EXACT SCHEDULING TECHNIQUES FOR HIGH LEVEL SYNTHESIS

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

in

The Department of Electrical and Computer Engineering

by
Mukund Narasimhan
B.Tech., Mangalore University, Manipal, India, 1995
May 1998

Acknowledgments

I would like to express my gratitude to Dr. Ramanujam for his guidance throughout this work. I would also like to thank Dr. R. Vaidyanathan and Dr. D. Koppelman for serving on my committee. I would also like to thank my family - my parents, my sister, and my wife for putting up with me. Last, but not the least, I would like to express my gratitude to all my friends who made my stay at LSU a pleasant one.

Contents

List of Tables

List of Figures

Abstract

High-level synthesis is the automatic synthesis of VLSI designs from behavioral descriptions. Behavioral synthesis offers several advantages over designs at other levels of abstraction. Since the level of abstraction is much higher, the designer does not have to worry about low level considerations, allowing him/her to concentrate on architectural and algorithmic issues. Lower level designs can be tedious, error prone, are harder to simulate and may be technology specific. High level design, on the other hand, allows for easy exploration of variations of the architectural and algorithmic parameters.

The design process is normally broken down into a number of phases, each phase converting the design from a higher level of abstraction to a lower one. One such phase would be the conversion of the behavioral description into a Control-Data Flow Graph. The scheduling phase, which has a tremendous impact on the the final design quality, is concerned with producing detailed timing information for all the operations. This thesis is concerned with the automation of the scheduling phase. Traditionally, scheduling algorithms have been heuristic in nature. Recently, there has been a lot of interest in producing schedules which are time and/or area optimal. In this thesis, techniques for computing such schedules will be presented, and analyzed. Bearing in mind that computing these schedules can be computationally very expensive, methods to improve the efficiency of exact scheduling algorithms are presented. Techniques for producing bounds on the area and time requirement of schedules will also be presented. These methods will allow designers to quickly eliminate points in the design space which will not produce optimal quality designs. Finally, a method of producing time optimal schedules which is superior to integer linear programming will be presented.

Chapter 1

Introduction and Motivation

In this thesis, the scheduling problem in *High Level Synthesis* (behavioral synthesis) will be examined, and several algorithms that compute exact solutions for this problem will be discussed. This problem is an extremely important phase of behavioral synthesis as this is probably the phase contributing most to the efficient utilization of the area available. With the rise in the complexity of integrated circuits (IC's) and their design, the design process has become very difficult to manage without the effective use of good CAD tools. CAD tools help automate a large portion of the design process, allowing the designer to abstract away from the low-level considerations (such as layout generation). This has lead to a reduction in the design time and has enabled designers to handle more complex (i.e., higher integration levels) designs. However, it is not just complex IC's which benefit from the use of CAD tools; applications which require specialized IC's (ASIC's), whose design may not otherwise be justified due to the low volume required, the use of CAD tools has reduced the design effort to a point where ASIC's have not only become practical but are rapidly becoming a significant market segment. Another major advantage of the usage of these tools is that it allows the designer to study the effect of variation of design parameters on the final design very easily – this is called design space exploration. All in all, the use of good tools can lead to a substantial reduction in the effort required to produce superior designs.

High level synthesis is the translating a behavioral (or high level) description of an hardware algorithm into an equivalent register-transfer level design, subject to certain criteria. A behavioral description of a circuit allows the designer to concentrate on the algorithm, and abstract away from the structural or the physical details. The behavior of the circuit is normally described using a Hardware Descriptor Language (e.g. VHDL or Verilog), which allow the designer to specify design constraints like concurrency, timing, structural interconnections, and interfaces.

The process of converting a behavioral description of the circuit into the actual physical implementation is often broken down into a number of phases. In the first phase, the HDL is examined, and a number of high level optimizations like code motion, common subexpression elimination, retiming, loop unrolling, and modifications taking advantage of the associativity and commutativity of operations are performed. This code is then converted to simpler intermediate code. Typically this intermediate code is more structural, in that it reduces the behavioral description into simpler operations which can be performed by standard components (such as addition or multiplication), and timing constraints between these operations. These timing constraints are normally relative, and are required to ensure that the data that these simpler operations require is ready before the operation is executed. The designer normally does not have to worry about the design of these standard components. However, in many cases, the designer has access to many libraries, and hence to many implementations of these resources, with varying area requirement, power consumption, and speeds. The phase in which the various implementations are examined to determine which one is most suitable for the design is called the *module selection phase*. Area and other considerations (like power) dictate the number of such resources that can actually be put on the chip, and hence it becomes very important to efficiently use the available resources or to minimize the resources required. The next phase—scheduling—is the one which is discussed in this thesis. This phase assigns a starting time to every operation, ensuring that when an operation begins to execute, all the data required for that operation is ready, and that at no time are more operations executing than can be executed on the resources available. Once the operations are scheduled, the *binding* phase assigns operations to instances of the resources (such as adders, registers, multipliers, etc.).

