$ is added into the system just to fix the origin of the schedule.
In addition, as the goal is to get an optimal schedule (a schedule with minimal execution time or *latency*) we must minimize the total time $\max_i(t_i + p_i)$.

If necessary, dependences between tasks can also be handled; they can be expressed as additional inequalities of the form $t_j - t_i \geq \delta_{i,j}$. When, $\delta_{i,j} \geq 0$, such a constraint means that task j must start at least $\delta_{i,j}$ steps after task i (a typical data dependence); when $\delta_{i,j} \leq 0$, it means that task i can start at most $-\delta_{i,j}$ steps after task j.

4.4 How To solve a System of Dis-equation?

First let us notice that, as defined, the problem of solving such a system of dis-equations while minimizing $\max_i t_i$, with $t_i \geq 0$, is an $\mathcal{NP}$-Complete problem.

Proof. The proof of this $\mathcal{NP}$-completeness is straightforward; we claim that the graph coloring problem is a particular case of the problem defined in (4.2). Indeed, if one takes all d^k_{ij} equal to 0 and all p_i are equal, then the solution, at the end, is to give i a different color than j. Now, the graph coloring problem, in its general forms, is well known as an $\mathcal{NP}$-Complete problem [110]. □

Consequently, we are sure that finding an optimal schedule can't be done in a polynomial time. Nevertheless, there are many methods for solving the system defined in (4.2):

- one can be satisfied with a greedy heuristic such as a list-scheduling algorithms [33]. In these methods, we simulate an execution by maintaining, at each time, a list of ready tasks. This list is ordered according to a given arbitrary order. In most cases, this order favorises the long tasks or tasks belonging to the critical path. Then, we launch the first ready tasks such that no resource is deliberately left idle.
- conversely, if we need more accuracy, in other words, if we search the optimality, there are also many solutions from operation research which are based on:
  - Branch and bound (BAB) techniques [59];
  - Integer Linear Programming techniques [79, 99].
  - Since there is an obvious bound for the t_i ($t_i \leq \sum_i p_i$), another solution can be employed by using constraint handling methods, provided by Constraint logic Programming (CP) paradigm [14]. Indeed, CP uses the concept of constraint solving in a specific computation domains. Generally these domains are finite domains.

Indeed, Integer Linear Programming and Constraint Logic Programming are two alternative approaches for solving combinatorial optimization problems.

Concerning the CP resolution, interval methods, which are the main techniques used by the solvers of the constraints, have shown their ability to locate and prove the existence of an optimal solution in rigorous way, unfortunately, these methods are rather slow. Indeed, a solution can be found by instantiating all domain variables with values from their respective domains. Instantiating a variable may cause the execution of constraints and therefore failure. On backtracking, another value from the domains has to be chosen. These backtrackings may slow down a lot the all process.

The situation is similar for the ILP techniques. In fact, the theoretical complexity of the ILP solutions is exponential in the product of the task number and the number of needed variable in the standard coding [25]. However, we have tried in Chapter 5 to define some economical codings and straightforward formulations.

In the same way, a *Branch-And-Bound*-based method has an exponential time complexity, but it can have a different behavior in practice. Indeed, *Branch-And-Bound* is a meta-method of guidance in the space of solutions. Its strategy of resolution depends strongly on the features of the problem to resolve. For this reason, we have developed a Branch-and-bound in Chapter 6.

In the rest, we will develop and experiment a greedy heuristic which will be used as yardstick for measuring the efficiency and robustness of the exact algorithms developed in both following chapters.

4.5 A Greedy Heuristic

First let us consider a classical greedy heuristic, which can be used for data-independent tasks. It is easy to adapt this heuristic to the case where dependences such as $t_j - t_i \geq \delta_{i,j}$ give rise to a directed acyclic graph.

4.5.1 Algorithm

We use a classical *greedy-scheduling* (GS) heuristic. Without any data dependences, all the tasks are ready at time zero. Tasks are scheduled one after the other. At each step, given a subset T_m of already-scheduled tasks, we check whether the next task i can be scheduled at time 0, i.e.,

Algorithm 1: Greedy Scheduling Algorithm.

```
Data:
  - ListTasks,
  - Resource reservation table for each task $i$ in ListTasks,
  - Forbidden distances.
begin
    forall $i \in$ ListTasks, following the order in ListTasks do
        StartTime $\leftarrow 0$;
        while not (isPossible($i$, StartTime)) do
            StartTime $\leftarrow$ StartTime + 1;
        end
        Schedule[$i$] $\leftarrow$ StartTime;
    end
end
```

if all forbidden distances between i and all tasks in T_m are respected. If not, the start time is incremented, and the process is reiterated.

Algorithm 1 constructs a global reservation table. After each scheduling step, this table is updated. Thus, it is important to emphasize that this algorithm can be used before resource allocation, as for any classical list-scheduling algorithm. Indeed, the `isPossible` procedure can use the information on the number of available resources and takes into account forbidden distances when there remains only one resource to share. In fact, freedom to place binding after or before scheduling gives this heuristic an advantage.

Our algorithm is a pseudo-polynomial heuristic, as its time complexity is $O(n|R|\sum_i p_i)$, where n is the number of tasks.

When the dependence constraints $t_j - t_i \geq \delta_{i,j}$ form an acyclic graph, one can develop a similar heuristic: consider tasks according to some topological order of the graph and place them in a greedy fashion as early as possible while respecting dependence and resource constraints. This is the standard list-scheduling approach [33, 25].

It is important to note that the order in which tasks are considered in the list influences the latency of the schedule. In this first version of the algorithm, we did not take this fact into account. Much work has been devoted to the construction of good priority rules, i.e., in the search for a good task ordering. Of course there are different possible strategies to decide which tasks are given priority in the (frequent) case where there are more free tasks than available resources. But a key result due to Coffman is that any strategie deciding *not to deliberately keep a resource idle* can be shown to achieve good performance [32].

4.5.2 Example 1, Continued

Let us return to the example of Section 4.2.4. An optimal solution (found for example by our exact algorithm in Chapter 6) is $t_1 = 0$, $t_2 = 3$, $t_3 = 1$, $t_4 = 2$, it needs only 5 cycles. The GS heuristic gives the solution $t_1 = 0$, $t_2 = 1$, $t_3 = 2$, $t_4 = 3$ with a latency of 6 cycles. Figure 4.3 diagrams both solutions. The GS algorithm reaches the optimum only for the ordered list Task 1, Task 3, Task 4, Task 2. Note that, in the general case, there may be no order in which the optimal is reached by the greedy scheduling. However, here a deviation of 1 cycle from the optimum is acceptable, in particular if one needs a fast compilation.

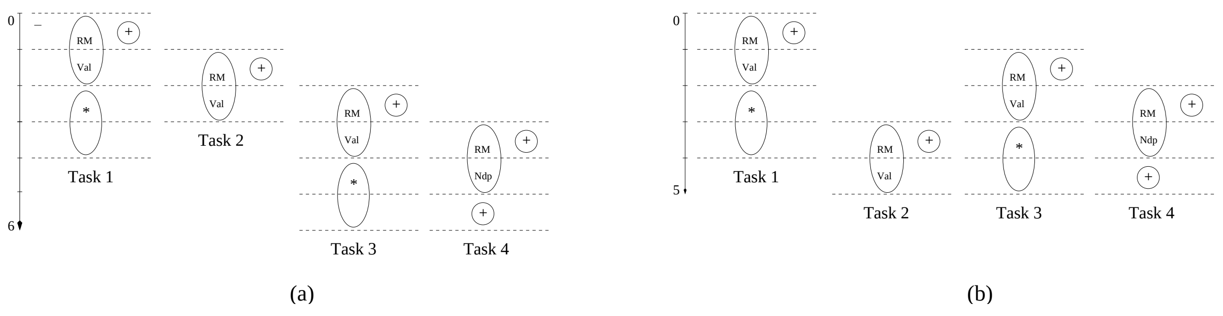


Figure 4.3: (a) Greedy Solution and (b) Optimal Solution.

4.5.3 Experiments

We have implemented this heuristic and tested it on groups of independent tasks from real-life applications. They consist of 22 tests from the PerfectClub [18] and HLSynth95 [85] benchmarks. The *PerfectClub* benchmarks represent applications in a number of areas of engineering and scientific computing and the *HLSynth95* benchmarks, more specifically, represent a repository of applications in embedded systems (see Appendix B for more details on these benchmarks). The runtime is computed in user seconds on a 1.73 GHz *Intel Pentium M* running Linux. Results are reported in Table 4.1.

The test programs are fairly small, they contain between 3 and 9 data-independent (possibly complex) tasks, each one containing between 1 and 15 operations. All kind of resources are considered: sequential resources like memory ports, and combinatorial ones such as adders, multipliers, comparators, and dividers. Also, more than one-cycle delays resources are taken into account. Miscellaneous resource features are considered, for example memories with two-ports, multipliers with two and three cycles, pipelined resources and so on.

Test	T	μT	Schedule latency	Deviation
css1	4	15	6	1
css11	4	15	5	2
css12	4	17	6	2
css2	9	32	7	1
css3	7	27	10	3
css5	3	9	5	0
css6	8	12	4	0
jac1	6	19	6	0
jac2	6	82	23	0
jac3	7	97	20	1
rasm1	3	9	5	0
wss3	5	11	4	0
wss31	5	11	6	1
wss32	5	11	4	0
woc1	4	13	5	0
woc2	7	9	4	1
wss1	4	44	21	5
wss11	4	44	19	3
wss2	3	23	11	1
wss12	4	44	17	4
wmt22	4	31	13	0
css21	9	32	11	1

Table 4.1: Greedy Scheduling Results.

The first three columns of Table 4.1 are the test names, the number n of tasks (column T), and the total number of micro-tasks (column μT) that compose them. For such instances, this heuristic is very fast; its runtime is less than the Linux clock resolution. So, we did not report its runtime in the table, which is about $0.0032s$ in average, a value obtained by timing one million repetitions of the algorithm.

To evaluate the stability of the algorithm, we have repeated it on a sample of n^2 random permutations of the tasks. The "Deviation" column gives the difference between the best and the worst schedule in such sample.

To get an idea about the quality of the computed latencies and the behaviour of the GS heuristic, we need some information relative to the optimal solutions hence we delayed this analysis once we present our exact methods.

4.6 Conclusion

We have presented a formalism, for high-level synthesis, to accurately express resource constraints for complex tasks represented as reservation tables. The resource constraints are modeled by dis-equations and finding an optimal schedule leads to resolving a system of dis-equations. The proposed formalism can be generalized to support problems of resource-constrained scheduling even when tasks are dependent.

Because our scheduling problem is $\mathcal{NP}$-complete, first, we have relied on heuristics. The most natural idea is to use a greedy strategy: at each time step, we try to schedule as many tasks as possible onto available resources. Hence our GS heuristic. The results have shown that this heuristic is very fast and enough stable.

Chapter 5

Integer Linear Programming Approach

Scheduling theory was originated from operations research thus, it is obvious to think about using some of its own techniques to resolve the scheduling problems. Indeed, the best understood and well known exact scheduling techniques are based on combinatorial optimization especially on Integer Linear Programming. The HLS literature offers a very rich variety of ILP formulations. Due to our particular kind of constraints (expressed by means dis-equations), in this chapter some ILP formulations of the scheduling problem defined in Section 4.3 are proposed. Then some experiments are reported and commented.

The main aim of our contribution isn't to bring some improvements in this field, as ILP formulations have been enough studied by many scholars [72, 61, 47, 27, 112, 65] but our ILP scheduling algorithm will be used as yardstick for measuring the efficiency and robustness of one of our branch-and-bound-based algorithms defined in the following chapters.

5.1 General Introduction

Integer and combinatorial optimizations deal with problems of minimizing or maximizing a function of many variables subject to inequalities and equalities constraints when the values of some or all of the variables are restricted to be integral. The versatility of the combinatorial optimization model stems from the fact that in many practical problems, activities and resources, such as machines, airplanes, people and clock cycles are indivisible.

Because of the robustness of the general model, it covers a rich variety of problems [53]. An important and widespread area of applications concerns the management and efficient use of scarce resources to increase productivity. These applications include finance, marketing, production scheduling, distribution of goods and machine sequencing. They also include design problems such as communication and transportation network design, VLSI-circuitry design and testing, the design of automated production systems and design of the layout of circuits to minimize the area dedicated to wires, design and analysis of data networks, solid-waste management, determination of ground states of spin-glasses and many other typical applications: portfolio analysis, high energy physics, x-ray crystallography and molecular biology ....

On the other hand, there are many real-world problems where it is impossible to write down all of the constraints in a mathematically "clean" way. Such problems often arise in scheduling where there are a myriad of constraints and other rules related to what constitutes a "feasible schedule". Furthermore, formulations based on ILP offer the possibility of integrating constraints in a homogeneous problem description and of solving them together.

In what follow, we will just sketch the basics of Integer Linear Programming, which are

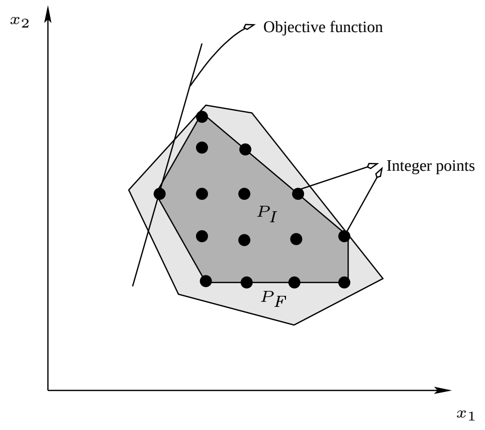


Figure 5.1: Feasible Areas.

essential for understanding of the presented ILP-formulations. For further information see [99] or [83].

Integer Linear Programming (ILP) is the following optimization problem:

$$\min z_{IP} = c^T x$$

$$x \in P_F \cap \mathbb{Z}^n$$

where

$$P_F = \{x \,|\, Ax \geq b,\ x \in \mathbb{R}^n_+\},\ c \in \mathbb{R}^n,\ b \in \mathbb{R}^m,\ A \in \mathbb{R}^{m \times n} \tag{5.1}$$

The set P_F is called *feasible region*. We will assume that $A \in \mathbb{Z}^{m \times n}$ and $b \in \mathbb{Z}^m$ holds. The optimal solution of an Integer Linear Program can be calculated by solving the following problem

$$\min z_{IP} = c^T x$$

$$x \in P_I \cap \mathbb{Z}^n$$

where

$$P_I = conv\,(\{x \mid x \in\ P_F\ \cap\ \mathbb{Z}^n\}) \tag{5.2}$$

Here, *conv* denotes the convex hull. For the two-dimensional case, a representation of P_F and P_I are given in Figure 5.1.

The integral points within P_F denote the feasible solutions to the integer linear problem; depending on the objective function, at least one of them represents an optimal solution. The feasible region defined by (5.1) consists only of the integer points, whereas the feasible region P_I, defined by (5.2), consists of the convex hull of these points.

Let us recall that solution techniques for ILP: solving combinatorial optimization problems can be a difficult task. The difficulty arises from the fact that unlike linear programming, for example, whose feasible region is a convex set, in combinatorial problems, one must search a lattice of feasible points to find an optimal solution. Thus, unlike linear programming where, due to the convexity of the problem, we can exploit the fact that any local solution is a global optimum, integer programming problems have many local optima and finding a global optimum to the problem requires one to prove that a particular solution dominates all feasible points by arguments other than the calculus-based derivative approaches of convex programming.

At least, there are three different approaches for solving integer programming problems, although they are frequently combined into "hybrid" solution procedures in computational practice. They are:

- *Enumerative techniques:* the simplest approach to solving a pure integer programming problem is to enumerate all finitely many possibilities $card(P_I)$. However, due to the "combinatorial explosion" resulting from the parameter "size", only the smallest instances could be solved by such an approach. The most commonly used enumerative approach is the branch and bound method.

- *Relaxation and decomposition techniques:* Relaxing the integrality restriction is not the only approach to relaxing the problem. An alternative approach to the solution of integer programming problems is to take a set of "complicating" constraints into the objective function (with fixed multipliers that are changed iteratively). This approach is known as Lagrangian relaxation. By removing the complicating constraints from the constraint set, the resulting sub-problem is frequently considerably easier to solve. The latter is a necessity for the approach to work because the subproblems must be solved repetitively until optimal values for the multipliers are found.

- *Cutting planes approaches based on polyhedral combinatorics:* The underlying idea is to replace the constraint set of an ILP problem by an alternative convexification of the feasible points and extreme rays of the problem. H. Weyl (1935) established the fact that a convex polyhedron can alternatively be defined as the intersection of finitely many halfspaces or as the convex hull plus the conical hull of some finite number of vectors or points. Original problem formulation are in rational numbers, it implies the existence of a finite system of linear inequalities whose solution set coincides with the convex hull of the integer points in P_F. Thus, if we can list the set of linear inequalities that completely define the convexification of P_F, then we can solve the integer programming problem by linear programming. Gomory derived a "cutting plane" algorithm for integer programming problems.

The complexity of these methods and many other facts make a general ILP problem a $\mathcal{NP}$-complete one [104]; yet many large instances of such problems can be solved. This, however, requires the selection of a structured formulation and no ad-hoc approach [27]. More details on how to formalize several kind of constraints that can be used for scheduling problems are given by Gebotys et al. [47], Kästner et al. [65], and Zhang [117]. We will adapt some of these formulations to our problem.

5.2 Integer Linear Programming Approaches

Let us recall that we have to resolve the following system of dis-equations in integer values:

$$\begin{cases} t_i - t_j \neq d_{i,j}^k & i, j \in T \\ t_i \geq 0 \end{cases} \tag{5.3}$$

while minimizing the total latency $\max_i(t_i + p_i)$.

The reservation tables scheduling can easily be formalized as an Integer Linear Programming problem (ILP) using standard coding techniques.

We propose two ILP-formulations: one using a standard 0/1 encoding that we optimize for our problem, and the second one using the “Big-M” trick. Before this, let us fix some notations.

5.2.1 Notations

We use the following notations: $x_{i,j}$ is a binary variable associated with task i where $x_{i,j} = 1$ if and only if task i is scheduled at the jth clock cycle. The indices j go from 0 to H, a maximal "horizon" for the schedule. We can easily have such upper bound as the p_i are inputs of the problem. In fact, one can set H to $\sum_i p_i$. The variable t_i is the starting date of task i, R the set of available resources, R_r the set of tasks that use the resource r, and $d_{i,r}$ the time step [16] (relative to the beginning of the task) at which the task i uses the resource r.

5.2.2 Standard 0/1 Encoding

A standard way of expressing our scheduling problem is the following. Fix H, the maximal schedule H, to an upper bound for the optimal latency. As explained, we can fix H to $\sum_i p_i$ or, better, to the latency of the solution found by the greedy heuristic GS. Then, minimize the schedule latency L subject to the following constraints (in addition to the fact that all variables are integers and the $x_{i,j}$ are 0/1 variables):

$$t_i = \sum_{j=0}^{H-p_i} j * x_{i,j} \qquad \forall i \in [1 \dots n] \tag{5.4}$$

$$0 \le t_i \le L - p_i \qquad \forall i \in [1 \dots n] \tag{5.5}$$

$$\sum_{j=0}^{H-p_i} x_{i,j} = 1 \qquad \forall i \in [1 \dots n] \tag{5.6}$$

$$\sum_{i \in R_r} x_{i,(t-d_{i,r})} \le 1 \qquad \forall r \in R, \forall t \in [0 \dots H] \tag{5.7}$$

The n equalities in (5.4) define the starting dates t_i as functions of the $x_{i,j}$ binary variables. The inequalities (5.5) express the latency to be minimized. For each task i, the equality (5.6) guarantees that i is executed exactly once. Finally, the inequalities (5.7) express resource constraints for each resource $r \in R$. Once the variables t_i are available – through the constraints (5.4) – the dependence constraints (if any) are naturally expressed as inequalities $t_j - t_i \ge \delta_{i,j}$.

> This ILP formulation, like the previous greedy heuristic, can be used before resource allocation: we have only to replace the right-hand side of the inequalities (5.7) by the number of available resource of type r.

5.2.3 0/1 Simplified Encoding

One reason for an ILP solver to be slow for the previous standard 0/1 formulation is the presence of the constraints (5.4) which have large coefficients, especially when the horizon H is large. To efficiently solve the scheduling problem, it is important to use a more structured formulation, and a mathematical analysis of the problem constraints is needed to find such structured formulation.

Let us recall that by construction, the tasks in T are data independant. Consequently, in such context, the variables t_i are needed only to express the objective function in the constraints (5.5), and we can get rid of these variables as well as the constraints (5.4) and (5.5) by the following

[16] It is possible that, in the same task i, a resource r is used in more than one micro-task. Again, for simplicity, we assume that each task uses each resource at most once, but this may be easily generalized.

trick. Instead of using the ILP solver as an *optimization tool*, we use it to test the *feasibility* of the following reduced system:

$$\sum_{j=0}^{H-p_i} x_{i,j} = 1 \qquad \forall i \in [1 \dots n]$$

$$\sum_{i \in R_r} x_{i,(t-d_{i,r})} \leq 1 \qquad \forall r \in R, \forall t \in [0 \dots H]$$

If the problem is feasible, it means that there is a schedule of latency H or less while, if the problem is unfeasible, H is too small. One can adjust H by decreasing it from some upper bound until a feasible problem is found, or by binary search on H. The smallest H for which there is a solution is the optimal latency L. This multiplies the number of calls to the ILP solver, but each call may be faster. In fact, one can argue that it would be better to do one call to the ILP solver to resolve the problem defined by the 0/1 standard encoding rather than multiple calls to resolve the previous simplified formulation. This seems of course true but in general solving ILP programs with integer variables is well known as more hard than ILP programs with binary variables [58]. We will show this effect in the experiments.