Important optimization criteria for the designer are (among others) the latency of the implementation and the area required to implement the design. It will be assumed that module selection has been done, and so, we know the area requirement and the speeds of all the resources. As far as the scheduling phase is concerned, the criteria of time and area are in a sense conflicting because it may be possible to improve the latency of the implementation if more resources are available thus increasing the area needed to implement the design in. Therefore the designer faces a number of tradeoffs and has to iterate the design procedure possibly a large number of times to arrive at the best possible implementation.

The scheduling problem plays a key role in the high-level synthesis part of the design automation system. Gajski et al. [GDP86] call it "perhaps the most important step during structure synthesis." In this thesis, two problems are considered

- The *Resource Constrained Scheduling* (RCS) Problem, *i.e.,* minimizing the latency of an implementation, given a fixed set of resources.

- *Time Constrained Scheduling* (TCS) Problem, *i.e.,* minimizing the area of an implementation, given a maximum latency.

Another problem, the *Time-Resource Constrained Scheduling* (TRCS) is concerned with deciding if a feasible solution exists when both latency and resource constraints are specified. This problem can easily be solved by recasting it either as a RCS problem or as a TCS problem, and so it is not considered.

The scheduling algorithms presented in this thesis will be only for specifications with no conditionals, and no loops. Even with these restrictions, both these problems are known to be NP-complete [Ull75] in general, and so the existence of a polynomial time algorithm for solving this problem is highly unlikely. In many high-level synthesis systems, heuristics are employed since the designer has to perform many iterations of the entire design process in a limited amount of time. These algorithms are generally fast and there has been a lot of work in the development of good heuristics [PK89, MPC90, CW91, Lin97]. Many years of research into the area of heuristic development has led to heuristics which do yield optimal solutions a considerable percentage of the time, while returning reasonable solutions in most of the other cases. However, in general these algorithms do not produce optimal solutions.

Even in those cases where they produce an optimal solution, the methods do not guarantee that the solution produced is indeed optimal; so one would have to use other techniques to verify optimality. We refer to the process of computing the optimal solutions or solutions whose optimization criteria are provably within some percentage of the optimal value as *Exact Scheduling*. This performance guarantee on the schedule is very important for several reasons.

- Given that there is a limit on the number of transistors that we can put on a IC and the speeds at which we can clock it, it has become extremely important to efficiently use the area (and hence the resources) available.
- Major performance improvements are most often algorithmic in nature, rather than technological. Given that we are trying to implement some particular algorithm (which we assume is as good as possible), we try to improve the next best thing — the scheduling of the algorithm.
- Another advantage arises due to the concept of design reuse. Many VLSI implementations are often improved to include more features, or to run faster. Most times, the new chip design simply consists of a modification of the old design. If the designer knows that the schedule is as good as it can get, then (s)he will not have to waste time trying to improve the schedule. Given that it can sometimes take an exceedingly long time to verify the absence of a better solution, even for simple circuits on powerful workstations, having to repeatedly do this can unnecessarily prolong the design cycle.

Because of the increased computational power available on most modern workstations, it is now feasible to explore the design space using exact algorithms. Blythe and Walker [BW95, BW96] have shown that complete design space exploration can in fact be made considerably more efficient by the use of good bounding strategies. It is shown that the use of bounding reduces the number of iterations of the complete scheduling algorithm that has to be performed. However, even with such techniques, the time taken to solve a few iterations of the scheduling algorithm can be large for medium sized problems, and so it is still very important to derive fast exact scheduling algorithms. Most exact algorithms transform the scheduling problem into an integer linear programming model, which is then solved by a general purpose ILP solver. Even though ILP can be computationally very intensive, Chaudhuri and Walker [CW94] and Gebotys and Elmasry [GE93] have shown that efficient formulations and the application of polyhedral theory can yield quick solutions for small and medium sized problems.

In this thesis, algorithms that compute optimal solutions to the scheduling problem are presented and analyzed. Of course, it should be kept in mind that it is imperative that these solutions be produced reasonably quickly, and in cases where this is not possible, schedules that are guaranteed to meet some some performance criteria should be produced. While there may be many models capable of representing the problem adequately in that an optimal solution can be obtained by solving the model using generic techniques, in practice, some models are more efficiently "solvable" than others. We will discuss both ILP and non-ILP based models. For ILP based models, We will be concerned with the process of solving such a model, and ways to improve the efficiency of this process. While it is possible to solve the ILP model using a generic ILP solver, using a custom ILP solver that takes advantage of information that cannot be easily encoded in the model will considerably enhance the model solution efficiency. It will be shown that there is a lot of problem specific information that can be incorporated into the model solver, and that doing so actually improves the performance considerably. Bounding the optimization criteria is extremely important for generating guarantees, and hence we will also discuss bounding techniques in detail.

The rest of the thesis is organized as follows. In Chapter 2 some of the mathematical preliminaries and notation is presented. We will attempt to compute exact solutions by computing upper bounds and lower bounds on the value that the objective function can achieve, given the constraints. Chapters 3 and 4 discuss various upper bounding techniques (heuristics) and lower bounding schemes respectively for this problem. Chapter 5 presents exact scheduling algorithms, and Chapter 6 lists the conclusions and avenues for further research.

Chapter 2

Mathematical Preliminaries and Notation

In this chapter, some mathematical background in graph theory and in integer linear programming is presented. Notation that will be used throughout this thesis along with some known results is also presented. We conclude this chapter with a discussion of the basic scheduling model.

2.1 Graph Theory

A directed graph $G = (T, E)$ is an ordered pair consisting of a set of vertices T and a set of edges E. An element $e \in E$ is an ordered pair $\langle t_i, t_j \rangle$. The first element of this ordered pair is known as the tail of the edge e and is denoted by $\mathsf{Tail}(e)$; the other element is the head of e and is denoted by $\mathsf{Head}(e)$. E can also be thought of as a relation on T. A Directed Acyclic Graph (DAG) is one in which the relation E is irreflexive. A subgraph of $G' = (T', E')$ of G is one satisfying a) $T' \subseteq T$ b) $E' \subseteq E$ and c) $E' \subseteq T' \times T'$. A path in a DAG is a sequence of edges $\langle e_1, e_2, \ldots e_n \rangle$, such that $\mathsf{Tail}(e_{i+1}) = \mathsf{Head}(e_i) \quad \forall 1 \leq i \leq n-1$. $\mathsf{Head}(e_n)$ is called the head of the path and $\mathsf{Tail}(e_1)$ the tail of the path. There is said to be a path from vertex t_i to vertex t_j if there is a path $P = \langle e_1, \ldots, e_n \rangle$ with $\mathsf{Tail}(P) = t_i$ and $\mathsf{Head}(P) = t_j$. The length of a path P is the number of edges in it. We will denote by $\Gamma_G(t_i, t_j)$ the length of the longest path from t_i to t_j in G. t_i is said to be an immediate predecessor of t_j if there is an edge $e \in E$ with $\mathsf{Head}(e) = t_j$ and $\mathsf{Tail}(e) = t_i$. t_i is an immediate successor of t_j if t_j is an immediate predecessor of t_i. t_i is a predecessor of t_j if there is a path $P \in G$ with $\mathsf{Head}(P) = t_j$ and $\mathsf{Tail}(P) = t_i$. t_i is a successor of t_j if t_j is a predecessor of t_i.

Let $T' \subseteq T$. Then $G' = (T', E')$ is said to be the subgraph of G induced by T' if

$$E' = \{e \in E : \mathsf{Head}(e) \in T' \text{ and } \mathsf{Tail}(e) \in T'\}$$

For any vertex $t_i \in T$, we denote by $G_{pred}(t_i)$ ($G_{succ}(t_i)$) the subgraph induced by the set of predecessors (successors) of t_i.

The predecessor relation induces a partial order on the vertices of G. This partial order can be extended to a complete order and such a complete ordering is called a *topological ordering (numbering)*. Notice that there may be many topological orderings for G.

Given a graph $G = (T, E)$, we define G^r, the reverse graph of G as the graph obtained by reversing all the (directed) edges. So $G^r = (T, E^r)$ with, $E^r = \{\langle t_i, t_j \rangle : \langle t_j, t_i \rangle \in E\}$.