Note that, if there are dependences, we still need the constraints (5.4) to express the dependences constraints, unless we use the technique of Gebotys et al. [47], which has the drawback of greatly increasing the number of constraints.

5.2.4 Big-M Encoding

If we use the standard 0/1 encoding, the number of binary variables is the product of the number of tasks and the number of cycles needed for the whole schedule. However, one can use a more economical encoding: the "Big-M" method. In this formulation we replace each dis-equation by the four inequalities:

$$t_i - t_j \neq d^k_{i,j} \Leftrightarrow \begin{cases} t_j - t_i + (1 - X_{i,j}).M \geq 1 - d^k_{i,j} \\ t_i - t_j + M.X_{i,j} \geq d^k_{i,j} + 1 \\ 0 \leq X_{i,j} \leq 1 \end{cases}$$

where M is a large number (larger than the sum of the p_i, where p_i is the latency of task i). Unlike the usage of the Big-M as a penalization of some variables in classic ILP problem here we use this technique to penalize an inequality. Indeed, the artificial binary variables $X_{i,j}$ and M are used to ensure the disjunction of the two first inequalities.

In this formulation, the number of variables is equal to the sum of the number of dis-equations and the number of tasks, which is independent of the latency of the schedule. However, the coefficients in the inequalities (such as M) are large.

CPLEX: an ILP Solver

Before reporting experimental results let us present the used ILP solver. There are many efficient implementations of ILP solvers available. For instance, we can find the GLPK[17] and Lp-Solver [18] solvers, both are open-source softs. In academic area we can quote PIP[19] an efficient Parametric Integer linear programming solver. . . . For more details on discrete tools, the reader can see Hans Mittelmann's webpage[20].

In our framework, the ILP problems are solved using the CPLEX[21] tool [84]. It is the most used tool by both academic and industrial communities. CPLEX is a powerful commercial tool for solving a general linear optimization problems and also some special optimization problems: network flow problems, quadratic programming problems, quadratic constrained programming problems and mixed integer programming (MIP) problems. In MIP problems, variables are further restricted to take integer values. CPLEX solves MIP problems using a general branch & cut algorithm.

The CPLEX kit offers three forms: 1/ an *Interactive Optimizer* which is an executable program that can interactively solve a problem given in certain standard formats, 2/ a *Concert Technology* which is a set of JAVA, C++ and .NET class libraries, and 3/ *Callable Library* which is a C library. In our experiments we have use both interactive Optimizer and the Callable library.

In addition, CPLEX allows many interaction and actions before and during the resolving process: *preprocessing* which tries to simplify the problem, *monitoring* the iteration by giving some log information and possibility to halt the resolution process and to recover it after some changes.

5.2.5 Experiments

We have implemented these ILP methods on the same benchmarks described previously in Chapter 4. The results are presented in Table 5.1. The third column reports the latency of the optimal schedule. The runtime for the original ILP formulation given by the constraints (5.4) to (5.7) is reported in the columns "0/1 Standard Encoding", with the schedule horizon H fixed to an upper bound for the optimal latency, either $\sum_i p_i$ or the latency given by the greedy heuristic GS, which reduces the number of variables. The column "0/1 Simplified Encoding" gives the runtimes when the latency L and the variables t_i are not expressed in the constraints so the constraints (5.4) and (5.5) are removed. The latency is optimized by decrementing H from the latency obtained by the greedy heuristic GS, in our case, GS heuristic gives solutions that are very close to optimal, so decrementing H is more efficient than a binary search. The last column gives the results for the Big-M method.

These results show that the first 0/1 Standard Encoding method is the slowest. This is due to the large number of unknowns. The Big-M method is faster for small problems, when the solver time is dominated by the time to set up the constraints. However, it gives running times of the same order of magnitude as the second 0/1 standard encoding method.

For some paradoxical cases, increasing the horizon (and hence the number of unknowns) actually reduce the running time. This is probably due to the well-known fact that ILP solvers

[17]A GNU Linear Programming solver. Available at `http://www.gnu.org/software/glpk/`

[18]A GNU Linear Programming solver. Available at `http://www.cs.sunysb.edu/~algorith/files/linear-programming.shtml`

[19]Available at `http://www.prism.uvsq.fr/~cedb/bastools/piplib.html`

[20]Available at `http://plato.asu.edu/bench.html.`

[21]http://www.ilog.com/

Test	μT	Opt. Sched.	ILP formulations			
			0/1 Standard Encoding		0/1 Simplified Encoding	Big-M
			H set to $\sum p_i$	H set by GS	initial H set by GS	
css1	15	5	0.19s	0.13s	0.2s	0.06s
css11	15	4	0.21s	0.14s	0.22s	0.08s
css12	17	5	0.24s	0.22s	0.21s	0.07s
css2	32	6	0.91s	0.95s	0.77s	0.27s
css3	27	9	1.09s	0.67s	0.3s	2.6s
css5	9	5	0.13s	0.29s	0.17s	0.09s
css6	12	4	0.26s	0.13s	0.18s	0.1s
jac1	19	6	0.36s	0.14s	0.13s	0.07s
jac2	82	22	4' 42s	7,32s	1.83s	5.52s
jac3	97	19	2' 02s	3,47s	2.57s	6.2s
rasm1	9	5	0.1s	0.09s	0.15s	0.06s
wss3	11	4	0.21s	0.15s	0.18s	0.08s
wss31	11	6	0.26s	0.13s	0.19s	0.1s
wss32	11	4	0.24s	0.13s	0.16s	0.12s
woc1	13	5	0.18s	0.12s	0.14s	0.07s
woc2	9	4	0.21s	0.13s	0.16s	0.1s
wss1	44	17	1.3s	0.8s	1.26s	0.5s
wss11	44	16	1.1s	0.54s	0.75s	0.3s
wss2	23	9	0.25s	0.42s	0.62s	0.07s
wss12	44	16	1.13s	2.75s	0.83s	0,23s
wmt22	31	13	0.6s	0.33s	0.25s	0.12s
css21	32	10	1' 34s	4.64s	0.48s	1' 57s

Table 5.1: Scheduling Results with the Different ILP Formulations.

are sensitive to the variables and constraints ordering. These variations are particularly visible for small runtimes.

The remaining version (0/1 Simplified encoding) gives the best running times for complex problems, which are the most important for practical applications.

5.3 Conclusion

We have presented two structured ILP formulations to resolve the reservation tables scheduling problem: a standard 0/1 encoding that we have simplified and an encoding using a Big-M trick. The first proposed encoding can be generalized to support problems of resource-constrained scheduling even when tasks are dependent. Furthermore, this scheduling solution can be done before resource assignation.

The ILP-based scheduling optimization are $\mathcal{NP}$-complete. This is of course true in general but, in our context we have relied on two facts. Firstly, let us recall that by construction, we deal with problems that size are relatively small (the size of a front). Secondly, the adopted ILP formulation uses only binary variables and it is well known that solving integer linear programs with just binary variables is easier than solving ILP programs with large integer ones [58].

The experiments have shown that the runtimes for all formulations are sufficiently acceptable, at least for our benchmarks, in contrast to the high exponential theoretic complexity of the ILP-based algorithms. But, the 0/1 Simplified encoding version gives the best running times for complex problems, which are the most important for practical applications.

Chapter 6

Branch-and-Bound-Based Longest-Path Computation Solution

The resource-constrained scheduling formulation, presented in Chapter 4, will be used here to develop another practicable scheduling alternative to the ILP one. In this chapter, an exact algorithm based on a branch-and-bound technique is developed as well as some techniques which are used to improve its runtime. Finally some experiments are reported and commented.

Despite the $\mathcal{NP}$-completeness of the defined scheduling problem, we look forward reaching an algorithm with a better *practical temporal complexity*. Indeed, we rely on the features of the branch-and-bound approach as a meta-method which can be a more adjustable method than the integer linear program techniques.

6.1 General Introduction

Branch-and-Bound is a general algorithmic meta-method for finding optimal solutions of various optimization problems, especially in discrete and combinatorial optimization. The method was first proposed by Land and Doig in 1960 for integer linear programming contexts [70] and by Murty et al. in 1962 in an unpublished paper at Case Institute of Technology: "The Traveling Salesman Problem: Solution by a Method of Ranking Assignments" in the context of a combinatorial problem. The branch-and-bound approach is used for a number of $\mathcal{NP}$-hard problems, such as: knapsack problem, integer programming, nonlinear programming, traveling salesman problem, quadratic assignment problem, maximum satisfiability problem and scheduling problems....

The essence of the branch-and-bound approach is the following observation: at any node in the total enumeration tree of the solution space, if one can show that the optimal solution cannot occur in any of its descendents. Thus, there is no need to consider those descendent nodes. Hence, one can "prune" the tree at that node. If we can prune enough branches of the tree in this way, we may be able to reduce it to a computationally manageable size. Note that, we are not ignoring those solutions in the leaves of the branches that we have pruned, we have left them out of consideration after we have made sure that the optimal solution cannot be at any one of these nodes. Thus, the branch-and-bound approach is not just a heuristic, or approximating, procedure, but it is an exact optimizing procedure. However, one can also use it as a basis of various heuristics. For example, one may wish to stop branching when the gap between the upper and lower bounds becomes smaller than a certain threshold.

A branch-and-bound algorithm requires two tools: the *Branch* and the *Bound* procedures;

hence its name. In the branch procedure, the solution space is split into disjoint subsets (feasible subregions) so that no solution will be lost. The procedure is repeated recursively in all the subregions and all produced subregions naturally form a tree structure, called search tree or branch-and-bound-tree. Its nodes are the constructed subregions. The bounding tool, is a fast way of finding upper or lower bounds for the optimal solution within a feasible subregion.

Now, how can we make sure that the optimal solution cannot be at one of the descendents of a particular node on the tree? It is always possible to find a feasible solution to a combinatorial or discrete optimization problem. If available, one can use some heuristics to obtain a "reasonably good" solution. Let us call this solution the *incumbent*. Then at any node of the tree, if we can compute a "bound" on the best possible solution that can be expected from any descendent of that node, we can compare the "bound" with the objective value of the incumbent. If what we have on hand, the incumbent, is better than what we can ever expect from any solution resulting from that node then it is safe to stop branching from that node. In other words, we can discard that part of the tree from further consideration.

The efficiency of the method depends critically on many facts: 1/ the effectiveness of the branching and bounding algorithms used; bad choices could lead to repeated branching, without any pruning, until the sub-regions become very small. In that case the method would be reduced to an exhaustive enumeration of the domain, which is often impractically large, 2/in addition, the efficiency depends also on the good formulation of the objective function and on how much the incumbent solution is tightened compared to the optimal solution.

This has been a brief introduction to the branch-and-bound approach. For a more detailed discussion, the reader is referred to Chap. 9 and 10 of [81].

It should be clear that, like dynamic programming, we cannot talk about a branch-and-bound algorithm that can solve all discrete and combinatorial optimization problems by a uniform model. There is no universal bounding algorithm that works for all problems. Indeed, branch-and-bound itself is just a meta-algorithm, which can be declined in many different directions. According to the general purpose of this approach, in what follow, we design a scheduling algorithm with a specially designed branching and bounding algorithms looking forward to a better alternative to the ILP algorithms.

6.2 An Exact Branch-and-Bound Solution

Let us recall that we have to resolve this system of dis-equations in integer values:

$$\left\{ \; t_i - t_j \neq d_{i,j}^k \qquad i, j \in T \right. \tag{6.1}$$

while minimizing the total latency $\max_i(t_i + p_i)$. We propose the following strategy, which progressively builds a search tree of subproblems:

- At the root, we start with the empty system (for data-independent tasks);
- At each node N of the tree structure, we deal with a new constraint (dis-equation e of the given system). It is clear that the dis-equation e can be seen as the disjunction of two inequalities [22]:

[22]Note that our framework will work the same if instead of a forbidden distance (i.e., a single value) we express a forbidden *interval*, e.g., when a resource is used in both tasks for several consecutive cycles.

$$t_i - t_j \neq d_{ij}^k \Leftrightarrow \begin{cases} e_1 : t_i - t_j \leq d_{ij}^k - 1 \\ \quad\quad \text{or} \\ e_2 : t_i - t_j \geq d_{ij}^k + 1 \end{cases}$$

hence we perform a separation by introducing the inequality e_1 (resp. e_2) into the left child (resp. right child) of N.

It is easy to prove that this branching is legal. Indeed, the inequalities e_1 and e_2 form two disjoint sets $e_1 \cap e_2 = \emptyset$ and their union is $e_1 \cup e_2 = e$, which means that we are neither losing nor duplicating any solution in branching.

Each leaf of the tree corresponds to a system of *inequalities* whose solutions are solutions to the system of *dis-equations* (6.1). Conversely, any solution to the system (6.1) is solution to the system defined at a leaf, for one and only one leaf.

- During the resolution process, we maintain the latency of the best schedule computed so far. At the beginning, we can set this value L_{best} to $\sum_i p_i$ (which is the latency of the sequential execution of the tasks). L_{best} is the incumbent solution.
- At each node N, we treat the system defined by the inequalities introduced by all nodes belonging to the branch from the root to this node N. Except for the leaves, a schedule for this system is a partial schedule; it is not a schedule for the whole system (6.1) as it respects only part of the constraints. However, the latency L_{local} of an optimal schedule for this partial system is a lower bound for the latency of any schedule for the system defined at any leaf of the subtree below N. The pruning can occur in both possible situations:
  1. If $L_{\text{local}} \geq L_{\text{best}}$, the subtree below N is not constructed as it will never lead to a better complete solution.
  2. The system may not be feasible; in this case, the subtree below N is not constructed either.
- At a leaf, we have exhausted all the constraints, so we can now compute an actual solution. If its latency is better than L_{best}, then L_{best} is updated.
- The algorithm stops when all the branches are explored; the whole space of solutions has been explored and L_{best} is returned as the optimum solution.

note It is important to note that this strategy can be applied even if, at the root the system is not empty but contains some other constraints such as data dependence between tasks of the form $t_j - t_i \geq \delta_{i,j}$ (classical precedence constraints). Therefore, our branch-and-bound method can deal with data-dependent tasks too even though we do not primarily need it in our context (by construction we get independent tasks when the symbolic schedule is performed). note

The core of the algorithm is the evaluation and eventual pruning of a node. We now explain this operation in details.

6.2.1 Finding the Local Bound

When the "branch" operation is done (i.e., once e_1 or e_2 is selected) at each node of the tree structure, we have to examine and resolve a system of l inequalities, where l is the level of the node. This system can be normalized as follows:

$$t_j - t_i \geq w_{i,j} \tag{6.2}$$

where $w_{i,j} \in \mathbb{Z}$ is the maximal value of the right-hand sides of all inequalities of type $t_j - t_i \geq \dots$ introduced so far. The values $w_{i,j}$ are integers of arbitrary sign. This problem can be modeled by a weighted directed graph $G = (V, E, w)$, with one vertex for each i and an edge from i to j with weight $w_{i,j}$ for each inequality. Note that G may have cycles. This model is known, in the scheduling literature, as a task graph [46].

In this formalism, the key point is that an optimal schedule is obtained by computing the simple paths of maximal weight in G. Let us see why. First, note that if we sum the inequalities $t_j - t_i \geq w_{i,j}$ along a cycle, we obtain an inequality of the form $0 \geq W$, where W is the weight of the cycle. Hence, if G has a cycle of positive weight then the problem has no solution. Conversely, if G has only nonpositive cycles, we can define, for each vertex i, the maximal weight a_i of a path leading to i (an empty path has weight 0). This is due to the fact that following a cycle cannot increase the weight of a path, hence all maximal weight paths are simple and each a_i is finite. As the maximal weight of a path leading to j is at least the weight of any path going first through i, we have $a_j \geq a_i + w_{i,j}$. Therefore, the a_i are a solution of the problem. Furthermore, any non-decreasing objective function of the t_i (for example the latency $\max_i(t_i + p_i)$) is minimized by a_i. Indeed, for any solution t_i, it is easy to see that t_i is at least the weight of any path leading to i (make an induction on the path length), thus $t_i \geq a_i$. This formulation can be simplified by introducing an initial task, with an edge of weight 0 from it to all other tasks, and a terminal task, with an edge of weight p_i from any task i to the terminal task. The latency is now given by the maximal weight of a simple path from the initial to the terminal task.

There are many algorithms for finding paths of maximal[23] weights in a graph [30]. We could use the Bellman-Ford algorithm or Floyd's algorithm directly at each node of the BAB tree. Moreover, we can use the fastest Dijkstra's algorithm if all edge weights are nonpositive. But we can do better: we can reduce the complexity of the method by noticing that, at each stage of the BAB algorithm, we add a new edge to a graph in which some information on paths of maximal weights has already been computed. What we need then is an incremental version of maximal-weight paths algorithm. In the following, according to our context we propose two algorithms based respectively on Floyd's and Dijkstra's algorithm.

6.2.1.1 Floyd-based Algorithm

Floyd's algorithm [42] computes, with complexity $O(|V|^3)$, the relation of accessibility in a graph $G = (V, E, w)$ by computing, for each couple of vertices (i, j), the maximal weight $a_{i,j}$ of a path from i to j. This algorithm assumes that G has no cycle with positive weight (otherwise there is a path with infinite weight in the graph), but it can be modified to also detect positive cycles, in which case the system, defined by (6.2), has no solution. Allison et al. [4] modified Floyd's algorithm to produce a dynamic algorithm, but in their variant, at a given stage in the dynamic algorithm, they introduce a new vertex plus a set of its arcs -to and from it-. According to our context, we need just an algorithm which updates the maximal-weight paths matrix, when adding at each stage, just one new arc.

To get this algorithm, let us recall that, at a node of the BAB process, we have to compute the maximal weight $a'_{i,j}$ of a path from i to j (for any i and j) in the graph $G' = (V, E \cup \{e\}, w)$, where $G = (V, E, w)$ is the graph at its parent node and the edge $e = (x, y)$ with weight $w_{x,y} = w_0$ represents the constraint to be added to this node. In G, we have already computed the maximal weight $a_{i,j}$ of a path from i to j for any i and j. It remains to consider paths that go through e.

[23]In the literature, these algorithms are often presented as finding paths of minimal weight. This is the same, one just have to change the weight signs. Our explanations are based on maximal weight paths.

We first need to check that G' has no cycle of positive weight. If this is the case, this means that there is a cycle of positive weight that goes through the new edge e (from x to y) with weight w_0 and then back to x, in particular through a path of maximal weight (in G), i.e., of weight $a_{y,x}$. Thus, G' has a cycle of positive weight if and only if $w_0 + a_{y,x} > 0$. Otherwise, the new $a'_{i,j}$ can be easily obtained by the relation $a^{'}_{i,j} = \max\{a_{i,j}, a_{i,x} + w_0 + a_{y,j}\}$. Note also that when $w_0 \leq a_{x,y}$, the new constraint is actually redundant and no update is necessary.

Our Floyd-based algorithm (Algorithm 2) follows this strategy. We get the dates $t_i = \max_j a_{j,i}$ and an evaluation of L_{local} as $\max_i t_i$, in $O(|V|^2)$ instead of $O(|V|^3)$. At the root of the BAB process, we set all $a(i,j)$ to ∞, as the system is empty -no constraint-.

One can notice that this algorithm don't use the same principle as in Floyd's original algorithm, we call it Floyd-based as it uses the same definition of the matrix and resolves the same problem.

Algorithm 2: Floyd-based Incremental Algorithm.

```
Data: $G = (V, E, w)$, Floyd's matrix $a$ for $G$, $e = (x, y, w_0)$ edge to add
begin
    if $w_0 + a_{y,x} > 0$ then
        Exit; /* Elimination, no solution below */
    end
    if $w_0 > a_{x,y}$ then
        /* Update is needed */
        for $i$ from 1 to $n$ do
            for $j$ from 1 to $n$ do
                $a_{i,j} = \max\{a_{i,j}, a_{i,x} + w_0 + a_{y,j}\}$ ;
            end
        end
    end
end
```

6.2.1.2 Dijkstra-based Incremental Algorithm

In this algorithm, we only compute the maximal weight t_i of a path leading to each vertex i, instead of all $a_{i,j}$ for any i and j. For that, we could apply the Bellman-Ford algorithm with complexity $O(|V||E|)$ but again, we can do better using the knowledge we have from the parent node. We use an idea similar to Johnson's algorithm [30] to be able to use Dijkstra's algorithm [36], which is the best known solution, by finding an equivalent system of nonpositive weights (*reweighting*).

In Algorithm 3, we compute the values t'_i in the graph $G^{'} = (V, E \cup \{e\}, w)$, defined as below. We assume that the t_i for G are available from the parent node. Again, we need to solve two problems. First, we need to check the feasibility of the problem, i.e., to check that no positive cycle is created when adding e. In a second time, if the problem is feasible, we need to compute the new solution t'_i.

Let us first explain the general mechanism we use in this algorithm to be able to use Dijkstra's algorithm. When all edge weights w in a graph $G = (V, E, w)$ are nonpositive, we can find a path of maximal weight from a source s to each vertex $i \in V$ by running Dijkstra's algorithm. If G has a positive weight, we will first modify the edge weights w into nonpositive weights w^r, thanks to a well-chosen reweighting function r (a function that assigns an integer r_i to each vertex i) such that $w^r_{i,j} = w_{i,j} + r_j - r_i \leq 0$. It is easy to see that $G = (V, E, w)$ has a cycle of positive weight if and only if $G^r = (V, E, w^r)$ has a cycle of positive weight because the cycle weights are not changed by reweighting. Furthermore, the weight $w^r(P)$ in G^r of a path P from i to j is equal to $w(P) + r_j - r_i$.

Using this reweighting mechanism, we get an incremental algorithm (Algorithm 3) faster than Algorithm 2, though more complicated. Again, we first check that the problem is feasible and then, if it is, we compute the new solution t'_i.