2.2 Linear and Integer Programming

Much of the material in this section is from [NW88]. The *linear programming* problem is

$$Z_{LP} = \min\{cx : Ax \leq b, \quad x \in \Re^n\}$$

Here, $x = (x_1, x_2, \ldots, x_n)$ and A is an $m \times n$ matrix. There are polynomial time algorithms to solve this problem [NW88]. When we add constraints requiring that the elements of the vector x be integral, then we have an *Integer Linear Programming* problem. This can be expressed as

$$Z_{ILP} = \min\{cx : Ax \leq b, \quad x \in \mathcal{Z}_+^n\}.$$

Now, $x = (x_1, x_2, \ldots, x_n)$ and A is an $m \times n$ matrix. There are no known polynomial time algorithms to solve this problem. Any point in the space which satisfies the constraints is called a feasible point. Given a constrained optimization problem such as the one above, relaxing some of the constraints will lead to a solution which is a lower bound on the value of the original problem. A commonly used relaxation for the (ILP) problem is its *Linear Relaxation*, in which all integrality constraints are discarded. The Linear Relaxation of the (ILP) problem can be expressed as

$$Z_{LR} = \min\{cx : Ax \leq b, \quad x \in \Re^n\},$$

and of course, we have $Z_{LR} \leq Z_{ILP}$.

Given a set of points $S \subseteq \Re^n$, a point $x \in \Re^n$ is a convex combination of the points of S if there exists a finite set of points $\{x^i\}_{i=1}^t \subseteq S$ and a $\lambda \in \Re_+^t$ with $\sum_{i=1}^t \lambda^i = 1$, and $x = \sum_{i=1}^t \lambda^i x^i$. The *convex hull* of S, denoted by $conv(S)$, is the set of all points that are convex combinations of the points in S.

A polyhedron $P \subseteq \Re^n$ is the set of points that satisfy a finite number of linear inequalities i.e. $P = \{x \in \Re^n : Ax \leq b\}$, where A is an $m \times n$ matrix and b a n element row vector. An inequality $\pi x \leq \pi_0$ is a valid inequality if it is satisfied by all points in P.

If $P = \{x \in \Re^n : Ax \leq b\}$ and $S = P \cap \mathcal{Z}^n$, then $conv(S)$ is a polyhedron. Therefore, it is possible (in principle) to solve a problem of the form

$$\max\{cx : x \in P \cap \mathcal{Z}_+^n\}$$

by solving a linear programming problem

$$\max\{cx : x \in conv(S)\}$$

We will call $conv(P \cap \mathcal{Z}^n)$ the integer hull of P, denoted by $integer(P)$. Further, we use $integer(A, b)$ to refer to the integer hull of the polyhedron $\{x : Ax \leq b, \quad x \in \Re^n\}$.

If for some (ILP) problem $\min\{cx : x \in P\}$ where $P = \{x : Ax \leq b, \quad x \in \mathcal{Z}_+^n\}$, the matrix A is such that every square submatrix in A has determinant in $\{-1, 0, 1\}$, then the matrix A is called *totally unimodular* then we have $integer(P) = P$ and the solution to this (ILP) problem is the same as the solution to its linear relaxation. We sometimes refer to P as (A, b).

Addition of an inequality which is satisfied by all the elements in $integer(P)$, but is not satisfied by at least one element of P will lead to a more efficient formulation of the ILP problem as the Linear Relaxation of the problem with these inequalities added will yield tighter bounds. Such an inequality would be a valid inequality for $integer(P)$.

2.3 Solving the ILP Model

In this section, some of the commonly used approaches for solving the (ILP) problem will be discussed. We will assume that we are trying to solve the ILP problem

$$Z^P_{ILP} = \min\{cx : x \in P, \quad x \in \mathcal{Z}^n_+\}$$

where P is a polyhedron. We will label the vector x corresponding to the the optimal solution X^P_{ILP}. Further we use Z^P_{LR} to refer to the bound returned by the linear relaxation to this problem, and X^P_{LR} will denote the vector corresponding to this minimum solution.

2.3.1 Cutting Plane techniques

In this approach, we start of by solving the linear relaxation of the ILP problem. If all the variables are integral, then we are done. If however, one or more of the variables are non integral, then clearly the solution vector corresponding to the linear relaxation problem is not in $integer(P)$. Hence, we add extra constraints (cutting planes) to the original ILP formulation which are violated by this particular solution. Notice that such constraints always exist. These cuts eliminate some of the non-integral optimal solutions (to the LR problem) but leave integral solutions intact, and so provide a tighter ILP formulation. The linear relaxation of the new ILP formulation is then solved, and the entire process is repeated until an integral solution is found. The cuts introduced are of two types:

General purpose cuts (or Gomory cuts). The technique used to generate these cuts are valid for any problem, and these were the cuts used in traditional cutting plane approaches [NW88, PR88, SM89]. However, in practice, a pure cutting plane approach using Gomory cuts has not had much success in solving large ILP problems [Wil85]. The main problem with this approach is that it takes many iterations to converge to an integral solution, and that the repeated addition of cutting planes tends to introduce numerical errors due to finite precision effects which accumulate due to the many iterations required. Further, the number of cuts introduced can be enormous.

Problem-specific cuts. These cuts are very problem-specific, and they exploit the problem structure. Since they are specifically designed for the problem, it stands to reason that these cuts can be stronger (i.e., they can remove larger portions of the polyhedron not part of the $integer(P)$). Recent experiments using problem specific cuts show that these are much more effective than Gomory cuts. These cuts, when combined with a branch and bound approach, can greatly improve the speed and the size of the problem that can be solved.

2.3.2 Logic Programming Based Techniques

For ILP problems in which all integer variables are either 0 or 1, there are a number of pseudo-boolean techniques to solve the problem. These techniques are essentially enumerative in nature, and use the logical structure of the problem. See [Bar95] for additional details.

2.3.3 Branch and Bound

Until recently, branch and bound was the technique that had proved most successful on large ILP problems. We begin with a description of a generic branch and bound technique, and then illustrate how this is technique is used to solve an ILP problem.

Let ϕ be the set of feasible solutions to the problem. Further, for any $x \in \phi$, let $\mathsf{obj}(x)$ denote the objective function that we are attempting to minimize. Note that for a (integer) linear programming formulation, this simply corresponds to $c \cdot x$ for some $c \in \Re^n$. We partition the set ϕ into sets $\phi_1, \phi_2, \ldots, \phi_m$ so that $\phi = \phi_1 \cup \phi_2 \cup \ldots \cup \phi_m$. Now, each subset of the set of feasible solutions can be individually solved, and hence the optimal solution in ϕ will be the minimum of the optimal solutions in all of the sets $\phi_1, \ldots, \phi_m$. This scheme is actually a total enumeration scheme. However, if we are able to avoid solving some of these subproblems, then we get a partial enumeration scheme. Clearly, we can avoid solving a subproblem, say finding the optimal solution in ϕ_k, only if we know that ϕ_k has no optimal solutions. Two ways of showing this are :

- If we can show that ϕ_k contains no feasible solutions at all, then ϕ_k cannot contain any optimal solution. Then ϕ_k is called infeasible.
- Suppose that ξ is the best solution that we have found so far. This solution may have been found either by using some other heuristic or by solving one of the other subproblems. Suppose, further, that we can show $\mathsf{obj}(\xi) \leq \mathsf{obj}(\psi) \quad \forall \psi \in \phi_k$. Then we know that there cannot be any solution in ϕ_k better than the ξ, and so there is no point actually finding the best solution in ϕ_k.

A branch and bound algorithm is defined as a "partial enumeration procedure employing tests of feasibility and comparison to an incumbent solution to fathom candidate problems" [PR88].

For the ILP problem, subproblems correspond naturally to pairwise disjoint polyhedra $P_1, P_2, \ldots, P_m$, such that all the elements in $integer(P)$ are contained in $P_1 \cup P_2 \cup \ldots \cup P_m$. The ILP problem of finding Z_{ILP} can then be cast as

$$Z_{ILP}^{P} = \min_{i=1}^{m} \left\{ Z_{ILP}^{P_i} = \min\left\{ cx : x \in P_i \quad x \in \mathcal{Z}_{+}^{n} \right\} \right\}$$

where finding $Z_{ILP}^{P_k}$ is the k^{th} subproblem. Now, we apply "pruning criteria of feasibility, optimality, and value dominance" [NW88]. Let us consider these one at a time.

- *Feasibility:* This occurs when there is no solution to the linear relaxation of this subproblem. Clearly this would mean that there is no integral vector in the polyhedron P_k corresponding to this subproblem since this polyhedron is empty.

- *Optimality:* If $X_{LR}^{P_k}$, the vector corresponding to the solution of the linear relaxation of this problem is integral, then the solution to this subproblem has been obtained.

- *Value Dominance :* When $Z_{LR}^{P_k}$ is greater than $c \cdot X$, where X represents the best integral solution found so far, then the polyhedron P_k cannot contain an optimal (integer) solution, and may be ignored.

Let us now consider the branching strategy.

On solving the linear relaxation of the ILP problem, and obtaining X_{LR}^P and Z_{LR}^P, if we see that X_{LR}^P is not integral, we pick some component of X_{LR}^P which is not integral. Let $X_{LR}^P[i]$ be the i^{th} component. If $X_{LR}^P[i]$ is non integral, we then partition the polyhedron into two polyhedrons

$$P_1 = \left\{x \in P : x[i] \geq \left\lceil X_{LR}^P[i] \right\rceil \right\}$$

and

$$P_2 = \left\{x \in P : x[i] \leq \left\lfloor X_{LR}^P[i] \right\rfloor \right\}$$

Notice that the vector X_{LR}^P is simply a vector of the variables in the ILP formulation. Partitioning P as above corresponds to branching on the variable corresponding to the i_{th} component of X_{LR}^P.