Feasibility

We use the same argument as for the Floyd-based incremental algorithm. The graph $G' = (V, E \cup \{e\}, w)$, where the weight of e is w_0, has a cycle of positive weight if and only if it has a cycle of positive weight that goes through e, since $G = (V, E, w)$ has no cycle of positive weight. As already mentioned, this is equivalent to the fact that $w_0 + a_{y,x} > 0$ where $a_{y,x}$ is the maximal weight of a path in G from y to x.

To compute $a_{y,x}$, thanks to Dijkstra's algorithm. We proceed as follows: remember that we are given t_i, for all $i \in V$, the maximal weight of a path in G leading to i. These values are such that, for each edge $(i, j) \in E$, $t_j - t_i \geq w_{i,j}$, i.e., they satisfy the system of constraints for G. Let us define G^r with $r = -t$. We have $w^r_{i,j} = w_{i,j} + r_j - r_i = w_{i,j} - t_j + t_i \leq 0$. We can therefore compute in G^r, using Dijkstra's algorithm, the maximal weight $a^r_{y,z}$ of a path from y to any reachable vertex z. We then obtain $a_{y,z}$ thanks to the relation:

$$a_{y,z} = a^r_{y,z} + r_y - r_z, \text{ i.e., } a_{y,z} = a^r_{y,z} + t_z - t_y.$$

We then conclude that the system of constraints defined by G' is feasible if and only if $w_0 + a^r_{y,x} + t_x - t_y \leq 0$ (pick $z = x$ in the previous relation) or x is not reachable from y in G (i.e., $a_{y,x} = a^r_{y,x} = -\infty$).

New Solution t'_i

If the problem is feasible, we still have to compute t'_i the maximal weight of a path leading to i in G'. We can do this by adding a fictive source in V, i.e., a new vertex s in V and for each i in V a new edge (s, i) of weight 0. We can then use Dijkstra's algorithm in G' if G' has nonpositive weights. If not, we have to perform a reweighting. Unfortunately, this time, $-t$ may not be an adequate reweighting function because of the new edge e of weight w_0, if $t_j - t_i < w_0$. However it is possible to find a reweighting function r thanks to the values $a_{y,i}$ we just computed during the feasibility test. Indeed, choose K such that $K \leq a_{y,j} - t_j$ for all j reachable from y and, if x is not reachable, $K \leq -t_x - w_0$. We claim that the function r defined by

$$r_i = \begin{cases} -a_{y,i} & \text{if } i \text{ is reachable from } y \\ -t_i - K & \text{otherwise} \end{cases}$$

is a valid reweighting, i.e., is such that $w_{i,j} + r_j - r_i \leq 0$ for each edge (i, j), including the new edge $e = (x, y)$. (Note: for s, we let $t_s = 0$. Then, for any vertex i in G, we have $t_i \geq t_s + w_{s,i}$ since $t_i \geq 0$ and $w_{s,i} = 0$. We also let $r_s = -t_s - K$ as for any vertex not reachable from y.)

Proof. Consider an edge $(i, j) \in E \cup \{e\}$. Only three situations are possible: neither i nor j are reachable from y, both i and j are reachable from y, or j is reachable from y but not i.

- In the first case, $(i, j) \neq e$ and $w^r_{i,j} = w_{i,j} - t_j - K + t_i + K = w_{i,j} + t_i - t_j \leq 0$.
- In the second case, $w^r_{i,j} = w_{i,j} - a_{y,j} + a_{y,i} \leq 0$ by definition of $a_{y,i}$ and $a_{y,j}$ as maximal path weights from y to i and from y to j.

- In the last case, $w^r_{i,j} = w_{i,j} - a_{y,j} + t_i + K$. If $(i,j) \neq e$ then $w^r_{i,j} \leq -a_{y,j} + K + t_j$, otherwise $w^r_{i,j} = w_0 + t_x + K$. In both cases, $w^r_{i,j} \leq 0$ by choice of K.

Therefore r is a valid reweighting. □

We can then compute, using Dijkstra's algorithm, the maximal weight t'^r of a path from s to any vertex i in the graph G'^r and we finally go back to t'_i with the relation $t'_i = t'^r_i - r_i + r_s$.

Note that, as in our Floyd-based incremental algorithm, we can add a preliminary test ($t_y \geq t_x + w_0$ in Algorithm 3) to minimize computations when it is possible to determine that the new constraint is redundant for the previously-computed solution $(t_i)_{i \in V}$. However, the edge should be nevertheless added to the graph as it may not be redundant for the constraints themselves, but just for this particular solution.

6.2.2 Complexity

Dijkstra's static algorithm has a complexity $O(n^2)$, for $n = |V|$ vertices and $m = O(n^2)$ edges. However, if one implements its priority queue with a specific data structure like a binary heap (resp. Fibonacci heap), the complexity is reduced to $O((n + m) \lg n)$ (resp. $O(n \lg n + m)$). Algorithm 3, whose core is Dijkstra's static algorithm, has the same complexity. Moreover, compared to the Floyd-based incremental algorithm which requires $O(n^2)$ memory, only $O(n)$ memory is needed here. Thus, Algorithm 3 is faster and less memory consuming. In addition, this algorithm can be speedup by replacing the second call to Dijkstra's algorithm by one of its dynamic versions recently published (the most important are the ones of Ramalingam and Reps [95] and Frigioni et al. [43]). The first call to Dijkstra's algorithm can't be replaced by a dynamic version as the source may change at each stage.

In the worst case, when no elimination has been done, we have to examine each node of the BAB tree structure. Thus, 2^m nodes, where m is the number of dis-equations. At each node, we update Floyd's matrix in $O(|V|^2)$ (resp. we update the maximal weight t_i, using Dijkstra-based incremental algorithm in $O(n \lg n + m)$. Hence, the worst-case complexity of the BAB algorithm is $O(|V|^2.2^m)$ (resp. $O((n \lg n + m).2^m)$). In fact, as an enumerative method, it is clear that the branch-and-bound method scans all or part of the solution space. In general, this leads to such exponential theoretical complexity. However, in practice, as we will see in the experiments, many branches are not explored and the algorithm is much faster, except for a few pathological cases.

6.2.3 Speeding up the BAB Algorithm

As discussed in the previous section, the BAB algorithm computational complexity can be very high. For this reason, we try to focus on some facts that could reduce the practical complexity of the designed algorithm. Let us recall that improving any BAB-based algorithm efficiency leads us to consider some critical facts, as:

1. the choice of the branching and bounding algorithms so that prunings holds early;
2. how much the initial best solution (incumbent solution) is close to the optimal solution;
3. objective function formulation. Nevertheless, in our context, it is difficult to intervene successfully here.

Algorithm 3: Dijkstra-based Incremental Algorithm.

```
Data: $t_i$, the maximal weight of a path leading to $i$ in $G = (V, E, w)$, $e = (x, y, w_0)$ edge to add
Result: $t'_i$, the maximal weight of a path leading to $i$ in $G' = (V, E \cup \{e\}, w)$.
begin
    if $t_y \geq t_x + w_0$ then
        Return $\{t_i\}_{i \in V}$; /* add $e$ but no update needed */
    else
        $r_i = -t_i$ for all $i \in V$;
        $\{a^r_{y,z}\}_{z \in V} \leftarrow$ DIJKSTRA$(G^r, y)$ ;
        $a_{y,z} = a^r_{y,z} + t_z - t_y$ for all $z \in V$;
        if $w_0 + a_{y,x} > 0$ then
            Exit; /* Elimination, no solution below */
        end
        add $s$ in $V$, $t_s = 0$, $\forall i$, add $(s, i)$ in $E$, $w_{s,i} = 0$;
        define $K$ such that $K \leq a_{y,j} - t_j$ for all $j$ with $a_{y,j} < +\infty$ and $K \leq -t_x - w_0$ if $a_{y,x} = +\infty$;
        $r_i = -a_{y,i}$ for all $i \in V$ reachable from $y$; $r_i = -t_i - K$ otherwise;
        $\{a'^r_{s,i}\}_{i \in V} \leftarrow$ DIJKSTRA$(G'^r, s)$ ;
        Return $\{t'_i = a'^r_{s,i} - r_i + r_s\}_{i \in V}$;
    end
end
```

First, let us examine the second possibility. Recall, that at the BAB process initialization, we set the L_{best} value to $\sum_i p_i$. If one can have a better bound than $\sum_i p_i$, it will avoid building some subtrees until a better lower bound is found. The GS heuristic (see Section 4.5) gives a schedule that seems quite close to the optimum. Its quality is unfortunately without guarantee, but it can be used as the L_{best} value initialization. See Section 6.2.4 for an evaluation of the effect of this initialization.

It remains to intervene in the first possibility. Indeed, as mentioned earlier, our BAB algorithm uses two tests to avoid building a subtree. The first one checks the feasibility of the problem, i.e., that no positive circuit is created when adding a new constraint (this is done by the test $w_0 + a_{y,x} > 0$ in both the Floyd-based incremental algorithm and the Dijkstra-based incremental algorithm as seen previously). The second one intervenes when a high lower bound is found ($L_{\text{local}} \geq L_{\text{best}}$). Thus, for improving the BAB runtime, we focus on these two facts.

In the following, as we will talk both about undirected and directed graphs, we use the following terminology to avoid ambiguities: we use edge and cycle for undirected graphs, and arc and circuit for directed ones.

Let us now consider a strategy that makes positive circuits appear as soon as possible. We did some experiments showing that the BAB runtime highly depends on the order in which constraints are examined. With some random permutations on the constraints, we observe in some case that the runtime decrease by a factor of 20. For this reason, we designed several heuristics which arrange the constraints to improve the BAB runtime. This reordering can be done statically as well as dynamically:

- Statically, the constraints are reordered before applying the BAB algorithm. At this level, we deal with dis-equations (i.e., edges) thus, it is difficult to guess all existing paths, and therefore all circuits. However, we can allow more time to reorder because we do it only once, as a pre-processing time.
- Dynamically, during the BAB algorithm coupled with the Floyd-based incremental algorithm, we try to get the more convenient constraint. Because we have more information

about the graph features (Floyd's matrix), we can choose the constraint that reduce the final explored tree size.

We now describe four reordering heuristics. The first three are static and the fourth is dynamic.

6.2.3.1 Heuristic 1

This heuristic, which is based on probabilities, is a greedy one. Our goal is to try to keep the subgraph defined by the constraints as connected as possible so that circuits (and maybe circuits of positive weights) appear. This algorithm builds the list of constraints by selecting constraints successively as follows: at each step, we maintain a list $\mathcal{L}$ of vertices that are visited, the criterion of selection favors the constraint $c : t_i - t_j \neq d_{i,j}$ according to the following order: a) i and j belong to $\mathcal{L}$, b) either i or j belongs to $\mathcal{L}$, c) i and j are involved in as many not-yet-treated constraints as possible, d) $w_{i,j}$ is maximal.

The first criterion guarantees that at least one circuit will appear soon during the BAB process. The second and the third ones may promote an earlier appearance of circuits in the following steps. The last one may increase the lower bound in one of the BAB branch below.

In the worst case i.e., where G has no cycle and each vertex has no more than one neighbor, the complexity of this heuristic is majored by $O(n^3)$. In fact, this worst case constrains the algorithm to verify all the above criterions

6.2.3.2 Heuristic 2

In this heuristic, we model the problem by an undirected graph $G = (V, E)$ which is obtained by representing each dis-equation $t_i - t_j \neq d_{i,j}$ by an edge (i, j). At start, edges are not weighted.

We build a basis of cycles of G using a standard spanning tree algorithm. A spanning tree classifies edges in two categories: tree edges and non-tree edges. Each non-tree edge defines, with the tree edges, a unique cycle. For each such cycle $C = (v_1, v_2, ..., v_p, v_1)$, we compute its weight in both directions $v_1, v_2 \rightsquigarrow v_1$ and $v_1, v_p \rightsquigarrow v_1$, giving to the edge (v_i, v_{i+1}) the weight $1 + d_{i,i+1}$ or $1 - d_{i,i+1}$ depending whether the edge is traversed from v_{i+1} to v_i or in the opposite direction. If at least one of these cycle weights is positive, the cycle is chosen.

These positive-weight cycles are sorted in an increasing order according to their length (number of edges). Then, the constraint list is built as follows: the first constraints are the constraints of the first cycle, then we add the constraints of the second cycle, except those already treated in the first one, etc. The list is completed by the constraints that do not belong to any of these positive-weight cycles.

This heuristic mainly uses a spanning tree algorithm and a sorting algorithm. A spanning tree of a given graph can be built thanks to the depth-first search (DFS) algorithm. With a good implementation, DFS algorithm has a complexity in $O(n+m)$. Using a fast sort algorithm (such quicksort or heapsort), the sort can be achieved in $O(q \lg q)$, where q is the cardinal of the found basic cycles set. Thus, the complexity of this heuristic is $O((q \lg q) + (n + m))$.

6.2.3.3 Heuristic 3

Another possibility is to represent each dis-equation by one of its two exclusive arcs. During the BAB algorithm each dis-equation leads to two arcs, one with weight $d_{i,j} + 1$, and the other with weight $1 - d_{i,j}$. We choose to represent each dis-equation by its nonnegative arc. Thus, in the resulting directed graph, all eventual circuits are positive. Then, we do as in heuristic 2, we enumerate circuits. Here, the non-tree edges are classified into forward, across, and back arcs, only the back arcs are part of circuits. Hence, the constraint list is built by the list of constraints composing each circuit followed by the remained constraints.

It can happen that both edges of a particular dis-equation (those whose weight is 0, 1, or -1) are nonnegative. In this case, the problem is to choose one of them. For this, we delay dealing with this kind of dis-equations after building the graph of all others constraints. Once the graph is built, for each particular dis-equation, we add the arc that may create a circuit, if not we choose arbitrarily one of them. The Roy-Warshall's algorithm, which computes the accessibility relation, is used for circuit detection.

Notice that this heuristic considers circuits that are exclusively composed of nonnegative arcs, hence, some positive circuits are ignored.

Similar to heuristic 2, this heuristic, whose core is a spanning tree algorithm and a sorting algorithm, has a complexity in $O((q \lg q) + (n + m))$.

6.2.3.4 Heuristic 4

As mentioned earlier, in a dynamic reordering, one can look for the most useful constraint since we have a lot of information in Floyd's matrix. But a dynamic heuristic is applied at each node of the branch-and-bound tree, and hence may have an exponential additional cost in pathological cases. For this reason, we use a very simple algorithm (Algorithm 4), which has a linear complexity in the worst case and even a constant time in most cases. We simply choose the first constraint that allows pruning (i.e., a subtree below is not constructed). If no such constraint exists, we select the first one in the list.

Algorithm 4: Dynamic Reordering Algorithm.

```
Data: ConstraintList, $a$: Floyd's matrix (of the parent node)
Result: $c$ constraint to add
begin
    for ($c \equiv (t_x - t_y \neq d)$) in ConstraintList do
        if ($a_{x,y} + d \geq 0$) and ($a_{y,x} - d \geq 0$) then
            return("Pruning");
        else
            if ($a_{x,y} + d \geq 0$) then
                return($c$, "Only left subtree", constraint $t_y - t_x \geq 1 - d$);
            else
                if ($a_{y,x} - d \geq 0$) then
                    return($c$, "Only right subtree", constraint $t_x - t_y \geq 1 + d$);
                end
            end
        end
    end
    return(first constraint in ConstraintList, "Construct the two subtrees");
end
```

In the worst case, the complexity of this heuristic is $O(n)$.

6.2.4 Experiments

We have implemented the BAB algorithm with both variants and heuristics of reordering constraints on the same benchmarks. Results are reported in Table 6.1. The third column (nbC) gives the size of the system of constraints (number of dis-equations), the fourth column (Opt.) gives the latency of an optimal schedule, the fifth and the seventh columns (Floyd, Dijk) give the scheduling runtime, without reordering constraints, respectively by the Floyd-based incremental algorithm and the Dijkstra-based incremental algorithm. The effect of the reordering heuristics are presented in the remaining columns. As the Dijkstra-based incremental procedure is better than the Floyd-based incremental algorithm, we give only the runtimes obtained with the heuristics applied to the first one (columns Dijk+H1, Dijk+H2, and Dijk+H3 for heuristic 1, heuristic 2, and heuristic 3), except for the dynamic reordering heuristic (column Floyd+H4), which can be applied only to the Floyd-based incremental algorithm (as it needs Floyd's matrix).

Test	T	nbC	Opt.	Floyd	Floyd+H4	Dijk	Dijk+H1	Dijk+H2	Dijk+H3
css1	4	9	5	0.11 s	0.09 s	< 0,01 s	0.04 s	0.04 s	0.05 s
css11	4	6	4	0.07 s	0.06 s	0.04 s	< 0,01 s	0.04 s	0.04 s
css12	4	9	5	0.06 s	0.06 s	0.06 s	0.04 s	0.04 s	0.06 s
css2	9	23	6	30.37 s	26.59 s	5.01 s	9.23 s	0.90 s	2.74 s
css3	7	36	9	41.26 s	17.55 s	3.42 s	2.97 s	2.4 s	5.67 s
css5	3	7	5	0.05 s	0.05 s	0.05 s	0.03 s	0.04 s	0.04 s
css6	8	7	4	1.04 s	1.00 s	0.14 s	0.11 s	0.11 s	0.14 s
jac1	6	7	6	0.32 s	0.28 s	0.07 s	0.05 s	0.04 s	0.04 s
jac2	6	75	22	54.56 s	35.67 s	9.76 s	18.23 s	10.70 s	1' 10 s
jac3	7	85	19	1' 3 s	49.29 s	8.5 s	7.42 s	9.04 s	2' 41 s
rasm1	3	1	5	0.01 s	0.01 s	0.03 s	0.03 s	0.03 s	0.03 s
wss3	5	7	4	0.1 s	0.10 s	< 0,01 s	0.04 s	< 0,01 s	0.05 s
wss31	5	12	6	0.9 s	0.87 s	0.25 s	0.14 s	0.1 s	0.16 s
wss32	5	6	4	0.11 s	0.10 s	0.05 s	0.03 s	0.04 s	0.05 s
woc1	4	5	5	0.05 s	0.04 s	0.04 s	0.02 s	< 0,01 s	0.04 s
woc2	7	10	4	1.90 s	1.82 s	0.2 s	0.26 s	0.13 s	0.26 s
wss1	4	54	17	1.79 s	1.37 s	0.46 s	0.46 s	0.43 s	1.13 s
wss11	4	49	16	2.07 s	1.39 s	0.46 s	0.32 s	0.62 s	0.99 s
wss2	3	9	9	0.05 s	0.05 s	0.04 s	0.02 s	0.04 s	0.03 s
wss12	4	49	16	2.27 s	1.45 s	0.48 s	0.24 s	0.75 s	1.47 s
wmt22	4	24	13	0.56 s	0.52 s	0.19 s	0.23 s	0.1 s	0.34 s
css21	9	44	10	3h 25' s	1h 20' s	15' 50 s	8' 22 s	2' 43 s	13' 16 s

Table 6.1: Scheduling Results for the Various Tests on the BAB Algorithms.

The analysis of the BAB algorithm runtimes shows that they are sufficiently acceptable in contrast to its high exponential theoretic complexity (except for one pathological case, program *css21*, presented hereafter). In addition, the results confirm the fact that the BAB algorithm, using the Dijkstra-based incremental procedure, is faster than the Floyd-based incremental procedure.

Concerning the reordering heuristics, the results show that heuristic 1 and heuristic 2 do improve the runtime. But it is difficult to choose one among them because there are some compromises; when one improves runtime for part of the cases, it increases the runtime for the other ones. Heuristic 3 has the worst runtime; this result can be explained by the fact that only positive circuits composed exclusively of positive arcs are taken into account while some positive circuits, which are composed by a mixture of positive and negative edges, are not taken into account. The dynamic heuristic (heuristic 4) improves the runtime too, however the BAB algorithm coupled with the Dijkstra-based incremental version is slightly better than this improvement. For the pathological case, the constraints reordering heuristic 2 gives the best runtime.

Table 6.2 presents the results for the BAB algorithms and heuristics when we initialize L_{best}, the best global lower bound during the BAB process, to the latency of the schedule obtained by the GS heuristic. For each algorithm, the results give the percentage of improvement due to this better initialization. Only significative improvements (more than 5%) are given. The results

Test	Branch-and-bound					
	Floyd	Floyd+H4	Dijk	Dijk+H1	Dijk+H2	Dijk+H3
css1	–	–	–	–	–	–
css11	38 %	32 %	–	10 %	11 %	–
css12	23 %	22 %	15 %	13 %	8 %	16 %
css2	5 %	5 %	16 %	22 %	16 %	–
css3	7 %	11 %	–	–	–	–
css5	7 %	–	9 %	16 %	–	–
css6	–	–	–	–	–	–
jac1	14 %	12 %	–	–	18 %	15 %
jac2	–	–	–	–	–	–
jac3	–	–	–	27 %	–	12 %
rasm1	–	–	–	–	–	–
wss3	32 %	34 %	23 %	36 %	20 %	23 %
wss31	18 %	18 %	40 %	12 %	23 %	14 %
wss32	23 %	22 %	12 %	13 %	14 %	11 %
woc1	12 %	8 %	–	–	5 %	6 %
woc2	6 %	6 %	–	9 %	10 %	–
wss1	–	–	–	–	–	–
wss11	10 %	7 %	–	–	50 %	5 %
wss2	7 %	15 %	–	–	18 %	–
wss12	16 %	12 %	–	–	5 %	22 %
wmt22	–	–	–	7 %	–	–
css21	–	–	–	–	–	–

Table 6.2: Improvements on the BAB Algorithm with L_{best} Set to the GS Schedule Latency.

clearly depend on the application.

Concerning the influence of our binding heuristic on the latency and the resolution time, when there are more than one copy of each resource, we have done some experiments. Indeed, on the same example with same resources, we have generated several bindings. Results are reported in Table 6.3. In the first batch of tests (first four tests), we have only changed resource bindings, while in the second batch, we have cut the tasks in smaller and smaller pieces. Results show that changing the bindings have no influence on the latency, and a small influence on the scheduling time. On the other hand, changing the tasks granularity has a small influence on the latency, with large variations on the scheduling time.

Test	wss_1	wss_2	wss_3	wss_4	wss_5	wss_6	wss_7	wss_8	wss_9	wss_10
T	4	4	4	4	5	6	7	8	9	10
nbC	48	50	51	50	75	64	67	64	60	64
Latency	16	16	16	16	15	14	14	14	15	14
Time	0,74s	0,7s	0,78s	0,7s	3,1s	1.3s	7s	1,2s	3,7s	1,3s

Table 6.3: Influence of the Binding Heuristic on Latency and Resolution Time.

Pathological case The pathological case we encountered (program *css21*) has only 9 tasks (but 32 micro-tasks). These independent tasks are taken from the SPICE program (from line 765 to line 773) of the PerfectClub benchmarks. What happens in this test is that all local lower bounds are close to the optimum, so no early elimination is possible, and this causes the total scan of the solution space. The problem is typical of the difficulties one may encounter when scheduling parallel loops. The code is the following:

```
Task 1:   GDPR=VALUE(LOCM+4)*AREA
Task 2:   GSPR=VALUE(LOCM+5)*AREA
Task 3:   GM=VALUE(LOCT+5)
Task 4:   GDS=VALUE(LOCT+6)
Task 5:   GGS=VALUE(LOCT+7)
Task 6:   XGS=VALUE(LOCT+9)*OMEGA
Task 7:   GGD=VALUE(LOCT+8)
Task 8:   XGD=VALUE(LOCT+11)*OMEGA
Task 9:   LOCY=LYNL+NODPLC(LOC+20)
```

Assume that we have one adder, one multiplier, a memory block `VAL` (where the `VALUE` array is mapped) and a memory block `Mdp` (where the `NODPLC` array is mapped) with one port. Assume also that memory access takes 2 cycles and is pipelined, while all the other resources take one cycle. Figure 6.1 diagrams the reservation table for the tasks – type (a) for tasks 3, 4, 5, and 7, type (b) for tasks 1, 2, 6, and 8, and type (c) for task 9 – and the optimal schedule, computed by the BAB scheduler, whose latency is 10. It corresponds to $t_1 = 0$, $t_2 = 1$, $t_3 = 2$, $t_4 = 3$, $t_5 = 4$, $t_6 = 5$, $t_7 = 8$, $t_8 = 6$, $t_9 = 7$. It is never obtained by the GS heuristic in a sample of n^2 permutations.

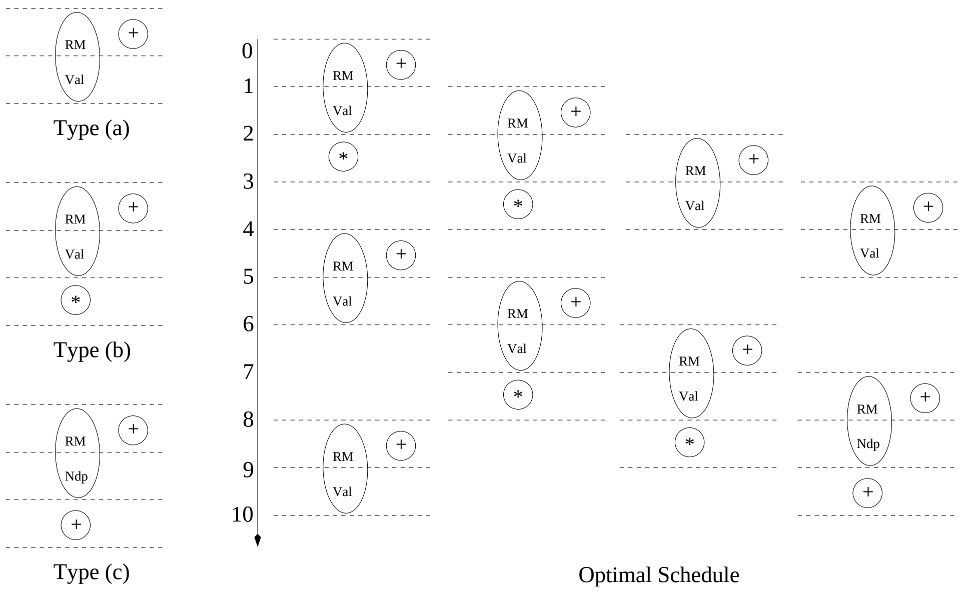


Figure 6.1: Pathological Case css21.

6.3 Conclusion

We have designed a new exact resource-constrained scheduling method in which ILP is replaced by longest-path calculations as tools for a branch-and-bound meta-algorithm. The longest-path computations are accelerated by variants of Floyd's or Dijkstra's algorithms. We have also designed four constraints ordering heuristics that we employed to perform prunings, as soon as possible, in the BAB process. Thus, improving the runtime of the algorithm.

This method is contextually designed to schedule data-independent tasks. However, it can be easily generalized to support problems of resource-constrained scheduling even when tasks are dependent.

Scheduling results show that, in effect, the BAB algorithm has an acceptable runtime at least on our benchmarks. Though it was observed that it can be vulnerable to some rare pathological cases. The use of the constraints ordering heuristics have really improved the runtime of the BAB algorithm; the results have shown that, in most cases, they give better runtime than the original solution.

Chapter 7

Comparative Study

In the previous chapters, we have presented some solutions to the resource-constrained scheduling problem presented in Section 4.3 in which tasks are represented by reservation tables and the resource constraints are modeled by dis-equations. These solutions consist on a greedy heuristic and two exact algorithms. The first uses ILP techniques and the second is based on the branch-and-bound meta-algorithm.

In this chapter, we report and comment some comparative results. The aim of this comparisons is to demonstrate the effectiveness of the proposed methods. We make some additional tests to analyze in more detail the parameters that influence their runtime and thus we give some guidelines for selecting the most effective one according to the context.

7.1 Comparative Results and Discussion

To compare the three methods, the GS heuristic, the BAB scheduler, and the ILP scheduler, we have chosen the best performances of each one. Comparative results are reported in Table 7.1. For the BAB algorithm, we have reported its runtime using the Dijkstra-based algorithm, coupled with the reordering heuristic 2, while setting the initial value of L_{best} to the GS schedule latency (column Dijk+H2). For the ILP techniques, we have selected the method which uses the 0/1 simplified encoding. For the GS heuristic, we have only presented its deviations from the optimum. Indeed, knowing that the GS heuristic is sensitive to the order of the task list, we ran the algorithm on a sample of permutation of tasks. The size of this sample is the square of the number of tasks, and the permutation are random. The maximum deviation (DevMax column) presents the difference between the worst schedule in the sample and the optimum as given by the BAB algorithm. The DevMin column presents the deviation of the best schedule, in the sample of permutations, from the optimum.

The results show that, despite its simplicity, the GS heuristic has a good behavior, at least for these examples: even the latency of the worst schedule (in the sample) is not very far from the optimum. The result in the DevMin column demonstrates that the best schedule is very close to the optimum. Hence one can find a good schedule by applying only GS to a small sample of permutations. At least for our benchmarks, the BAB algorithm is often faster than the ILP technique. However, there are exceptions. Hence, both methods can be useful for practical applications. We made some additional tests to analyze in more detail the parameters that influence their runtimes.

As a rule of thumb, ILP works well whenever the task durations (the p_i) are small and especially when they are all equal to 1. But if one multiplies the duration and resource occupation

Test	T	μT	Greedy scheduling		ILP	Branch-and-Bound	
			DevMax	DevMin	ILP (0/1)	nbC	Dijk+H2
css1	4	15	2	1	0.2 s	9	0.04 s
css11	4	15	2	0	0.22 s	6	< 0,01 s
css12	4	17	3	1	0.21 s	9	0.04 s
css2	9	32	2	1	0.77 s	23	0.75 s
css3	7	27	3	0	0.3 s	36	2.36 s
css5	3	9	0	0	0.17 s	7	0.04 s
css6	8	12	0	0	0.18 s	7	0.11 s
jac1	6	19	0	0	0.13 s	7	0.03 s
jac2	6	82	1	1	2.83 s	75	10.70 s
jac3	7	97	1	0	2.57 s	85	9.0 s
rasm1	3	9	0	0	0.15 s	1	0.03 s
wss3	5	11	0	0	0.18 s	7	< 0,01 s
wss31	5	11	1	0	0.19 s	12	0.09 s
wss32	5	11	0	0	0.16 s	6	< 0,01 s
woc1	4	13	0	0	0.14 s	5	0.03 s
woc2	7	9	1	0	0.16 s	10	0.12 s
wss1	4	44	5	0	1.26 s	54	0.43 s
wss11	4	44	4	1	0.75 s	49	0.30 s
wss2	3	23	1	0	0.62 s	9	< 0,01 s
wss12	4	44	5	1	0.83 s	49	0.71 s
wmt22	4	31	0	0	0.25 s	24	0.1 s
css21	9	32	2	1	0.48 s	44	2' 43 s

Table 7.1: Comparative Results.

of each micro-task by a constant factor f (which means the corresponding resource is non-pipelined and is used during f steps), the complexity of the ILP problem increases dramatically, both in terms of the number of unknowns and of the size of the coefficients because the schedule horizon H increases. In contrast, the BAB algorithm is not particularly sensitive to the involved numbers size but more to the dis-equations number. If each micro-task uses a resource during f steps, we can describe the corresponding resource constraint by a dis-equation expressing a forbidden interval of length f (i.e., the two corresponding inequalities $t_i - t_j \leq d_{i,j} - f$ or $t_i - t_j \geq d_{i,j} + f$). This extension does not increase the BAB algorithm complexity.

Test	$\sum p_i$	ILP	BAB
css3 original	27	0.3 s	2.36 s
css3 (2)	47	0.69 s	0.38 s
css3 (3)	67	0.9 s	0.38 s
css3 (4)	87	1.18 s	0.43 s
css3 (5)	107	1.45 s	0.38 s
css3 (6)	127	1.76 s	0.39 s
css3 (7)	147	2.06 s	0.3 s
css3 (8)	167	2.43 s	0.38 s
css3 (9)	187	2.82 s	0.39 s
css3 (10)	207	3.29 s	0.40 s

Table 7.2: Comparative Results when the p_i Vary.

To demonstrate this effect, we generated variants of the program test *css3* for which the ILP approach was faster than the BAB algorithm. These variants consists in duplicating the tasks twice, four times, etc. The results are given in Table 7.2.

Concerning data dependences, integrating them in the BAB algorithm is almost for free as we just have to plug them as constraints at the root node of the BAB tree. For the ILP approach however, we cannot use the 0/1 simplified formulation anymore as we need the constraints (5.4) to express the dependences. So, in general it takes more time than without dependences. This effect is demonstrated in Tables 7.3 and 7.4. To get the results of Table 7.3, we add a few artificial (i.e., they are not in the initial program) data dependences between the tasks. The ILP approach gets slower as we have to use the 0/1 standard encoding, while the BAB algorithm gets usually faster. Indeed, at each node of the BAB process, more edges need to be traversed (so this should be more costly), but the solution space gets smaller (some task orders are now impossible) and also some subtrees are not searched anymore because their new L_{local} is now larger than the current best evaluation L_{best}.

Test	nb Dep.	ILP		BAB	
		Without dep.	With dep.	Without dep.	With dep.
css2	3	0.77 s	0.8 s	0.75 s	0.74 s
css3	5	0.3 s	0.86 s	2.36 s	0.18 s
css5	3	0.17 s	0.33 s	0.04 s	< 0,01 s
css6	6	0.18 s	0.38 s	0.11 s	0.02 s
jac1	5	0.13 s	0.26 s	0.03 s	< 0,01 s
jac2	5	2.83 s	3.18 s	10.70 s	0.67 s
jac3	4	2.57 s	4.02 s	9.0 s	0.85 s
wss1	4	1.26 s	2.12s	0.43 s	0.07 s
wss11	4	0.75 s	1.14 s	0.62 s	0.09 s
wss12	4	0.83 s	1.44 s	0.71 s	0.06 s
rasm1	2	0.15 s	0.36 s	0.03 s	< 0,01 s
css21	5	0.48 s	1.03 s	2' 43 s	1.59 s

Table 7.3: Comparative Results with Artificial Data Dependances.

One can argue that this comparison is not fair as we should compare with original programs containing actual data dependences. To get such programs, we consider some of our benchmarks and we decompose a few macro-tasks into 2 or 3 data-dependent sub-tasks. The results are given in Table 7.4. The ILP approach still slows down a bit, but now the BAB algorithm slows down too although it remains in general faster than the ILP algorithm for these examples. The reason of this slow-down is that by splitting a task T into two sub-tasks T_1 and T_2, we sometimes increase the number of dis-equations. Indeed, if T is is involved with another macro-task U with two dis-equations combined into one because they have the same forbidden distance, we may now have two different dis-equations to consider: one involving T_1 and U and the other involving T_2 and U. Table 7.4 gives, in addition to the runtimes, the number of corresponding dis-equations. To summarize this study, the BAB algorithm seems to be more suitable when the number of dis-equations is small and when the ILP solver may take too much time because data dependences need to be expressed, for a large schedule horizon H.

Test	T	nb Dep.	old nbC	new nbC	ILP		BAB	
					Without dep.	With dep.	Without dep.	With dep.
css2	12	3	23	26	0.77 s	0.90 s	0.75 s	0.47 s
css3	11	5	36	36	0.3 s	0.98 s	2.39 s	0.71 s
css5	5	2	7	8	0.17 s	0.32 s	0.04 s	< 0,01 s
css6	10	2	7	10	0.18 s	0.36 s	0.11 s	0.12 s
jac1	8	2	7	9	0.13 s	0.28 s	0.03 s	0.14 s
jac2	9	3	75	93	2.83 s	3.28 s	10.70 s	1' 16 s
jac3	10	4	85	107	2.57 s	4.02 s	9.0 s	11.03 s
wss12	7	3	49	71	0.83 s	2.26 s	0.71 s	1.68 s
css21	12	3	44	49	0.48 s	1.21 s	2' 43 s	3' 34 s

Table 7.4: Comparative Results when Splitting a few Macro-tasks.

7.2 Conclusion

In this first part of the thesis, we presented a formalism, for HLS scheduling, to accurately express resource constraints for complex tasks represented as reservation tables. The resource constraints are modeled by dis-equations and finding an optimal schedule entails resolving a system of dis-equations. The proposed formalism can be applied to problems of resource-constrained scheduling where tasks may be linked by data dependences.

We have proposed some solutions for scheduling such tasks: a greedy heuristic and two exact algorithms. The first use ILP technique and the second is based on the branch-and-bound meta-algorithm. Scheduling results show that, in effect, the greedy heuristic has a suitable behavior. Similarly, the BAB algorithm has an acceptable runtime but can be vulnerable to some rare pathological cases. For improving the runtime of the BAB algorithm, we have designed four constraints ordering heuristics. The results have shown that, in most cases, they give better runtime than the original solution. Compared to the ILP technique, the BAB algorithm has shown better behavior when tasks execution times are large.

Part II

Resource-Constrained Scheduling using Graph Coloring

Chapter 8

Scheduling via Branch-and-Bound-Based Graph Coloring

In the first part of this thesis, we have developed some scheduling solutions that are integrated in a three-step scheduling approach (§ 3.3.2). However, one can emphasize that the partitioning into steps 2 and 3 in the three-step scheduling approach seems artificial. In other words, one can ask: “is it a good methodology to first "micro-schedule" each macro-task independently and then, schedule them together ?”. For this reason, in this chapter, we try alternatively to formalize the problem so that the steps 2 and 3 are performed in a unified way. Using this formalism, we design a novel exact scheduling algorithm in which an optimal schedule is computed by properly coloring the graph that represents both resource constraints and data dependencies. Some commented experiences illustrate the effectiveness and the efficiency of this scheduler.

8.1 Formalisms

8.1.1 Task Model

Conversely to the model task of the first part, where a task was a sequence of micro-tasks (elementary operations), here, we consider directly all the elementary operations of the macro-tasks. So a task i is an elementary operation such as an addition, a multiplication, a shift.... We assume that this elementary operation is already mapped on the available resource. Here also, a simple binding is used: each functional operation is mapped to the first free resource, resources are allocated on a cyclic way. In what follow, for the sake of simplicity, we use task instead of elementary operation or micro-task.

We denote by T the set of tasks which represents the set of the DFG (Data Flow Graph) nodes, R the set of resources, t_i the starting date of the task i, each task latency is assumed to be one “unit” (the unit is the clock cycle). As will be seen in what follows, this assumption allows us using easily a graph coloring model.

However, this hypothesis should not preclude the multi-cycle and/or pipelined functional units use. In fact, we allow such resource features by the following tricks. When a task is mapped to a multi-cycling resource which delay is more than one cycle, there are two possible scenarios:

1. **the resource can be pipelined:** here, we add supplementary operations *nop* –fictitious operation as they do not use real resource– to T;

2. **the resource doesn't allow pipelining:** we add supplementary operations *nop* to T but here they are not fictitious as they do use the resource.

In both situations, some "false data dependences" have to be added to the DFG in order to preserve the data-flow dependences constraints.

Let us illustrate these tricks by means an example in which a functional operation *mult* is mapped to a three-cycle multiplier, as described in Figure 8.1 (a). If the multiplier can be pipelined then we add two *nop* tasks to T, we add also two data dependences as in Figure 8.1 (b) and all successors of *mult*, in the original DFG, become successors of the last *nop* operation. If the multiplier is a simple resource, i.e., it can't be pipelined, similar additions are performed but now the *nop* operations are bounded to the multiplier resource i.e., they are assumed using the multiplier (see Figure 8.1 (c).

This simulation validity imposes us to guarantee that each *nop* operation will be exactly scheduled 1 cycle after its direct predecessor node. We will see that it is possible to perform.

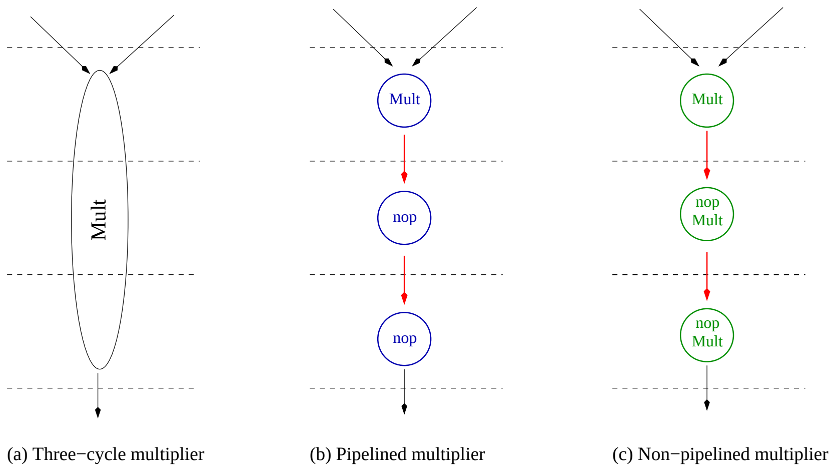


Figure 8.1: One unit-cycle resource simulation.

8.1.2 Data Dependences and Resource Constraints Formalism

Finding legal and optimal schedules for T entails more precision into the way of expressing resource constraints and data dependences between its tasks. In the following, we will express uniformly both kinds of constraints, using dis-equations. Here, by dis-equation we mean "a negation of equation where the second member is null", conversely to what we have defined in the first part, where it is the most general definition, "a dis-equation is a negation of equation").

Let i and j be two tasks and t_i and t_j their respective starting dates. First, in a valid schedule, i and j can start at any date except those which put them into a resource conflict or data dependence conflict. Indeed, if a resource r is used by both i and j, then t_i and t_j have to take different values. Thus, the intuitive idea is to express resource constraint by:

$$t_i - t_j \neq 0.$$

Similarly, a data dependence constraint between i and j can also be expressed with a dis-equation. Indeed, when the task j depends on the task i it implies that the task j must be executed after the task i. This can be expressed by the constraint $t_j - t_i \geq 1$ or more general by $t_j - t_i \geq \delta_{i,j}$ (classical precedence constraints). Here again, t_i and t_j have to take different values, thus yielding the inequality $t_j - t_i \geq 1$ as $t_j - t_i \neq 0$. It should be noted that solutions for $t_j - t_i \geq 1$ are included in the set of solutions of $t_j - t_i \neq 0$. Thus, replacing $t_j - t_i \geq 1$ inequality by the dis-equation $t_j - t_i \neq 0$ represents a relaxation that we have to compensate when a solution is found to guarantee the schedule validity. We will return to this fact later.

It follows that, for the set T of tasks, 1) all the resource constraints can be expressed by defining for each couple of tasks (i, j) the dis-equation expressing the resource constraint, if i and j share the same resource, 2) all the data dependences are expressed by defining for each couple of tasks (i, j) the dis-equation expressing the data dependence constraint if i and j are linked by a data dependence.

8.2 Scheduling Problem Formulation

Using this formalism, finding a schedule for T entails solving the following system of dis-equations on integer values:

$$\left\{ \; t_i - t_j \neq 0 \qquad i, j \in T \wedge ((i, j \text{ share a resource}) \vee (i, j \text{ are linked by a data dependence})) \right. \tag{8.1}$$

then choosing one solution, among the set of solutions, which respects precedence constraints. Indeed, the previous relaxation, in which we have replaced $t_j - t_i \geq 1$ by $t_j - t_i \neq 0$, have to be verified.