2.4 Schedule Models

We model the set of operations (tasks) as the vertex set T of a DAG G. As mentioned before, each operation $t_i \in T$ requires some data to operate on, and so may not be scheduled before all the data required is ready. This data may be produced by other operations and so t_i may not begin executing until all the other operations which produce some data that is required for t_i have completed execution. This sets up a precedence relation E between the operations in T which is modeled as the set of directed edges of G. For an operation $t_i \in T$, an incoming edge from another operation t_j means that t_i requires some data which is produced by t_j, and hence the number of incoming edges is the number of operands that t_i has. This is also called the fan-in of operation t_i. In practice, operations of T have bounded (and small) fan-in. Hence, if k is the maximum of the fan-in of all the operations in T, then $k \cdot |T|$ is an upper bound on the number of edges in this graph. So the graph is a sparse graph.

It should be mentioned that we will only be dealing with synchronous designs, and so each operation can only begin executing at some time $c \cdot CLOCK$ where $CLOCK$ is the clock cycle length. Hence we can assume that time is discretized, and all times specified will be in clock steps. Each operation t_i takes $\mathsf{delay(t_i)}$ time steps to complete executing. Each dependency $\langle t_i, t_j \rangle \in E$ implies that operation $t_i \in T$ has to be completed before operation t_j can begin executing. So, if operation t_i begins executing at time step k, then $\langle t_i, t_j \rangle \in E \Rightarrow$ operation t_j can begin executing only at or after time step $k + \mathsf{delay}(t_i)$. Each edge in $\langle t_i, t_j \rangle \in E$ has a weight equal to $\mathsf{delay}(t_i)$.

We assume that there are a number of resource classes (adders, multipliers, etc.) and that each operation can execute only on a resource belonging to a particular class. We assume that the number of such resource classes is R and that if any resource constraints are specified, then they are of the form that the number of simultaneously active (allocated to some task) resources of class r is no greater than M_r. We will let the function $\mathsf{Resource} : T \to \{1, 2, \ldots, R\}$ be a mapping of operations to resource classes. We say that a schedule $\mathcal{S}$ satisfies these resource constraints if

$$|\{t_i \in T : \mathsf{Resource}(t_i) = r \text{ and } \mathcal{S}(t_i) = l\}| \leq M_r \qquad \forall\, 1 \leq r \leq R, \quad \forall\, l$$

A schedule of G is a function $\mathcal{S} : T \to \mathcal{Z}^+$, where each operation t_i is scheduled to begin execution at time step $\mathcal{S}(t_i)$. The latency of a schedule is defined to be the maximum completion time of all operations under that schedule. Therefore, the latency of a schedule $\mathcal{S}$ is given by

$$\mathsf{Latency}(\mathcal{S}) = \max_{t_i \in T} \{\mathcal{S}(t_i) + \mathsf{delay}(t_i) - 1\}$$

Throughout this thesis, it will be assumed that all operations take exactly one time step to complete execution. There is no loss of generality in such an assumption, because, given a graph $G = (T, E)$, in which some of the operations in T require more that one time step to complete execution, we can always create another graph $G' = (T', E')$, such that an optimal schedule for G' corresponds to an optimal schedule for G. This is done by replacing all operations t in T with $\mathsf{delay}(t) > 1$ with the set of operations $\left\{t_1, t_2, \ldots, t_{\mathsf{delay}(t)}\right\}$. Then edges of the form $\langle t, u\rangle \in E$ are replaced with $\langle t_{\mathsf{delay}(t)}, u\rangle$ and edges of the form $\langle u, t\rangle$ are replaced with $\langle u, t_1\rangle$. Finally we add edges $\langle t_i, t_{i+1}\rangle \quad 1 \leq i \leq delay(t) - 1$ to get E'.

With this in mind, we can assume that all operations complete in one cycle or $\mathsf{delay}(t_i) = 1 \quad \forall t_i \in T$. Therefore, the definition of $\mathsf{Latency}$ becomes

$$\mathsf{Latency}(\mathcal{S}) = \max_{t_i \in T} \{\mathcal{S}(t_i)\}$$

Note that we will refer to T' as T for the rest of the thesis.

A latency constraint simply specifies that the latency of the schedule should be less than or equal to some specified limit.

A *valid* schedule is a schedule which satisfies all the dependence constraints. We will require optimal schedules which also satisfy other criteria (i.e. resource constraints or latency constraints). A schedule which satisfies the additional criteria specified is called a *feasible* schedule.

We use $T_{pred}(t_i)$ (resp. $T_{succ}(t_i)$) to denote the set of direct and indirect predecessors (resp. successors) of task t_i. $G_{pred}(t_i)$ (resp. $G_{succ}(t_i)$) is the subgraph induced by $T_{pred}(t_i)$ (resp. $T_{succ}(t_i)$). Clearly, all elements in $T_{pred}(t_i)$ must have completed execution before t_i can be scheduled, and no operation in $T_{succ}(t_i)$ can begin execution before t_i completes executing.

Longest weighted directed paths in G are called *critical paths* of G. The length of a critical path of G is denoted by $\mathsf{CP}(G)$. A lower bound on the execution time of G is clearly $\mathsf{CP}(G) + 1$.

2.5 ILP Models

In this section, ILP models for the scheduling problem are presented. Most algorithms which yield an exact (optimal) solution model the scheduling problem using Integer Linear Programming, and the ILP model is then solved using a generic ILP solver to get the optimal schedule. See Lin [Lin97] for a recent survey on techniques for high-level synthesis. Many different ILP models have been proposed [CW91, HLH91, HH93a, HH93b]. Chaudhuri and Walker [CW93, CW94] and Gebotys and Elmasry [GE90, GE93] have shown that efficient formulations lead to reduced execution times for the ILP approach. In this section the ILP formulations due to [HLH91, CW94] and [GE93] will be described.

2.5.1 The Scheduling ILP model

We start with the formulation described in [HLH91]. For each operation $t_i \in T$, $\mathsf{TimeFrame}(t_i)$ is the set of time steps in which the operation must be scheduled. One possibility is to set $\mathsf{TimeFrame}(t_i) = \{1, 2, \ldots, UB\}$, where UB is an upper bound on the number of steps required[1]. This means that in every valid schedule t_i cannot be scheduled in any of the steps in $\mathcal{Z}^+ - \mathsf{TimeFrame}(t_i)$. Therefore, we only have to decide the step $\in \mathsf{TimeFrame}(t_i)$ to which t_i is to be scheduled. We must decide this for all the operations in T. A set of decision variables, $\{y_{ik} : k \in \mathsf{TimeFrame}(t_i)\}$ is associated with each operation t_i. The variable y_{ik} is 1 if and only if operation t_i is scheduled to time step k. Since t_i must be scheduled to one and only one step, we have

$$\sum_{k \in \mathsf{TimeFrame}(t_i)} y_{ik} = 1.$$

Resource constraints are modeled as follows

$$\sum_{\mathsf{Resource}(t_i)=\ell} y_{ik} \leq M_\ell \qquad \forall k \in Z.$$

A dependence constraint $\langle t_i, t_j \rangle$ is modeled by the inequality

$$\sum_{k \in \mathsf{TimeFrame}(t_j)} k \cdot y_{jk} - \sum_{k \in \mathsf{TimeFrame}(t_i)} k \cdot y_{ik} \;\; \geq \;\; 1$$

This model is extremely simple and does not take into account the structure of the scheduling polytope. It has been shown [NW88] that the introduction of valid inequalities and the extraction of integral facets can result in formulations that are more easily and more efficiently solvable. Chaudhuri and Walker [CW94] and Gebotys and Elmasry [GE93] present models that differ from this simple model. Both models differ only in their modeling of precedence constraints. We have used the model described in [CW94] and so we describe this model next.

Chaudhuri and Walker [CW94] model the precedence constraint as

[1]The procedure for refining this definition will be presented in Chapter 3

$$\sum_{\substack{k \in \mathsf{TimeFrame}(t_i) \\ k \geq \ell}} y_{ik} + \sum_{\substack{k \in \mathsf{TimeFrame}(t_j) \\ k \leq \ell}} y_{jk} \leq 1 \qquad \begin{array}{c} \forall \ell \in \mathsf{TimeFrame}(t_i) \cap \mathsf{TimeFrame}(t_j) \\ \forall \langle t_i, t_j \rangle \in E \end{array} .$$

Chaudhuri and Walker [CW94] show that this model can be viewed as the intersection of two integral polytopes.

Gebotys and Elmasry [GE93] discuss a model based on the node packing polytope [NW88]. They have demonstrated the advantages of this particular model by exploiting the known facets of this problem, along with other properties unique to this problem. Both the above formulations have been shown to yield better times than the simple formulation due to Hwang et al. [HLH91].