First, let us mention that this system is usually feasible; it has at least one solution, the solution corresponding to the sequential execution order.

In addition, these dis-equations can be represented by an undirected graph $G = (V, E)$, where an edge between two tasks i and j means that i and j cannot be scheduled at the same time. G is easily built by merging both graphs G^r and G^d, where G^r represents the interference graph (or conflict graph); there is an edge between i and j if they share a same resource and G^d represents the graph obtained by performing a transitive closure on the DFG and replacing all directed edges by undirected ones. The transitive closure operation guarantees that all data dependences, implicit and explicit ones, will be expressed by an edge. Indeed, the edges in the DFG express only explicit data dependences, hence implicit data dependences, expressed by any path $i \rightsquigarrow j$, have to be expressed by dis-equation $t_j - t_i \neq 0$. Transforming all directed edges in G^d by undirected ones, here also, is a relaxation which must be compensated when a solution is found to guarantee that the computed schedule is valid. We will return to this fact later.

Let us mention that G can have any structure. We denote by $n(G)$ the cardinal of V which represents the task number (i.e., the cardinal of the set T) and m the number of distinct dis-equations. As formalized, it is easy to see, that finding a schedule for these tasks entails properly coloring[24] the graph G, then establishing an order on colors which respects the precedence constraints.

Further, for getting an optimal schedule, we have to minimize the colors number needed for properly coloring G. Thus, finding the chromatic number [25] $\chi(G)$ of G.

[24]In graph theory, let us recall that "graph coloring" is an assignment of "colors" to certain objects in a graph subject to certain constraints. Here, we use its simplest form which is a way of coloring the vertices of a graph such that no two adjacent vertices share the same color, called a "vertex coloring".

[25]The chromatic number $\chi(G)$ of G is the smallest number of colors needed to properly color G.

8.3 How To Color a Graph?

First, let us mention that there are several practical problems that are modeled by graph coloring. Although graph coloring takes its name from the map-coloring application, it enjoys several theoretical challenges. Beside the classical types of problems, the problem of coloring a graph has found a number of applications especially register allocation in compilers, scheduling, frequency assignment in mobile radios and pattern matching.

Specially, graph coloring and its generalizations are useful tools in modeling a wide variety of scheduling and assignment problems, such as precoloring extension, list coloring, multicoloring, minimum sum coloring.... More details on these applications in scheduling are given by Marx in [76].

Unfortunately, optimally coloring a graph i.e., determining its chromatic number is an $\mathcal{NP}$-Complete problem (Chap 5 of [110]). Nevertheless, there are many methods for coloring a graph G, or solving the system defined in (8.1):

- one can be satisfied with greedy coloring heuristics where coloring is done vertex by vertex, the order of the vertex coloring can be established statically as well as dynamically;
- We can also use meta-heuristics such as simulated annealing, Tabu search, and genetic-based algorithms;
- for optimality, some solutions from operations research are availables, they can be based on:
  - *Coloring* using a Branch-And-Bound meta-method;
  - *Integer Linear Programming* techniques [99].
  - Since there is an obvious bound for the $\chi(G)$ ($\chi(G) \leq n$), we can also use finite domain constraint satisfaction programming [14].

As discussed earlier, Integer Linear Programming and Constraint Logic Programming are the main alternative approaches used to solve combinatorial optimization problems. They have shown their ability to locate and prove the existence of an optimal solution in rigorous ways. Unfortunately, these methods are rather slow.

Similarly, the *Branch-And-Bound* method has an exponential theoretical complexity, but conversely, it can have a better behavior in practice if it is well instrumented. Indeed, *Branch-And-Bound* is a meta-method of guidance in the space of solutions, its resolution strategy depends strongly on the problem features to resolve. Hence, in what follows we will design an exact coloring graph by mean of Branch-And-Bound technique.

Example

Let us consider the same example seen in Section 4.2.4

```
Task 1:   GSPR = VALUE(LOCM+2)*AREA
Task 2:   GEQ  = VALUE(LOCT+2)
Task 3:   XCEQ = VALUE(LOCT+4)*OMEGA
Task 4:   LOCY = LYNL+NODPLC(LOC+13)
```

With the same assumptions: the available resources are one adder, one multiplier, and two memory blocks: `Val` (where the `VALUE` array is mapped) and `Ndp` (where the `NODPLC` array is mapped). Assume also that both memory access and multiplication take two cycles and that both can be pipelined. Scalar variables like `AREA` or `LOCY` are assumed to be allocated to registers, where they can be accessed in no time.

Figure 8.2 diagrams the DFG of the macro-tasks with one possible binding, where the label `RM Val` (resp. `RM Ndp`) means to read the memory block `Val` (resp. `Ndp`). Here we chose the binding that greedily allocates all the available resources to tasks, i.e., we assign all the resources to the functional operations of each task. Let us note that the fictitious operations *nop* are added to satisfy the 2-cycle resources (multiplier and memory blocks). Operations 1 to 17 are the elementaries tasks.

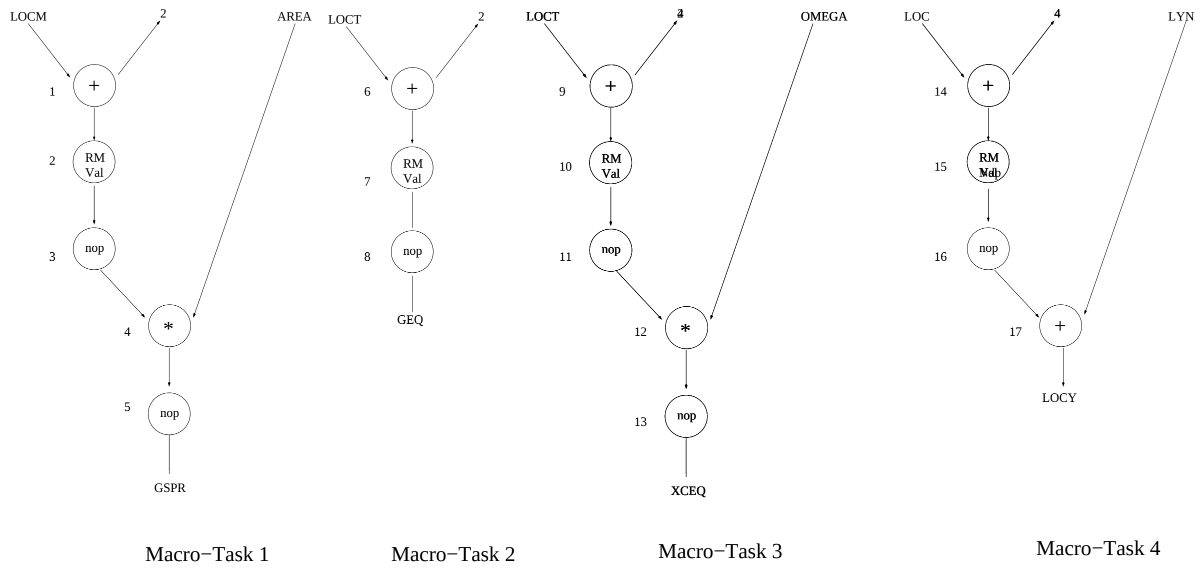


Figure 8.2: Binding for the Example.

For this example, the resource constraints system is composed of 15 constraints defined as follows:

$$\begin{cases} t_1 - t_6 \neq 0 & t_1 - t_9 \neq 0 & t_1 - t_{14} \neq 0 & t_1 - t_{17} \neq 0 & t_2 - t_7 \neq 0 \\ t_2 - t_{10} \neq 0 & t_4 - t_{12} \neq 0 & t_6 - t_9 \neq 0 & t_6 - t_{14} \neq 0 & t_6 - t_{17} \neq 0 \\ t_7 - t_{10} \neq 0 & t_9 - t_{14} \neq 0 & t_9 - t_{17} \neq 0 & t_{14} - t_{17} \neq 0 \end{cases}$$

For instance, the constraint $t_1 - t_6 \neq 0$ expresses the fact that the operation 1 and 6 cannot start at the same time because, among other reasons, both use the adder. This system of constraints is used to build the conflict graph G^r.

Explicit data dependence constraints are extracted from the DFG. For example the data dependence between the operation 5 and 4 defined by the inequality $t_5 - t_4 \geq 1$ will be replaced by the constraint $t_5 - t_4 \neq 0$ and so on. We obtain the following system of direct data dependences constraints, which is used as a basis to construct the data dependences graph G^d:

$$\begin{cases} t_5 - t_4 \neq 0 & t_4 - t_3 \neq 0 & t_3 - t_2 \neq 0 & t_2 - t_1 \neq 0 & t_8 - t_7 \neq 0 \\ t_7 - t_6 \neq 0 & t_{13} - t_{12} \neq 0 & t_{12} - t_{11} \neq 0 & t_{11} - t_{10} \neq 0 & t_{10} - t_9 \neq 0 \\ t_{17} - t_{16} \neq 0 & t_{16} - t_{15} \neq 0 & t_{15} - t_{14} \neq 0 \end{cases}$$

8.4 Branch-and-Bound-Based Graph Coloring Solution

Let us recall that *Branch-And-Bound* is an implicit enumerative meta-method which searches, in the solution space, a solution according to an objective function. Its resolution strategy depends strongly on the feature of the objective function and the quality of the lower and upper bounds used for pruning.

Let $G = (V, E)$ be the graph which formalizes the dis-equation system given by (8.1). Let $n(G) = |V|$ be the number of tasks and $m = |E|$, the number of dis-equations.

In our context, let us recall that we consider only valid colorings (schedulable ones) of G. A coloring is valid if we can get an order on colors such that no precedence constraint is violated. Indeed, the precedence constraints can be violated due to the previous relaxations, in which we have replaced $t_j - t_i \geq 1$ by $t_j - t_i \neq 0$ and replaced an arc by an edge in G^d. In other words, a coloring is valid if we can build a valid schedule from this coloring.

Before explaining our algorithm strategy, first let us explain how the branching is done, which lower bound is used and how it is evaluated.

8.4.1 Branching Rule

The branching rule used here is inspired from the idea of Béla Bollobàs (Chap. 5 of [13]) for coloring any graph[26]. The idea is based on coloring a graph G by reducing the problem to coloring two other graphs derived from G. Let u and v be nonadjacent vertices of a graph G. Let G' be obtained from G by joining u and v, and let G'' be obtained from G by identifying (merging) u and v. Thus, in G'' there is a new vertex (uv) instead of u and v which is joined to vertices adjacent to at least one of u and v (see Fig. 8.3).

These operations are even more natural if we start with G': then G is obtained by cutting the edge (u, v), and G is obtained from G'' by exploding the vertex (uv).

This separation guarantees that we are not losing any solution. Indeed, let us note that colorings of G' and colorings of G'' are disjoint sets, because colorings of G' give u and v different colors and colorings of G'' give them the same color. In addition, the colorings of G in which u and v get distinct colors are in 1-to-1 correspondence with the colorings of G'. Indeed, $c: \quad V \rightarrow \{1, 2, \ldots, k\}$ is a coloring of G with $c(u) \neq c(v)$ iff c is a coloring of G'. Similarly, the colorings of G in which u and v get the same color are in 1-to-1 correspondence with the colorings of G''. In particular, if for a natural number x and a graph H, we denote $p_H(x)$[27] for the number of colorings of a graph H with x colors, then

$$p_G(x) = p_{G'}(x) + p_{G''}(x).$$

By definition $\chi(G)$ is the least natural number k for which $p_G(x) \geq 1$. Thus, these remarks imply that $\chi(G)$ can be defined as:

$$\chi(G) = \min\{\chi(G'),\, \chi(G'')\} \tag{8.2}$$

8.4.2 Evaluation -Bounding- Procedure

Many upper and lower bounds for the chromatic number are proposed in the literature [13, 111]. First, let us consider the upper bounds. Most upper bounds come from algorithms that produce

[26] This Divide-to-Conquer technique is originally designed to get some information about the number of colorings of a graph with a given set of colors.

[27] Called the chromatic polynomial.

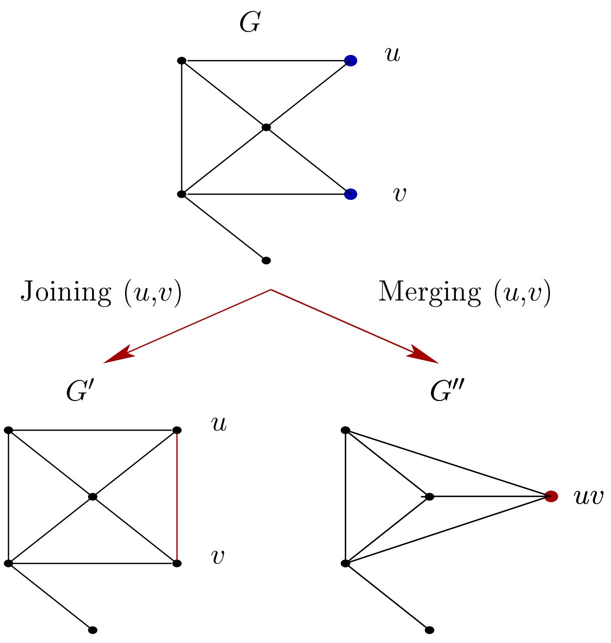


Figure 8.3: The Graphs G, G' and G''.

colorings. For example assigning distinct colors to the vertices yields $\chi(G) \leq n$. This coloring uses nothing about the structure of G; we can do better by coloring the vertices in some order. For example, a *greedy coloring* relative to a vertex ordering $v_1, \ldots, v_n$ of V can be obtained by coloring vertices in the order $v_1, \ldots, v_n$, assigning to v_i the smallest-indexed color not already used on its lower-indexed neighbors. Each vertex has at most $\Delta(G)$ neighbors, so the greedy coloring cannot be forced to use more than $\Delta(G)+1$, this is the worst upper bound that a greedy coloring could produce ($\chi(G) \leq \Delta(G) + 1$) colors.

Welsh-Powell [109] proposed another greedy coloring, in which they apply the previous greedy coloring to the vertices in non-increasing order of degree $d_1 \geq \ldots \geq d_n$, when we color the ith vertex v_i, it has at most $\min\{d_i, i-1\}$ earlier neighbors, so at most this many colors appear on its neighbors. Hence, the color we assign to v_i is at most $1 + \min\{d_i, i-1\}$. This holds for each vertex. So, we obtain an upper bound; we maximize over i to obtain the upper bound on the maximum color used: $\chi(G) \leq 1 + \max_i \min\{d_i, i-1\}$.

For our coloring algorithm (see below) the most important bounds for the chromatic number used are the lower ones. Let us quote that for any graph G, we have:

- the most known lower bound:
$$\chi(G) \geq \omega(G), \tag{8.3}$$
where $\omega(G)$ is the clique number: the size of the largest set of pairwise adjacent vertices in G (*maximal clique*).

- and a second lower bound:
$$\chi(G) \geq n/\alpha(G), \tag{8.4}$$
where $\alpha(G)$ is the independence number: the cardinal of the largest set of vertices in V so that no two vertices are adjacent (*maximal size of an independent set*).

The first bound holds because vertices of a clique require distinct colors. The second bound holds because in a proper coloring the set of vertices of each color is an independent set and thus has at most $\alpha(G)$ vertices. Both upper bounds are exact when G is a complete graph. However, for certain graphs inequalities (8.3) and (8.4) may be very weak; it can happens that $\omega(G)$ can be much smaller than $\chi(G)$. But according to our context, we rely on the particularity of the built graphs (not very large and high probably may be very connected).

Unfortunately again, in general both maximal clique problem and maximal independent set problem are $\mathcal{NP}$-Hard [110], although good approximation algorithms can be found in [58].

However, a clique computation, as a lower bound to the clique number, can be obtained by several methods:

- one can be satisfied with a greedy clique heuristic. For example, that which we use, build a clique C progressively as follows: we start with the vertex which has the maximal degree, then we add, as long as there is, the vertex with a maximal degree and which is adjacent to all vertices in C. Thus, we obtain in a polynomial time a lower bound such that: $\chi(G) \geq \omega(G) \geq |C|$.

- for optimality, one can use Integer Linear Program formulation. Indeed, knowing that computing the clique number entails computing the independence number for $\bar{G}$, the complementary graph, since the maximal clique problem is complementary to the maximal independent set problem. This fact allows us to use the following natural ILP formulation of the Independant Set problem (IS):

$$\text{IP2:} \left\{ \begin{array}{lll} \max & \sum_{i=1}^{n} x_i & \\ \text{subject to:} & x_i + x_j \leq 1 & \text{for every edge } (i,j) \text{ in } \bar{G} \\ & 0 \leq x_i \leq 1 & \qquad (i = 1, \ldots, n) \end{array} \right.$$

  where x_i are binary variables, $x_i = 1$ if the vertex i belongs to the maximal independent set of $\bar{G}$.

One can argue that computing an exact clique may be very expensive, as it uses ILP formulation, this is true in general, but let us mention that this last formulation is known as binary integer program IP2[28] or 2-SAT[29] which has some powerful properties than a general IP problem. Indeed, it turns out that solutions of this IP2 problem always have denominators not greater than 2, which guarantees that in the process of an integer resolution, no number explosion will be occur. In addition, this property guarantees that all basic solutions of linear relaxation of this 2-SAT are integer multiples of 1/2 (Chap. 3 of [58]).

This property follows from the following statement: the determinants of all nonseparable submatrices of the 2-SAT linear programing problem have absolute value of at most 2. A matrix is nonseparable if there is no partition of the columns and rows to two subsets (or more) C_1, C_2 and R_1, R_2 such that all nonzero entries in every row and column appear only in the submatrices defined by sets C_1XR_1, C_2XR_2.

Proof. Let A denote the constraint matrix of this 2-SAT integer program. Thus, A has at most non-zero entries in every row . Let's do it by induction on the size of the submatrix. Since the entries of A are from $\{-1, 0, 1\}$, the claim holds for $1X1$ submatrices. Assume that it holds for

[28] Integer Programming with two variables per inequality.

[29] 2-satisfiability boolean formula on variables $x_1, \ldots, x_n$ where the objective is to find an assignment satisfying all clauses such that $\sum_{i=1}^{n} x_i$ is maximized.

any $(m-1)X(m-1)$ submatrix and we show that the claim holds for any mXm submatrix. Let A_{ij} denotes the submatrix obtained by deleting the i'th row and the j'th column from A. Without loss of generality, we assume that the two non-zero elements in row i of A are in columns i and $i+1$ (modulo m). Due to the nonseparability of the matrix, this can be achieved by appropriate row and column interchanges, thus:

$$det(A) = A[1,1].det(A_{11}) - (-1)^m A[m,1].det(A_{m1})$$

The absolute values of A_{11} and A_{m1} determinants are equal to 1, since both are triangular matrices with nonzero diagonal elements. Therefore, the A determinant absolute value is at most 2. □

8.4.3 Algorithm

According to these Branch and Bound procedures, we have designed a BAB algorithm which progressively builds a tree of subproblems as follows:

- At the root, we start with the original graph G, which is obtained by expressing both resource constraints and data dependencies between tasks of T;
- At each tree structure node N , we get two nonadjacent vertices and we branch using the previous branching rule.
- During the resolution process, we maintain L_{best}, the the best schedule latency computed so far which corresponds to the number of colors needed by a valid coloring of G. At the beginning, we can set L_{best} to one of the $\chi(G)$ upper bounds, previously seen; thus set to $1 + \max_i \min\{d_i, i-1\}$ or $\Delta(G)+1$.
- At each node N, we treat G^N, the graph obtained by the branch operation. Except for the leaves, we compute clique number (or a greedy clique which is a lower bound to the clique number) L_{local} of G^N. As seen above L_{local} is a lower bound of $\chi(G^N)$ and so of G. If $L_{\text{local}} \geq L_{\text{best}}$ the subtree below N is not constructed as it will not lead to a better complete solution.
- A leaf is reached if there is no nonadjacent vertices in the obtained graph G^l, which means that the graph is complete. It is well known that for such complete graph we have $\chi(G^l) = \Delta(G^l)+1 = \omega(G^l) = |V^l|$. So now we have an actual solution. We check if this coloring is valid; so no precedence constraints is violated, then if it is better than L_{best} then L_{best} is updated.
- The algorithm stops when all branches are explored; Thus, whole solution space has been explored and L_{best} is returned as the optimum solution which satisfied the objective function.

It is easy to prove that this algorithm terminates. Indeed, when we branch, there are two possible situations. In the first one, we merge two vertices so we decrease the vertex number of the graph, no more than $(n-1)$ merges are possible for a given G. In the second situation, we connect two vertices, at most $k = (n^2/2 - m)$ additions are allowed, where k is the number of pairs of vertices not connected in G; it represents the number of edges to be added to G for being a complete graph. Thus, the algorithm terminates and the depth of the BAB tree is at most $\max(k, n-1)$; these values correspond to the length of the extremal branches.

In this algorithm variant, we wait until a solution is found (reaching a leaf) for checking the computed coloring validity, thus the schedule validity. Another possibility can be considered as this test can be dynamically performed. Indeed, a coloring can't be valid if a contraction of two nonadjacent vertices, i and j causes a creation of a cycle in the original DFG (which expresses the precedence constraints). For this reason, we can guard a contraction by a test in which we check if no path, in the DFG, join between i and j (in both directions so both paths $i \rightsquigarrow j$ and $j \rightsquigarrow i$ are considered). This fact entails maintaining a Roy-Warshall's matrix[30] after each contraction. This solution can improve considerably the algorithm runtime by performing soon the prunings. In contrast, it may slow down the algorithm as maintaining a Roy-Warshall's matrix requires q^2, where q is the cardinal of the graph vertex set defined at each level of the BAB tree.

In addition, when a coloring is computed, other conditions have to be verified. In fact, the one unit-cycle resource simulations, which are performed at the beginning in order to allow multi-cycle resource use, have to be verified. Thus, an optimal schedule will be rejected if it does not verify that each *nop* operation is exactly scheduled 1 cycle after its direct predecessor node.

Further, let us also mention that for this algorithm version, the choice of the couple of nonadjacent vertices is random, or rather it is greedy; i.e., we get the first two nonadjacent vertices. One can speed up the algorithm runtime by getting the most adequate two nonadjacent vertices to perform the branching.

8.4.4 Complexity

It is difficult to give the actual complexity of any BAB algorithm except perhaps for the worst cases, where no elimination are done. Specially, for our algorithm when it is coupled with the maximal clique computation algorithm which is based on an integer linear program. For this reason, here, we just try to bring some information which can give an idea on the complexity of the designed algorithm.

First, let us notice that any branch length from the root to a leaf is variable and has a value in the interval $[\min(n-1,k), \max(n-1,k)]$, where $k = (n^2/2 - m)$ is the number of pairs of vertices non connected in G. As explained above, these values correspond to the length of the extremal branches of the BAB tree. Indeed, not more than $(n-1)$ merges are possible for a given graph to become complete and not more than $k = (n^2/2 - m)$ joinings are allowed in a given graph to become complete. The interval $[\min(n-1,k), \max(n-1,k)]$ is included in $[n-1, n^2/2)]$.

On the other hand, the variant of the greedy clique heuristic, which we use, is a pseudo-polynomial heuristic, as its time complexity is $O(n\Delta(G))$, where for $\Delta(G)$ is the degree of G.

8.5 Integer Linear Program Solution

To compare with the previous BAB approach, we design another exact algorithm based on an integer linear program. Indeed, as seen in Chapter 5, our scheduling problem can be formalized by mean of standard coding techniques.

We use the following notations: $x_{i,j}$ is a binary variable associated with task i where $x_{i,j} = 1$ if and only if task i is scheduled at the jth clock cycle. The indices j go from 0 to H, a maximal "horizon" for the schedule. The variable t_i is the starting date of task i, R the set of available resources, R_r the set of tasks that use the resource r, and $d_{i,r}$ the time step [31] (relative to the

[30]A Roy-Warshall's matrix is a boolean matrix which reports the accessibility relation [108]; an entry (i, j) = True in this matrix, if there is a path from i to j in G.

[31]It is possible that, in the same task i, a resource r is used in more than one micro-task. Again, for simplicity, we assume that each task uses each resource at most once, but this may be easily generalized.

beginning of the task) at which task i uses resource r.

Let us recall that a standard way of expressing our scheduling problem is the following. Fix H, the maximal schedule horizon, to an upper bound for the optimal latency. For example, fix H to $card(T)$, which correspond to the sequential execution order latency. Then, minimize the schedule latency L subject to the following constraints (in addition to the fact that all variables are integers and the $x_{i,j}$ are 0/1 variables):

$$t_i = \sum_{j=0}^{H-p_i} j * x_{i,j} \qquad \forall i \in [1 \dots n] \tag{8.5}$$

$$0 \le t_i \le L - p_i \qquad \forall i \in [1 \dots n] \tag{8.6}$$

$$\sum_{j=0}^{H-p_i} x_{i,j} = 1 \qquad \forall i \in [1 \dots n] \tag{8.7}$$

$$\sum_{i \in R_r} x_{i,t} \le 1 \qquad \forall r \in R,\ \forall t \in [0 \dots H] \tag{8.8}$$

$$t_j - t_i \ge 1 \qquad \forall \text{ edge } (i,j) \in \text{DFG} \tag{8.9}$$

The n equalities in (8.5) define the starting dates t_i as functions of the $x_{i,j}$ binary variables. The inequalities (8.6) express the latency to be minimized. For each task i, the equality (8.7) guarantees that i is executed exactly once. Finally, the inequalities (8.8) express resource constraints for each resource $r \in R$. Once the variables t_i are available – through the constraints (8.5), the dependence constraints are naturally expressed through the constraints (8.9). These last inequalities can also be replaced by general precedence constraints: $t_j - t_i \ge \delta_{i,j}$.

8.6 Experimental Results and Discussion

We implemented the algorithms presented previously, in our framework, on the same benchmarks and on the same machine. We also implemented the BAB algorithm with both clique computation variants (maximal clique and greedy clique). Results are reported in Table 8.1. In these first experiments, only 12 test programs, among those used previously, are treated.

In the first two columns of Table 8.1, we report the name of the test and the number of included tasks. The third column represents the chromatic number; so the optimal schedule. For each evaluation procedure variant, we report the runtime of the BAB algorithm and the number of nodes (Nb nodes) actually constructed by the BAB algorithm. The Eighth column represents the ILP technique runtimes.

Experimental results show that, in effect, our exact branch-and-bound approach has an acceptable runtime despite its theoretical complexity. However, the results show that the version of the BAB using the greedy clique algorithm is more faster than the one using the maximal clique computation. This is due to the large number of ILP solver calls, this number can be quantified by the corresponding "Nb nodes" column.

Concerning comparison, except for few cases, the results show that our BAB algorithm has a better behavior than the ILP technique.

8.7 Conclusion

Given a set of tasks linked by some data dependences, under resource constraints, we formalize the problem that finding an optimal schedule lead us to properly color their conflict graph. Indeed,

Test	nbT	Optimum	*Branch-and-bound*				ILP formulation
			Maximal Clique (IP2-Pip)		*Greedy Clique*		
			Time	Nb nodes	Time	Nb nodes	
css1	15	5	3,54 s	343	1.09 s	2200	0.9 s
css11	15	4	0.40 s	42	0.02 s	88	0.6 s
css12	17	10	5. s	5870	1.14 s	1505	0.83 s
css5	9	4	5.39 s	233	0.2 s	49	0.28 s
css6	12	4	1 s	13	0.07 s	35	0.32 s
wss3	11	5	1.32 s	104	0.05 s	360	0,51 s
wss32	11	4	1.63 s	235	0.07 s	482	0,4 s
woc1	13	5	0.08 s	8	0.01 s	73	0,62 s
woc2	9	4	0.03 s	4	0.02 s	44	0,27 s
rasm1	9	5	0.90 s	19	0.03 s	22	0.24 s
rasm2	7	4	0.15 s	2	0.01 s	6	0.15 s
jac1	19	6	1' 08 s	1025	1.98 s	1678	2.1 s

Table 8.1: Scheduling Results for the BAB with both Maximal and Greedy Clique Bounding Algorithms and the ILP.

we have accurately and uniformly expressed both resource constraints and data dependences using dis-equations.

Conversely to classic graph coloring algorithms, we design an exact algorithm in which the coloring is done by means a branch-and-bound meta method [28]. Each evaluation is accelerated by either maximal or greedy clique computation.

Results show that, in effect, our exact branch-and-bound approach has an acceptable runtime despite its theoretical complexity. However, the BAB version in which it is coupled with the greedy clique algorithm is faster than the one using a maximal clique computation.

Furthermore, the effectiveness of this algorithm is also proven by some comparative experiments. We compare the designed algorithm to a classical ILP solution. In fact, except for few cases, the results show that our BAB algorithm has a better behavior than the ILP technique.

Besides, the results show that this simple designed algorithm variant, in our context instrumented to compute an optimal schedule, deserves more attention as it can be used as a solution to many graph coloring problems in others contexts. Indeed, there are several interesting real-life applications that are modeled by graph coloring. In fact using a fast implementation and many others tricks: getting the most adequate two nonadjacent vertices to perform the branching or improving the greedy clique computation heuristic may considerably reduce the algorithm runtimes.

Chapter 9

General Conclusion and Future Directions

Scheduling is one key process in HLS. Under resource constraints, scheduling operations while minimize the total duration is an $\mathcal{NP}$-complete problem since too many constraints and objectives interact. These considerations lead to the idea of designing gradual approaches to schedule programs especially with loops.

Indeed, in order to efficiently exhibit and exploit parallelism we use a symbolic scheduling algorithm. This produces a sequence of logical steps, each of which contains a pool of macro-tasks (with no loops), where each macro-task is a complex sequence of elementary operations. Despite that many hardware constraints can be roughly taken into account by this scheduling pass, it remains that this symbolic scheduling technique is quite complex and cannot take into account all the operations and the architectural resources they need. Thus, we need at least another step to schedule locally all operations of the macro-tasks belonging to the same logical step to satisfy the resource constraints and data dependencies between the elementary operations of the same macro-task. In this work, we have investigated many possibilities.

9.1 Contribution

In a first one and for complexity reasons, we have again divided the problem into two subproblems: 1/ mapping and scheduling each macro-task independently taking into account all peculiarities of the target architecture. This produces a reservation table for each macro-task. 2/ refining each logical step by scheduling all its macro-tasks, now represented by the reservation tables, while respecting resource constraints.

In another possibility, we have simultaneously scheduled all operations of the macro-tasks while satisfying both resource constraints and data dependencies between the elementary operations of the same macro-task.

In this thesis and for both approaches, we have proposed some solutions to certain scheduling steps. Indeed, for the three-step approach, we have focused on the third scheduling step problem: *"scheduling tasks whose resource usage is described by reservation tables"*. First, we have presented a formalism that accurately expresses resource constraints for complex tasks represented by reservation tables. The resource constraints are modeled by dis-equations and finding an optimal schedule lead us to resolve a system of dis-equations. In a second time, we have proposed some solutions for scheduling such problem: a greedy heuristic and two exact algorithms. The first uses ILP techniques and the second is based on the branch-and-bound meta-algorithm.

The last one represents our main algorithmic contribution. It consists on a new exact resource-constrained scheduling method in which classic ILP is replaced by longest-path calculations as tools for a branch-and-bound meta-algorithm. The longest-path computations are accelerated by either variant of Floyd's or Dijkstra's algorithms. In order to improve this algorithm runtime, we have also designed four constraints ordering heuristics that we employed to perform prunings, as soon as possible, in the BAB process. Most of them are based on graph cycle detection.

Scheduling results show that, in effect, the greedy heuristic has a suitable behavior, at least on our benchmarks. On the other hand, the BAB algorithm has an acceptable runtime but can be vulnerable to some rare pathological cases. Concerning the constraints ordering heuristics, the results show that, in most cases, they give better runtime than the original solution. Compared to the ILP technique, the BAB algorithm show better behavior when tasks execution times are large.

Contextually, the BAB method is designed to schedule data-independent tasks. However, it can be easily generalized to support problems of resource-constrained scheduling even when tasks are dependent. Quite paradoxically, we saw that the case of independent tasks is the most difficult one for our branch-and-bound formulation; adding dependences is easy and reduces the size of the solution space.

Furthermore, in the last chapter we have proposed some solutions to perform the last scheduling step –the simultaneous scheduling– defined in the two-step approach. Similar as in the first part, we use "dis-equations", as mathematical way to express uniformly both constraints: resource constraints and data dependences. Consequently, to find an optimal schedule, this formalism led us to properly coloring the obtained conflict graph. In order to optimally color this graph and conversely to classic graph coloring algorithms, we design a new method so that coloring is done by means of a branch-and-bound that is accelerated by clique computation algorithms. The clique computation can be accomplished exactly as well as greedily.

Illustrated by means of some practical benchmarks, the effectiveness and efficiency of the method are good. However, the algorithm version which is coupled with the greedy clique algorithm, is faster than the one using a maximal clique computation. Moreover, the effectiveness of this algorithm is also proven by some comparative experiments. Indeed, we have compared the designed algorithm to a classical ILP solution. In fact, except for few cases, the results show that our BAB algorithm has a better behavior than the ILP technique.

Finally, let us argue why we opt for exact solutions to resolve the defined scheduling problems. Indeed, this choice interest is twofold. Firstly, it is true that embedded systems designers tolerate much longer compilation times than high-performance programmers. A design is the result of many iterations in which different architectural options are evaluated. It is likely that scheduling, even when using complex techniques like BAB or ILP, takes negligible time comparing with extensive simulation or place-and-route synthesis. GS is well suited for the initial exploration. In the final phases, when one must meet strict performance constraints, the use of an optimal method like the BAB or the ILP algorithms may be warranted. Secondly, we believe that integrating exact methods into hierarchical scheduling approaches, that could identify code fragments (so with manageable size) and schedule them optimally couldn't affect a lot the whole scheduling runtime.

9.2 Future Directions

Modeling scheduling problems and designing their correspondent algorithms is known as a laborious task, especially for constrained systems such as the embedded ones. In this thesis, we have mainly tried to get some exact methods but we believe that the performances of these algorithms can be improved. We have thought about to many ideas which will be exploited. Currently, we investigate some ones and left for future work some others.

Concerning the greedy algorithm, it might be interesting to design *a priori* task reordering heuristics, using ideas similar to those we applied to the BAB algorithm.

The situation is similar for both Branch-and-Bound-based algorithms, many improvements can be envisaged as Branch-and-Bound itself is a meta-algorithm, which can be configured in many different directions according to the context.

About the first Branch-and-Bound Algorithm

In fact, the Branch-and-Bound coupled with the Dijkstra's algorithm variant, we have chosen, is the most obvious. One may consider variants, in which the lower bound –computation of the longest paths– is not computed for all the nodes, or in which the order of elaboration of the nodes is best-first instead of depth-first.

In addition, we have used an incremental version of the longest-paths computation, whose core is Dijkstra's static algorithm. This algorithm is faster and less memory consuming comparing with some others. However, as we have already mentioned, that it can be sped up by replacing the second call to Dijkstra's algorithm by one of its dynamic versions recently published.

About the Graph Coloring Algorithm

On the other hand, experimental results about the designed branch-and-bound-based graph coloring algorithm, although they have to be proven by more larger graphs, lead us to take more attention about this graph coloring kernel algorithm which is instrumented to compute a schedule in our context. Indeed, "graph coloring" and its generalizations are useful tools in modeling a wide variety of scheduling and assignment problems and many interesting practical problems. Hence, we have thought about many tricks which are able to improve this method performances –specially its runtime–.

First, let us recall that, for certain graphs, both chosen chromatic number lower bounds, ($\chi(G) \geq n/\alpha(G)$ and $\chi(G) \geq \omega(G)$) may be very weak; it can happens that $\omega(G)$ can be much smaller than $\chi(G)$. In such case, one can search, in the wide graphs' literature, for more better lower bounds by analyzing deeply and extracting the real nature and features of the built graphs in our context. In fact, we currently explore another lower bound which is surely ideal for our coloring strategy. Indeed, Grötschel, Lovász, and Schrijver proved [54], thanks to their "sandwich theorem", that we can compute in polynomial time a real number that is "sandwiched" between these hard-to-compute integers ($\omega(G)$ and $\chi(G)$). They call this lower bound the "theta function":

$$\omega(G) \leq \theta(G) \leq \chi(G).$$

Additional improvements can be envisaged before the branching procedure i.e., when getting the pair of nonadjacent vertices. In the current version, we get greedly, in a polynomial time, the first pair that respects the condition. One can use an algorithm which chooses the most adequate couple of nonadjacent vertices such that it will improve soon the lower bounds along a branch. Indeed, we are currently trying to get an *incremental* version of a greedy clique computation algorithm that will be linked to the algorithm which chooses the pair of nonadjacent vertices. The aim is to reach a complexity of $O(1)$ instead of the currently pseudo-polynomial one. The principle of this algorithm is described as follows:

- At the root of the BAB tree structure, we get a greedy clique of G using the current static greedy clique computation algorithm. Let C be this clique.
- At each node, during the choice of the nonadjacent vertices, first, we try to examine the greedy clique vertices so that the choosen vertex pair contents, if it is possible, one vertex among the clique vertices. Let (c_1, c_2) be the chosen pair of vertices.
- Successively, at each step, C is increased, of course if it is possible, by one vertex. The candidate vertex is chosen among the set c_1, c_2 i.e., among the couple of the two nonadjacent vertices. This choice criterion may at least guarantee that the successive joining operations will early increase the size of C. Indeed, after the nonadjacent vertices choice two situations are possible. 1/ c_1 or c_2 belongs to C, and successive joinings will increase soon the size of C, 2/ neither c_1 nor c_2 belongs to C, and we know that C is an independent component of G, then if its size is greater than $n/2$ thus it is the exact clique number else we can try to construct another greedy clique.

Further future work consists in comparing the designed coloring technique with some other exact techniques [21, 91] especially those designed for resolving VLSI CAD problem [31].

In the interest of fast prototyping, all our algorithms are implemented in the programming language of the MuPAD computer algebra system, which is interpretive. Despite the suitable results, a C or Fortran implementation would certainly improve more the runtime of the designed algorithms.

Glossary

ASAP: –As Soon As Possible– a scheduling technique. It schedules operations taking into account only the data dependences constraints.

ALAP: –As Late As Possible– a scheduling technique.

AFAP: –As Fast As Possible– a scheduling technique.

Architectural synthesis: see Behavioral synthesis.

BAB: Branch-And-Bound technique is a general algorithmic meta-method for finding optimal solutions of various optimization problems, especially in discrete and combinatorial optimization.

Behavioral synthesis: the process which takes a program –behavioral specification– of a system to be performed on a dedicated circuit – and finds a structural description –description of a circuit– that implements this behavior.

Binding: assignment of operations, memory accesses, and interconnections from the behavioral design description to hardware units for optimal area and performance.

CAD: –Computed-Aided Design–.

Chaining: the scheduling of two data-dependent operations in the same control-step.

Clique: For an undirected graph G, a clique represents a set of pairwise adjacent vertices in G.

DFS: –Deepth-First Search–.

Dis-equation: negation of equation.

DSP: –Digital Signal Processing–.

FPGA: –Field Programmable Gates Array– a programmable circuit.

FSM: –Finite-State machine– a design model for representing controllers that assigns boolean constants to output signals in every clock cycle.

FSMD: –Finite-State Machine with Datapath– is a model for representing control-dominated and data-dominated designs that augments FSM model with variables and expressions that specify conditions and actions in each state.

ILP : –Integer Linear Programming (Program)–.

Latency: It represents the needed runtime for a circuit between the the first data reception and the last data emission.

Logic synthesis: synthesis of a structural view, described via memory elements and boolean expressions, into logic gates.

RTL: –Register Transfer Level– a description level where the register assignments are done.

Multi-cycling : a scheduling technique in which a ”complex” operation is scheduled in multiple clock cycles. This echnique allows slower functional units use within faster clocks.

Netlist A list, by name, of each signal and each symbol component and pin logically connected to the signal (or net). A netlist can be generated automatically by a Computer-Aided Design system.

Resources-constrained Scheduling: scheduling operations while respecting a given resource constraints set.

Retiming : is the technique of moving the structural location of registers in a circuit to improve its performance, area, and/or power characteristics in such a way that preserves its functional behavior at its outputs.
Scheduling : partitioning the design behavior into time control steps.
Set Independent: –SI– for a graph $G = (V, E)$, a set independent represents a vertex set from V, so that no two vertices are adjacent.
SoC : –System On a Chip–.
Sub-optimal solution: near optimal solution.
Time-constrained Scheduling : assignment of operations into control steps, given a fixed execution time.
Verilog: a hardware description language (IEEE Standard 1364-2005).
VHDL: a hardware description language (IEE Std 1076-1987) used by designers to describe design behavior and structure at various abstraction levels.

Personal Bibliography

Journal articles

- Hadda Cherroun, Alain Darte and Paul Feautrier. Reservation Table Scheduling: Branch-and-bound based Optimization vs. Integer Linear Programming Techniques. RAIRO-OR, 41(4): 427–454, December 2007.

Conference articles

- Hadda Cherroun and Paul Feautrier. An Exact Resource Constrained-Scheduler using Graph Coloring Technique. In *AICCSA'07: Proceedings of the 5th ACS/IEEE International Conference on Computer Systems and Applications*, pages 554–561. IEEE Computer Society. Amman, Jordan, May 2007. Note: Best paper award.
- Hadda Cherroun, Alain Darte and Paul Feautrier. Reservation Table Scheduling: a Branch-and-bound Based Optimization vs. Integer Linear Programming Techniques. In *Colloque sur l'Optimisation et les Systèmes d'Information (COSI'06)*, Algiers, Algeria, June 2006.
- Hadda Cherroun, Alain Darte and Paul Feautrier. Scheduling under Resource Constraints using Dis-equations. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 1067–1072, Munich, Germany, March 2006.
- H. Cherroun and B. Ziani Retour sur expérience: un premier pas vers l'introduction des TIC dans l'enseignement à l'université de Laghouat. In *CEMAFORAD'O2, Congrès Euro-méditéranéen sur l'approfondissement sur la formation à distance.* Bejaia, Algérie, Novembre 2005.
- H. Cherroun, P. Feautrier and A. Nacer. Conception des paralleliseurs automatiques: Stic, un nouveau test de dépendance de données. In *ISPS'01: Proceedings of the Fifth International Symposium on Programming Systems*, pages 152–162, Algiers, Algeria, May 2001.

Internal Reports

- Hadda Cherroun, Alain Darte and Paul Feautrier. Scheduling with Resource Constraints using Dis-equations. Technical Report 2005-40, LIP, ENS-Lyon, September 2005.

Bibliography

[1] DCDL 2000. Quick reference guide for the Design Constraints Description Language (DCDL). version 0.3.7, `http://www.vhdl.org/dcwg/`.

[2] R. Airiau, J.-M. Bergé, V. Olive, and J. Rouillard. *VHDL: langage, modélisation, synthèse.* Presses polytechnique et universitaire romandes, 1990.

[3] Jonathan Allen. Computer Architecture for Digital Signal Processing. In *Proceedings of the IEEE, 73, 5*, pages 852–873, May 1985.

[4] L. Allison, T. I. Dix, and C. N. Yee. Shortest path and closure algorithms for banded matrices. *Inf. Proc. Lett.*, 40:317–322, Dec. 1991.

[5] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, june 1991.

[6] I. Auge, F. Petrot, F. Donnet, and P. Gomez. Platform-based design from parallel C specifications. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 24(12):1811 – 1826, Dec. 2005.

[7] R. M. Badia, J. Cortadella, and E. Ayguadé. Computer-Aided Synthesis of Data-Path by Using a Simulated Annealing Based Approach. In *Proc. 9th LASTED Int'l Sympo. Applied Informatics*, pages 326–229, 1992.

[8] Brian Bailey and Dan Gajski. RTL Semantic and Methodology. In *Proceedings of the International Symposium on System Synthesis*, pages 69–74. ACM Press, 2001.

[9] Philippe Baptiste. *A Theoretical and Experimental Study of Resource Constraint Propagation.* PhD thesis, University of Compiègne, 1998.

[10] Karen Bartleson. A New Standard for System-Level Design. Synopsis, Inc., 1999.

[11] C. Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC'03 IEEE International Symposium on Parallel and Distributed Computing*, pages 23–30, Ljubjana, October 2003.

[12] C. Bastoul, A. Cohen, A. Girbal, S. Sharma, and O. Temam. Putting Polyhedral Loop Transformations to Work. In *LCPC'16 International Workshop on Languages and Compilers for Parallel Computers, LNCS 2958*, pages 209–225, College Station, october 2003.

[13] Béla Bollobas. *Modern Graph Theory.* Springer, 1996.

[14] F. Benhamou and A. Colmerauer. *Constraint Logic Programming, Selected Research.* MIT Press, 1993.

[15] Reinaldo A. Bergamaschi, R. Camposano, and M. Payer. Scheduling under resource constraints and module assignment. *INTEGRATION, the VLSI Journal*, 12:1–19, Dec 1991.

[16] Reinaldo A. Bergamaschi, Raul Camposano, and Michael Payer. Area and performance optimizations in path-based scheduling. In *EURO-DAC '91: Proceedings of the conference on European design automation*, pages 304–310, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[17] Reinaldo A. Bergamaschi, Salil Raje, and Louise Trevillyan. Control-flow versus data-flow-based scheduling: combining both approaches in an adaptive scheduling system. *IEEE Trans. Very Large Scale Integr. Syst.*, 5(1):82–100, 1997.

[18] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Scheider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The PERFECT Club benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, 1989.

[19] Pangrle B.M and Gajski D.D. Slicer: A state synthesizer for intelligent silicon compilation. In *Proc. IEEE Int. Conf. Computer Design: VLSI un Computers and Processors.*, 1987.

[20] Pierre Boulet and Paul Feautrier. Scanning polyhedra without do-loops. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, page 4. IEEE Computer Society, October 1998.

[21] Daniel Brélaz. New methods to color vertices of a graph. *Commun. ACM*, 22(4):251–256, 1979.

[22] Cadence Design System, Inc. Cadence Kits.
`http://www.cadence.com/`, site visited on 04/08/2006.

[23] Raul Camposano. Path-based scheduling for synthesis. In *IEEE Trans. Computer-Aided Design, vol. 10*, pages 85–93, Jan 1991.

[24] Raul Camposano. Behavioral synthesis. In *33rd Design Automation Conferences*, 1996.

[25] J. Carlier and P. Chrétienne. *Problèmes d'ordonnancement: modélisation, complexité et algorithmes.* Masson, 1988.

[26] En-Shou Chang and Daniel D.Gajski. A connection-oriented binding model for binding algorithms. Technical report, Center for Embedded Computer Systems, University of California,Irvine, 1996.

[27] Samit Chaudhuri and Robert Walker. ILP-Based scheduling with time and resource constraints in high level synthesis. In *Proc. of VLSI Design'94 (India)*, pages 17–25, 1994.

[28] Hadda Cherroun and Paul Feautrier. An exact resource constrained-scheduler using graph coloring technique. In *AICCSA'07: Proceedings of the 5th ACS/IEEE International Conference on Computer Systems and Applications*, pages 554–561. IEEE Computer Society, May 2007. Note: Best paper award.

[29] David R. Coelho. VHDL: a call for standards. In *DAC '88: Proceedings of the 25th ACM/IEEE conference on Design automation*, pages 40–47, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[30] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press and McGraw-Hill Book Company, 1989.

[31] Olivier Coudert. Exact coloring of real-life graphs is easy. In *Design Automation Conference*, pages 121–126, 1997.

[32] Alain Darte, Yves Robert, and Frédéric Vivien. *Scheduling and Automatic Parallelization.* Birkhauser Boston, 2000.

[33] David Landskov and Scott Davidson and Bruce Shriver and Patrick W. Mallett. Local Microcode Compaction Techniques. *ACM Comput. Surv.*, 12(3):261–294, 1980.

[34] Giovani De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, 1994.

[35] M. Dhodhi, F. Hielscher, R. Storer, and J. Bhasker. Datapath synthesis using a problem-space genetic algorithm. In *IEEE Trans. on CAD*, volume 14, pages 934–944, Aug. 1995.

[36] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Monthly*, 91(6):333–352, 1959.

[37] François Donnet. *Synthèse de haut niveau contrôlée par l'utilisateur.* PhD thesis, Université Paris VI, January 2004.

[38] Nils Ellmenreich, Peter Faber, Martin Griebl, Robert Günz, Harald Keimer, Wolfgang Meisl, Sabine Wetzel, Christian Wieninger, and Alexander Wüst. LooPo - Loop Parallelization in the Polytope Model.

[39] Paul Feautrier. Dataflow Analysis of Array and Scalar References. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[40] Paul Feautrier. Some efficient solutions to the affine scheduling problem. part II: Multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[41] Paul Feautrier. Scalable and modular scheduling. In Springer Verlag, editor, *In Andy D. Pimentel and Stamatis Vassiliadis, editors, Computer Systems: Architectures, Modeling and Simulation (SAMOS 2004)*, volume LNCS 3133, pages 433–442, July 2004.

[42] Robert W. Floyd. Algorithm 97 (Shortest Paths). *Communication of the ACM*, 5(6):345, 1962.

[43] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Incremental Algorithms for the Single-Source Shortest Path Problem. In *Proceedings of the 14th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 113–124, London, UK, 1994. Springer-Verlag.

[44] D. D. Gajski and L. Ramachandran. Introduction to High-Level Synthesis. *IEEE Design and Test of Computers*, 11(4):44–54, 1994.

[45] Daniel D. Gajski. *Principle of Digital Design.* Prentice Hall International Edition, 1997.

[46] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., 1979.

[47] Catherine H. Gebotys and Mohamed Elmasry. Simultaneous Scheduling and Allocation for Cost Constrained Optimal Architectural Synthesis. In *28th Annual ACM/IEEE Design Automation Conference (DAC'91)*, pages 2–7, San Francisco, CA, USA, 1991.

[48] E.F. Girczyc. Applicability of a Subset of ADA as an Algorithmic Hardware Description Language for Graph-Based Hardware Compilation.

[49] E.F. Girczyc and d.P. Knight. An ADA to Standard Cell Hardware Compiler Based on Graph Grammars and Scheduling. In *Proc. of the IEEE International Conference on Computer Design (ICCD)*, pages 726–731, October 1984.

[50] Milind Girkar and Constantine D. Polychronopoulos. The hierarchical task graph as a universal intermediate representation. *Int. J. Parallel Program.*, 22(5):519–551, 1994.

[51] D.E. Goldberg. *Genetic Algorithm in Search, Optimization and Machine Learning.* Addison-Wesley, Reading, MA., 1989.

[52] Sriram Gorindarajan. Scheduling algorithms for high-level Synthesis, March 1995. "Methods and algorithms for system design" notes of course, Faculty of Electrical Engineering Mekelweg Delft, The Netherlands.

[53] Martin Grötschel. Discrete Mathematics in Manufacturing. In Robert E. O'Malley, editor, *ICIAM 1991: Proceedings of the Second International Conference on Industrial and Applied Mathematics*, pages 119–145. SIAM, 1991.

[54] Martin Grötschel, László Lovász, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization.* Springer-Verlag, 1988.

[55] Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. SPARK: A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations. In *VLSID'03: Proceedings of the 16th International Conference on VLSI Design (VLSI'03)*, page 461. IEEE Computer Society, 2003.

[56] P. Gutberlet, J. Müller, H. Krämer, and W. Rosenstiel. Automatic module allocation in high level synthesis. In *European Design Automation Conference EURODAC'92, Hamburg, 1992*, pages 328–333, 1992.

[57] M. J. M. Heijligers, L. J. M. Cluitmans, and J. A. G. Jess. High-level synthesis scheduling and allocation using genetic algorithms. In *ASP-DAC '95: Proceedings of the 1995 conference on Asia Pacific design automation (CD-ROM)*, page 11, New York, NY, USA, 1995. ACM Press.

[58] Dorit S. Hochbaum, editor. *Approximation algorithms for NP-hard problems.* PWS Publishing Co., Boston, MA, USA, 1997.

[59] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, 1978.

[60] S. H. Huang, Y. L. Jeang, C. T. Hwang, Y. C. Hsu, and J. F. Wang. A tree-based scheduling algorithm for control-dominated circuits. In *DAC '93: Proceedings of the 30th international conference on Design automation*, pages 578–582, New York, NY, USA, 1993. ACM Press.

[61] C.T. Hwang, T.H. Lee, and Y. C. Hsu. A formal approach to the scheduling problem in high level synthesis. *IEEE Transactions on CAD*, April 1991.

[62] Benzakki J. and M. Israel. OSYS : A High-Level Synthesis Tool of VLSI and HW/SW CoDesign. In *Second IFIP International Workshop on Hardware/Software Codesign (Codes/CASHE'93), Innsbruck-Igls, Austria* , pages 24–27, May 1993.

[63] Rajiv Jain, Ashutosh Mujumdar, Alok Sharma, and Hueymin Wang. Empirical evaluation of some high-level synthesis scheduling heuristics. In *DAC '91: Proceedings of the 28th conference on ACM/IEEE design automation*, pages 686–689, New York, NY, USA, 1991. ACM Press.

[64] Gilles Kahn. The Semantics of a Simple Language for Parallel Programming. In *IFIP Congress*, pages 471–475, 1974.

[65] Daniel Kästner and Marc Langenbach. Integer Linear Programming vs. Graph-Based Methods in Code Generation. Technical Report A/01/98, Universität des Saarlandes, February 1998.

[66] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983.

[67] P. Kission, H. Ding, and A. A. Jerraya. Structured Design Methodology For High-Level Design. In *In Proc. of 31st Design Automation Conference* , June 1994.

[68] Krzysztof Kuchcinski. Constraints-Driven Scheduling and Resource Assignment. *ACM Trans. Des. Autom. Electron. Syst.*, 8(3):355–383, 2003.

[69] Monica Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proc. SIGPLAN Conf. Programming Language Design and Implementation*, pages 318–328,, June 1988.

[70] A. H. Land and A. G. Doig. An Automatic Method of Solving Discrete Programming Problems. *Econometrica*, 28(3):497–520, Jul 1960.

[71] B. Landwehr, P. Marwedel, and R. Doemer. OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming. In *European Design Automation Conference*, 1994.

[72] J. Lee, Y. Hsu, and Y. Lin. A new Integer Linear Programming Formulation for the Scheduling Problem in Data-Path Synthesis. In *Proc. of Int. Conf. on Computer-Aided Design*, pages 20–23, 1989.

[73] Tai Ly, David Knapp, Ron Miller, and Don MacMillen. Scheduling using behavioral templates. In *DAC'95: Proceedings of the 32nd ACM/IEEE Conference on Design Automation*, pages 101–106, New York, NY, USA, 1995. ACM Press.

[74] E. Martin, O. Stentieys, H. Dubois, and J.L. Philippe. GAUT : An Architectural Synthesis Tool for Dedicated Signal Processors. In *EURO-DAC'93, Hambourg, Germany*, pages 20–24, Sep. 1993.

[75] P. Marwedel. Matching System and Component Behaviour in the MIMOLA Synthesis Tools, 1990.

[76] Dániel Marx. Graph Coloring Problems and their Applications in Scheduling. *Periodica Polytechnica Ser. El. Eng.*, 48(1-2):5–10, 2004.

[77] M. McFarland. Using bottom-up design techniques in the synthesis of hardware from abstract behavioral descriptions. In *Proceedings of the ACM/IEEE 23rd Design Automation Conference*, pages 474–480, 1986.

[78] Mentor. Graphics Catapult.
`http://www.mentor.com/products/c-based_design/catapult_c_synthesis`.

[79] M. Minoux. *Programmation mathématique: théorie et algorithmes.* Dunod, Paris, 1983.

[80] Steven S. Muchnick. *Advanced Compiler Design Implementation.* Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[81] K.G. Murty. *Operations Research: Deterministic Optimization Models.* Prentice-Hall, 1995.

[82] M. Held N. Wehn and M. Glesner. A novel scheduling/allocation approach for datapath synthesis based on genetic paradigms. In *Proc. TC10/W10.5 Workshop Logic and Architectural Synthesis*, pages 47–56, 1990.

[83] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization.* J. Wiley, New York, 1988.

[84] CPLEX Optimization . Using the CPLEX Callable Library, 1995.

[85] Preeti R. Panda and Nikil D. Dutt. 1995 High-Level Synthesis Design Repository. In *ISSS '95: Proceedings of the 8th International Symposium on System Synthesis*, pages 170–174, New York, NY, USA, 1995. ACM Press.

[86] B. M. Pangrle and D. D. Gajski. Design Tools for Intelligent Silicon Compilation. *IEEE Transactions on Computer-Aided Design*, 6(6):1098–1112, November 1987.

[87] In-Cheol Park and Chong-Min Kyung. Fast and near optimal scheduling in automatic data path synthesis. In *DAC '91: Proceedings of the 28th conference on ACM/IEEE design automation*, pages 680–685, New York, NY, USA, 1991. ACM Press.

[88] Alice C. Parker, Jorge T. Pizarro, and Mitch Mlinar. MAHA: a program for datapath synthesis. In *DAC '86: Proceedings of the 23rd ACM/IEEE conference on Design automation*, pages 461–466, Piscataway, NJ, USA, 1986. IEEE Press.

[89] Pierre G. Paulin and John P. Knight. Force-directed scheduling in automatic data path synthesis. In *24th ACM/IEEE DAC*, 1987.

[90] Pierre G. Paulin and John P. Knight. Algorithms for High-Level Synthesis. *IEEE Des. Test*, 6(6):18–31, 1989.

[91] Jürgen Peemöller. A Correction to Brelaz's Modification of Brown's Coloring Algorithm. *Commun. ACM*, 26(8):595–597, 1983.

[92] Van Laarhoven P.M.J and Aart E.H.L. *Simulated Annealing: Theory and Application.* Group, Dordrecht, Kluwer Academic, 1987.

[93] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, October 2000.

[94] Ivan Radivojevic and Forrest Brewer. A New Symbolic Technique for Control-Dependent Scheduling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(1):45–57, January 1996.

[95] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 1992.

[96] D. S. Rao and F. J. Kurdahi. Partitioning by regularity extraction. In *DAC '92: Proceedings of the 29th ACM/IEEE conference on Design automation*, pages 235–238, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[97] Tangy Risset. Contribution à la compilation de nids de boucles sur silicium. Habilitation à diriger des recherches, Université de Rennes 1, Octobre 2000.

[98] Samir Palnitkar. *Verilog HDL : A Guide to Digital Design and Synthesis.* Prentice Hall, 1998.

[99] Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, Inc., 1986.

[100] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An Empirical Study of Fortran Programs for Parallelizing Compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, 1990.

[101] L. Stok and R. Van Den Born. EASY : Multiprocessor Architecture Optimization, 1988.

[102] Synopsys. Inc. Behavioral SystemC Compiler. `http://www.synopsys.com/products/bc`.

[103] Synopsys. Inc. CoCentric SystemC Compiler. `http://www.synopsys.com/products/cocentric`.

[104] I.D. Ullman. NP-complete Scheduling Problems. *Journal Comput. System Sci.*, 10(10):384–393, 1975.

[105] W. F. J. Verhaegh, P. E. R. Lippens, E. H. L. Aarts, J. H. M. Korst, J. L. van Meerbergen, and A. van der Werf. Improved Force-directed Scheduling in High-Throughput Digital Signal Processing. In *IEEE Trans. Computer-Aided Design*, volume 14, pages 945–960, Aug. 1995.

[106] W. G. J. Verhaegh, E. H. L. Aarts, P. C. N. Van Gorp, and P. E. R. Lippens. A Two-Stage Solution Approach to Multidimensional Periodic Scheduling. *IEEE Transactions on Computer-Aided Design*, 20(10):1185–1199, October 2001.

[107] Jurij Šilc. Scheduling strategies in high-level synthesis. *Informatica (Slovenia)*, 18(1), 1994.

[108] S. Warshall. A theorem on boolean matrices. *JACM*, 9(1):11–21, 1962.

[109] D. Welsh and M . Powell. An upper bound for the chromatic number of a graph and its applications to timetabling problems. *Comput. J.*, 10:85–86, 1967.

[110] Douglas B. West. *Introduction to Graph Theory.* Prentice Hall, 1996.

[111] Douglas B. West. *Introduction to Graph Theory, Second Edition.* Prentice Hall, 1996.

[112] T.C. Wilson, N. Mukherjee, M.K. Garg, and D. K. Banerji. An ILP Solution for Optimum Scheduling, Module and Register Allocation, and Operation Binding in Datapath Synthesis. *VLSI Design*, 1995.

[113] Xilinx Inc. `http://www.xilinx.com`, visited on 26/03/2006.

[114] Jerry Chih-Yuan Yang, Giovanni De Micheli, and Maurizio Damiani. Scheduling and Control Generation with Environmental Constraints Based on Automata Representations. In *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, pages 166–183, Feb. 1996.

[115] Peng Yang and Francky Catthoor. Pareto-optimization-based run-time task scheduling for embedded systems. In *CODES+ISSS'03: Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (ISSS'03)*, pages 120–125. ACM Press, 2003.

[116] Peng Yang, Chun Wong, Paul Marchal, Francky Catthoor, Dirk Desmet, Diederik Verkest, and Rudy Lauwereins. Energy-Aware Runtime Scheduling for Embedded-Multiprocessor SoCs. *IEEE Des. Test*, 18(5):46–58, 2001.

[117] L. Zhang. *SILP: Scheduling and Allocating with Integer Linear Programming* . PhD thesis, Technische Fakultät der Universität des Saarlandes, 1996.

Appendix A

Graph Algorithms

The most algorithms designed in this thesis are based on graph elementary algorithms. For sake of clarifying their principle and their complexity, we have dressed this appendix. Indeed, in this appendix we recall some fundamental graph algorithms and theirs complexities are detailed if necessary.

Given a graph $G = (V, E)$, where V is a set of n vertices and E the set of m edges. In what follow, as we will talk about both undirected and directed graphs, we use the following terminology: we use edge and cycle for undirected graphs, and arc and circuit for directed graph.

A.1 Depth-First Search Algorithm and its Features

Depth-First Search (DFS) algorithm is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms.

The strategy followed by depth-first search is, as its name implies, to search "deeper" in the graph whenever possible. In depth-first search, edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it. When all edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then one of them is selected as a new source and the search is repeated until all vertices are discovered.

Depth-first search provides many information about the structure of a graph and has many interesting features, we can quote:

1. Depth-first search builds a sub graph of predecessors $G_\pi = (V, E_\pi)$, where : $E_\pi = \{(\pi(v), v) : \quad v \in V \text{ and } \pi(v) \neq NIL\}$. G_π forms a **depth-first forest**. In the case of an undirected (resp. directed) graph, the edges (resp. arcs) of E_π are called **tree edges** (tree arcs).

2. Besides creating a depth-first forest, depth-first search also "time-stamps" each vertex. Each vertex v has two time-stamps: the first time-stamp $d[v]$ records when v is first discovered (and grayed), and the second time-stamp $f[v]$ records when the search finished examining all the adjacency list and blackens v. Thus, a vertex v is a proper descendent of a vertex u in the depth-first forest of a given directed/undirected graph iff $d[u] < d[v] < f[v] < f[u]$.

3. Another interesting property of depth-first search is that the search can be used to classify the edges of the input graph G. We can define four edge types in terms of the depth-first forest G_π produced by a depth-first search on G.

Algorithm 5: Depth-First Search Algorithm

```
begin
    foreach vertex $u \in V[G]$ do
        $color[u] \leftarrow$ White;
        $\pi[u] \leftarrow$ NULL;
    end
    $time \leftarrow 0$;
    foreach vertex $u \in V[G]$ do
        if $color[u] = White$ then
            PP-Visit (u);
        end
    end
end
PP-Visit($u$)
begin
    $color[u] \leftarrow$ Gray;
    $d[u] \leftarrow time + 1$ ;
    foreach vertex $v \in Adj[v]$ do
        if $color[u] = White$ then
            $\pi[u] \leftarrow u$;
        end
    end
    $color[u] =$ Black;
    $f[u] \leftarrow time + 1$ ;
end
```

- **Tree edges** : are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).
- **Back edges (R)** are those edges (u, v) connecting a vertex u to an ancestor v in the depth-first tree G_{pi}. Self-loops, which may occur in directed graphs, are considered to be back edges.
- **Forward edges (T)** are those non-tree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
- **Cross edges (A)** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

It should be mentioned that in the case of undirected graphs, we can distinguished only two kinds of edges: edges belonging to the depth-first forest and those which don't belong to it. Let us see Figure. A.1: (a) represents the steps of the DFS, where each vertex u is labeled by its $d[u]/d[f]$, and (b) represents the depth-first forest (bold arrows) and the classification of arcs.

The complexity of Algorithm 5 is in $O(n + m)$. This algorithm can serve to enumerate a basis of cycles and paths of G. In addition, in the case of graph without cycles, one can establish the topological sort by exploiting the values of f. Furthermore, the DFS allows a cycle detection and it can be used to enumerate the strongly connected components of a graph ....

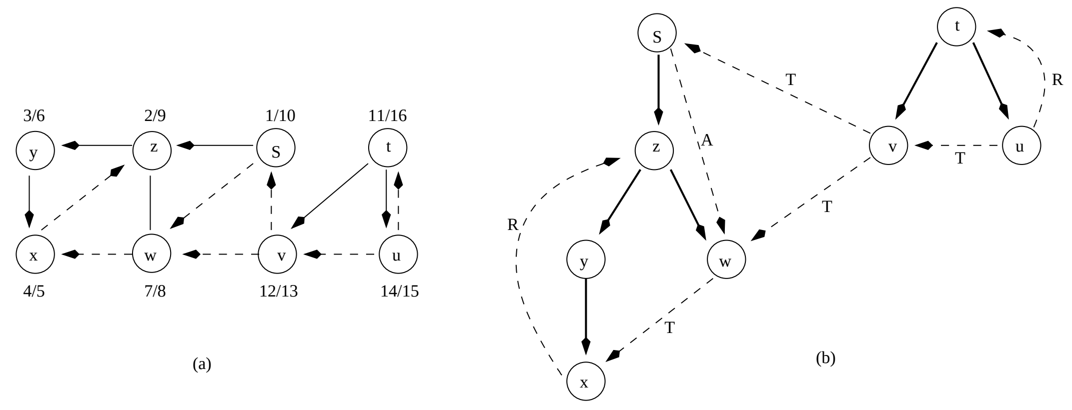


Figure A.1: DFS Properties

A.2 Maximal-weight Paths

In the problem of finding paths of maximal weights [32] (for brevity we use longest-paths) in a graph [30] we are given a directed graph $G = (V, E, w)$, where V is a set of n vertices and E a set of arcs with a weight function $w : E \to \mathcal{Z}$ which affects each arc (i, j) by a real value –weight– $w(i, j)$. The weight of path $P = [v_0, v_1, \ldots, v_k)$ is the sum of the weights of its constituent edges: $w(P) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$. We define the path of maximal weight from u to v by:

$$\Delta(u,v) = \begin{cases} \max_p w(p) : u \overset{p}{\rightsquigarrow} v & \text{if there are paths from } u \text{ to } v \\ +\infty & \textit{else} \end{cases}$$

In the literature, we can distinguished two kinds of algorithms resolving this problem according to what we consider: 1/ searching paths of maximal weight from a given vertex s –called source– to all the others, the problem is called “Single-source longest-paths problem” or 2/ searching paths of maximal weights from u to v for every pair of vertices u and v, the problem is called “All-pairs longest-paths problem”.

A.2.1 Relaxation Technique

First, let us explain the relaxation technique that is used in both algorithms, described in what follow, which finds paths of maximal weight from a given source s to all the others.

In this technique, for each vertex $v \in V$, we maintain an attribute $d[v]$, which is an upper bound on the weight of a longest-path from source s to v. We call $d[v]$ the “estimate of the path of maximal weight ”. We initialize the estimate paths of maximal weight by $d[v] = -\infty$ for each v. We define $\pi(v)$ the predecessor of vertex v for which $d[v]$ has been updated for the last time.

```
RELAX $(u, v, w)$
if $d[v] < d[u]$ then
    $d[v] \leftarrow d[u] + w(u, v)$;
    $\pi(v) \leftarrow u$;
end
```

[32]In the literature, these algorithms are often presented as finding paths of minimal weight. This is the same, one just have to change the weight signs. Our explanations are based on maximal-weight paths.

Algorithm 7: Dijkstra's Algorithm

```
début
    Initialize-source($G, s$);
    $S \leftarrow \emptyset$;
    $Q \leftarrow V$;
    while $Q \neq \emptyset$ do
        $u \leftarrow$ Extract-Max($Q$);
        $S \leftarrow \cup \{u\}$;
        foreach $v \in Adj[u]$ do
            RELAX($u, v, w$)
        end
    end
fin
```

The process of relaxing an arc (u, v, w) consists of testing whether we can improve the "path of maximal weight estimate" of v, going through u and, if so, updating $d[u]$ and $\pi(v)$.

A.2.2 Dijkstra's Algorithm

Dijkstra's algorithm resolves the "Single-source longest-paths problem" on a weighted graph whose all edge weights are nonpositive ($w(u, v) \leq 0$). Dijkstra's algorithm maintains a set S of vertices whose longest-paths from the source s have already been determined such that for each vertex $v \in S$ we have $d[u] = \Delta(u, v)$. At each step, the algorithm selects a vertex $u \in V - S$ whose $d[u]$ is maximal –which is done by `Extract-Max` primitive–, adds u to S, and relaxes all edges leaving u.

In Algorithm 7, the data-structure of Q represents a max-priority queue, which contains vertices in $V - S$, keyed by their d values. If the priority queue is implemented with a simple array then the `Extract-Max` procedure will be executed in $O(n)$. In Dijkstra's algorithm the **While** loop is repeated n times. In the whole execution, the **For** loop is repeated $|E|$ times i.e. m times, the number of arcs, thus for each iteration, it takes $O(1)$. Consequently, the total runtime of the algorithm is $O(n^2 + m) \equiv O(n^2)$[33].

However, when the graph is dense –$|E|$ is close to $|V|^2$–, it is more advised to implement the max-priority queue by a *binary heap*[34]. With such implementation, the procedure `Extract-Max` will be in $O(\lg n)$ and eventual assignation $d[v] \leftarrow d[u] + w(u, v)$ will be realized by the primitive `Increase-key` $(Q, v, d[u] + w(u, v))$ in $O(lgn)$, in which case the complexity of the algorithm 7 becames $O((n + m). \lg n)$.

However, the most suitable data structure for speeding up this algorithm is the **Fibonacci-heap** [30]. With this data structure, the amortized cost of each call to `Extract-Max` primitive [35] in $O(\lg n)$, and each of the $|E|$ calls to the `Increase-key` takes only an amortized time in $O(1)$. Consequently, Dijkstra's algorithm can be executed in $O(n. \lg n + m)$.

[33]In what follow, often the $O(|E|)$ is majored by $O(n^2)$

[34]A (binary) heap data structure is an array object that can be viewed as a nearly complete binary search tree which has special basic primitives such : Increase-key (Q, v, k) that increase the key of a node v in $O(lgn)$ while preserving the queue order.

[35]The amortized cost is the required time to perform a sequence of data-structure operations is averaged over all the operations performed. Amortized analysis differs from average-case analysis in that probability is not involved; an amortized analysis guarantees the average performance is an of each operation in the worst case.

Algorithm 8: Bellman-Ford's Algorithm.

```
begin
    Initialize-source($G, s$);
    for $i \leftarrow 1$ to $|V| - 1$ do
        foreach arc $(u, v) \in E$ do
            RELAX($u, v, w$);
        end
    end
    foreach arc $(u, v) \in E$ do
        if $d[v] < d[u] + w(u, v)$ then
            return FALSE ;
        end
    end
    return TRUE ;
end
```

A.2.3 Bellman-Ford's Algorithm

The Bellman-Ford's algorithm solves the single-source longest-paths in the general; i.e. the case where edge weights are arbitrary. Given a weighted, directed $G = (V, E, w)$ with a source s, the Bellman-Ford's algorithm returns a boolean value indicating whether or not there is a positive-weight cycle, that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the longest-paths from the source s to each vertex $v \in V$ and their weights. Bellman-Ford's algorithm –Algorithm 8– runs in $O(n^3)$ time.

In the longest-paths problem, one wishes to find for each pair of vertices u, $v \in V$, the longest-path from u to v, where the weight of a path is the sum of the weights of its constituent edges. We typically want the output in tabular form: the entry in u's row and v's column should be the weight of a longest path from u to v.

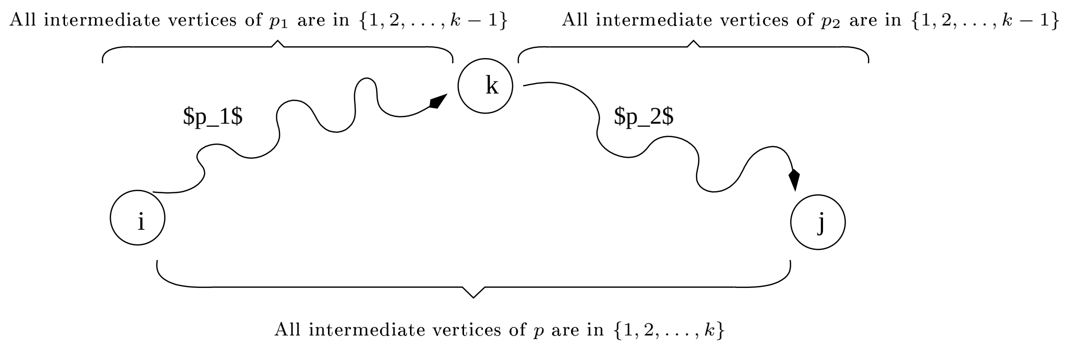
One can solve the all-pairs longest-paths problem by running a single-source longest-paths algorithm $|V|$ times, once for each vertex as the source. If all edge-weights are nonpositive the n calls to Dijkstra's algorithm, according to its implementation needs $O(n^3)$ with an array, $(n^2 + nm) \lg n)$ with a binary heap, or $O(n^2 \lg n + nm))$ with a Fibonacci heap. But in general case, the n calls to the Bellman-Ford's algorithm gives results in $O(n^4)$.

In the literature, there are some solutions which do better. In this short appendix, we report the two well known algorithms to find all-pairs longest-paths: Floyd-Warshall's algorithm and Johnson's algorithm.

A.2.4 Floyd-Warshall's Algorithm

Floyd-Warshall's algorithm, assume that there are no positive-weight cycles. This algorithm uses a dynamic-programming formulation to resolve the all-pairs longest-paths problem. Before proceeding, let us briefly recap the steps for developing any dynamic-programming algorithm, indeed one has to : 1/ characterize the structure of an optimal solution, 2/ recursively define the value of an optimal solution, 3/ compute the value of an optimal solution in a bottom-up fashion.

Indeed, Floyd and Warshall start by characterizing the structure of an optimal solution. Assume that the graph is represented by an adjacency matrix $W = (w_{ij})$.


All intermediate vertices of p_1 are in $\{1, 2, \dots, k-1\}$
All intermediate vertices of p_2 are in $\{1, 2, \dots, k-1\}$
k
p_1
p_2
i
j
All intermediate vertices of p are in $\{1, 2, \dots, k\}$

The algorithm is based on the following observation. Let $V = \{1, 2, \dots, n\}$ be the set of all vertices of G, for each $k \in V$, consider the subset $\{1, 2, \dots, k\}$. Let p be the longest-path from vertices i to j. p is an elementary path as there are nonpositive cycles in G.

The algorithm uses the relationship between the path p and all longest paths from i to j whose components vertices are in the set $\{1, 2, \dots, k-1\}$: (1) If k is not an intermediate vertex in the path p, so all intermediate vertices of p are in $\{1, 2, \dots, k-1\}$, thus the longest-paths from i to j whose intermediates vertices are in $\{1, 2, \dots, k-1\}$ is also a longest path from i to j whose intermediates vertices are in $\{1, 2, \dots, k\}$. (2) If k is an intermediate vertex in p then we cut the path into two sub-paths: $p : i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$, as shown in Figure. A.2.4. The path p_1 (resp. p_2) is a longest-path from i to k (resp. from k to j whose intermediate vertices are in $\{1, 2, \dots, k-1\}$. Indeed, k isn't an intermediate vertex in both paths p_1 and p_2. This is induces directly from the fact that the sub-paths of a longest-path are them self longest-path.

Given a graph where all cycles are negative or null, Floyd-Warshall's algorithm, based on the dynamic programming principle described above, defines $a_{ij}^{(k)}$ as the longest-path from i to j whose intermediates vertices are belonged to $\{1, 2, \dots, k\}$. When $k = 0$, $a_{ij}^{(k)} = w_{ij}$, thus the recursive definition of the longest path is:

$$a_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ max\{a_{ij}^{(k-1)}, a_{ik}^{(k-1)} + a_{kj}^{(k-1)}\} & \text{if } k > 0. \end{cases}$$

Floyd-Warshall's algorithm –Algorithm 9– runs in (n^3) time and its space requirement is about n^2. In this algorithm, the superscript k in $a^{(k)}$ is added only to indicate the number of the iteration. When implementing this algorithm, it does not appear.

A.2.5 Johnson's Algorithm

Johnson's algorithm –Algorithm 10– resolves the all-pairs longest-paths problem in $O(n^2 lgn + m.n)$. For sparse graphs, it is asymptotically better than the Floyd-Warshall's algorithm which is in $O(n^3)$. The algorithm either returns a matrix of longest-path weights for all pairs of vertices or reports that the input graph contains at least one positive-weight cycle. Johnson's algorithm uses, as subroutines, both Dijkstra's algorithm and the Bellman-Ford's algorithm, which are described previously.

Johnson's algorithm uses the technique of **reweighting**, which works as follows. If all edge-weights w in G are non positives, we can find longest paths between all pairs of vertices by running Dijkstra's algorithm one from each vertex. If G has positive-weight edges but no positive-weight cycles

Algorithm 9: Floyd-Warshall's Algorithm

```
begin
    for i from 1 to n do
        for j from 1 to n do
            $a_{ij} \leftarrow w_{ij}$;
        end
        $a_{ii} \leftarrow 0$ ;
    end
    for k from 1 to n do
        for i from 1 to n do
            for j from 1 to n do
                $a_{ij}^{(k)} \leftarrow max\{a_{ij}^{(k-1)}, a_{ik}^{(k-1)} + a_{kj}^{(k-1)}\}$ ;
            end
        end
    end
end
```

This technique defines a new reweighting function w^r which satisfies two proprieties: (1) For each couple of vertices $i, j \in V$, if we knew the longest-paths for $G = (V, E, w^r)$ –using the reweighting function w^r–, it will be easy to find the longest-paths from i to j by using the function w, we talk about "longest-path conservation", (2) for each pair (i, j), the new weight $w^r(i, j)$ is positive or null.

To get the new reweighting function w^r, we define the function $h : V \to \mathbb{R}$ which associate a real value to each vertex. Indeed, for each edge $(u, v) \in E$, we define a new weight $w^r(i, j) = w(u, v) + h(v) - h(u)$.

In Algorithm 10, Bellman-Ford's algorithm is used to check whether a positive-weight cycle, in which case no solution exists. In the opposite case, the longest-paths from the fictitious source s to each vertex $v \in V$, $\Delta(s, v)$ will serve to define the reweighting function h.

Table A.1 summarizes, for each algorithm, described above, their respective complexity and the features of the graph for which they are most suitable.

A.3 Roy-Warshall's Algorithm

Roy-Warshall's algorithm computes, for a directed graph $G = (V, E)$, the reachability relationship between all-pairs of vertices; it returns, for each pair of vertices i, j, a boolean value indicating whether or not there is a path from i to j. Roy-Warshall's algorithm follows the dynamic programming principles same as in Floyd-Warchall's algorithm[36]. Indeed, we define $Acc_{ij}^{(k)}$ the boolean value which attests that j is reachable from i by a path whose intermediates vertices are in the set $\{1, 2, \ldots, k\}$. Thus, we define recursively Acc_{ij} as:

$$Acc_{ij}^{(k)} = \begin{cases} TRUE & \text{If an edge from } i \text{ to } j \text{ exists and } k = 0 \\ FALSE & \text{If no edge from } i \text{ to } j \text{ and } k = 0 \\ Acc_{ij}^{(k-1)} \text{ or } (Acc_{ik}^{(k-1)} \text{ and } Acc_{kj}^{(k-1)}) & \text{if } k > 0. \end{cases}$$

[36]Historically, the Floyd-Warchall's algorithm is inspired from the principle of the Roy-Warshall's algorithm by performing the following substitutions : "or = max" and "and = +"

Algorithm 10: Johnson's algorithm

```
begin
    Build $G'$, Where $V[G'] = V[G] \cup \{s\}$ and
    $E[G'] = E[G] \cup \{(s,v); v \in V[G]\}$;
    if Bellman-Ford $(G', w, s)$ = False then
        " $G$ contains positive-weights cycles"
    end
    foreach vertex $v \in V[G']$ do
        $h(v) \leftarrow -\Delta(s,v)$
        /* Where $\Delta(s,v)$ is computed by Bellman-Ford's Algorithm */
    end
    foreach edge $(u,v) \in E[G']$ do
        $w^r(u,v) \leftarrow w(u,v) + h(v) - h(u)$
    end
    foreach vertex $u \in V[G]$ do
        $\Delta^r(u,v) \leftarrow$ Dijkstra$(G, w^r, u)$;
        foreach vertex $v \in V[G]$ do
            $a_{u,v} \leftarrow -\Delta^r(u,v) + h(u) - h(v)$
        end
    end
    Return $D$ ;
end
```

Algorithm	Problem	Graph properties	Time Complexity			Space Complexity
Dijkstra	Single-source longest-paths	Edge weights must be positive	Using an array	Binary heap	Fibonacci heap (Sparse graph)	$O(n)$
			$O(n^2)$	$O((n+m)\lg n)$	$O(n \lg n + m)$	
Bellman-Ford	//	any graph	$O(n^3$			$O(n)$
Floyd-Warshall	All-pairs longest-paths	no positive-weight cycles	$O(n^3)$			$O(n^2)$
Johnson	//	Sparse graph	$O(n^2 \lg n + n.m)$			$O(n)$

Table A.1: Summary of Longest-Paths Algorithms.

Same as Floyd-Warchall's algorithm, Roy-Warshall's algorithm –Algorithm 11– runs in n^3 time and requires n^2 memory space.

Algorithm 11: Roy-Warshall's Algorithm

```
begin
    for each pair $(i, j) \in (VXV)E$ do
        if edge $(i, j)$ then
            $Acc_{ij} \leftarrow TRUE$;
        else
            $Acc_{ij} \leftarrow FALSE$;
        end
    end
    for $k$ from 1 to $n$ do
        for $i$ from 1 to $n$ do
            for $j$ from 1 to $n$ do
                $Acc_{ij} \leftarrow Acc_{ij}$ or $(Acc_{ik}$ and $Acc_{kj})$ ;
            end
        end
    end
end
```

Appendix B

Benchmark Description

In order to measure the real performances of our algorithms, we have tested them on some test programs get from real-life applications. These consist of code fragments from the *PerfectClub* [18] and *HLSynth95* [85] benchmarks. The *PerfectClub* benchmarks represent applications in a number of areas of engineering and scientific computing and the *HLSynth95* benchmarks, more specifically, represent a repository of applications in embedded systems.

B.1 1995' High-Level Synthesis Design Repository

HLSynth95 represents a set of designs that can serve as examples for High-Level Synthesis systems. Indeed, it contains many applications and modules widely used in embedded systems. The designs vary in complexity from simple behavioral finite state machines to more complex designs such as microprocessors, floating point units, image processing applications, ...

The designs are in C or VHDL languages. All of these designs are available from the design repository at U.C Irvine university (anonymous ftp://ics.uci.edu/pub/hlsynth). Mainly, we have selected the designs described at the behavioral level. Table B.1 summarizes some of the important aspects related to the functionality of these designs, such as typical control features present, style of description and major data types used. The number of lines of code is mentioned to give a rough idea of the design's size.

B.2 PerfectClub Benchmarks

The PerfectClub –For "**PERF**ormance **E**valuation for **C**ost-effective **T**ransformations"– suite is a set of thirteen programs that total well over $50,000$ lines of source code [18]. They represent application in a number of area of engineering and scientific computing, and in many cases they represent codes that are currently used by a number of computational research and development groups. This repository has been originally developed in order to measure performances of parallel architectures and supercomputers.

Our benchmark are simple kernels rather than full-fledged applications, which are much more complex. However, the added complexity is a problem for symbolic scheduling. Indeed, the PerfectClub benchmarks are chosen as they contain scientific applications which contain more parallelism through nested loops. These applications are the most aimed by our general scheduling strategies. Table B.2 reports some features of a part of these designs.

Design Name	Design Description	Design Level	Control features	Data types	Lines of Code
FP_Adder	Floating Point Adder	Algorithmic Behavior	Nested Ifs For loops Proc/Func	Bit Vector Integer Enum	640
FP_Mult	Floating Point Multiplier	Algorithmic Behavior	Nested Ifs For loops Proc/Func	Bit Vector Integer Enum	425
FP_Divider	Floating Point Divider	Algorithmic Behavior	For loops Case Stmt	Bit Vector Integer Enum	410
Barcode	Barcode Reader	Algorithmic Behavior	Nested Loops	Bit Vector Integer	110
Adaptive	Chip Adaptive Interpolation	Algorithmic Behavior	Func Calls Nested Loops	Multi Dimensional Integer arrays	810
Memory (7 models)	Image Processing Applications	Algorithmic Behavior	Func Nested Loops	2-Dimensional Float arrays	140
Beamformer	Filter	Vector Product/Summation	Nested For Loops (4 levels)	3-Dimensional Integer arrays	100
Jacobian	Robot Motion Computation	Algorithmic Behavior	For Loops	2-Dimensional Double Integer arrays	450
FFT	Fast Fourier Transform	Algorithmic Behavior	Nested While Loops	Array of Bit Vector	145

Table B.1: Features of Designs.

Program	Application	Control Features	Lines of Code
ADM	Air Pollution	Nested Ifs, For loops	6 142
SPICE	Circuit Simulation	For loops	18 304
OCEAN	Computational Fluid Dynamics	Nested Ifs, For loops	4215
SPEC77	Weather Simulation	Nested Ifs, For loops	3880
MDG	Liquid Water Simulation	For loops	1231

Table B.2: Part of the PerfectClub Benchmarks Suite.