One common problem with most of the ILP based techniques is that a large number of variables may be required if the upper bound on the optimal latency UB is large. In the worst case, the number of variables is $O(|T| \cdot UB)$. This may cause the ILP solver to take a long time to solve the problem [Wal97]. We have noticed that even with efficient formulations, tight resource constraints result in ILP models that take inordinately large amounts of time to solve. In Chapter 5, techniques to reduce the time taken to solve the ILP model is presented. Non-ILP techniques are also presented.

Chapter 3

Upper Bounding Techniques

Much of the early work in the area of scheduling for high level synthesis has been targeted toward developing good heuristics. Heuristics have the advantage of being fast (at least most of them do) while producing results that are close to the optimal solutions most of the time. This thesis is primarily concerned with the computation of exact schedules, and as such it would seem that there is no role for a heuristic algorithm in an exact solution calculation procedure. However, any schedules produced by these algorithms represent upper bounds on the optimizing criteria (assuming of course that we are minimizing the objective function), and so as will be shown in Chapter 5, even though these techniques do not guarantee optimality of the solutions produced, these solutions produced can be used in the calculation of optimal solutions. The ASAP and the ALAP schedules will also be presented and analyzed. These schedules will be shown to be especially important for calculating the time frames of all the operations, as well as for calculating upper bounds on the resource requirement of schedules. Some of the algorithms (both exact and heuristic) presented will require that these schedules be computed before hand.

In this chapter, the ASAP and the ALAP are formally described and analyzed. The close relationship between these two schedules is also shown. Besides this, some heuristic algorithms are presented. This chapter does not purport to describe all existing heuristics for this problem. However, some of the commonly used heuristic techniques are described in detail, especially those which will be used for the computation of exact solutions.

3.1 Scheduling in the absence of constraints

In this section the "As Soon As Possible" (ASAP) and the "As Late As Possible" (ALAP) schedules are examined. Roughly speaking, the ASAP schedule can be considered to be the solution to the problem of finding a schedule for which each operations is scheduled as early as possible, given an unlimited set of resources. Similarly, the ALAP schedule can be considered the solution to the problem of finding a schedule which has the lowest possible latency, while scheduling each operations to the latest step possible, given an unlimited set of resources.

It will be shown that these two schedules are extremely closely related, and that the procedure to calculate the ASAP can in fact also be used to calculate the ALAP.

Procedure $\mathsf{ASAP}(T,\, E)$
Compute topological numbering of $G = (T, E)$
for $i \leftarrow 1$ **to** $|T|$ **do**
  $t_i \leftarrow$ task whose topological numbering is i
  $\mathsf{ASAP}(t_i) \leftarrow$ earliest step after all predecessors of t_i
enddo

Figure 3.1: The ASAP scheduling Algorithm

All schedules considered will be at least valid, i.e., they satisfy all the dependence constraints. It should be noted that the ASAP and the ALAP schedules, though valid, may not actually be feasible for the RCS problem[1].

3.1.1 The ASAP schedule

In this section, a characterization of the ASAP schedule is shown. Then an algorithm for computing this schedule is presented.

Proposition 3.1 *The* ASAP *schedule is given by*

$$\mathsf{ASAP}(t_i) = \begin{cases} \mathsf{CP}(G_{pred}(t_i)) + 1 & t_i \in T, T_{pred}(t_i) \neq \phi \\ 1 & t_i \in T, T_{pred}(t_i) = \phi \end{cases}$$

Proof. Suppose that there is some dependence constraint $\langle t_i, t_j \rangle$. Then

$$\mathsf{CP}(G_{pred}(t_j)) \geq \mathsf{CP}(G_{pred}(t_i)) + 1$$

Hence, the ASAP schedule as defined satisfies all the precedence constraints. So it is a valid schedule.

A lower bound on the schedule time of any operation $t_i \in T$ (for all valid schedules) is clearly $\mathsf{CP}(G_{pred}(t_i)) + 1$. Since the $\mathsf{ASAP}(t_i)$ is precisely this value, it satisfies the stated requirement. □

A simple algorithm to calculate the ASAP schedule is shown in Figure 3.1. Since $|E|$ is $O(|T|)$, the topological sort will require $O(|T|)$ time, and hence the entire algorithm has $O(|T|)$ time complexity.

3.1.2 The ALAP schedule

Proposition 3.2 *The* ALAP *schedule is given by*

$$\mathsf{ALAP}(t_i) = \mathsf{CP}(G_{succ}(t_i)) + 1 \qquad \forall t_i \in T$$

[1]Note that they will always be feasible for any TCS problem with at least one solution

PROOF. It can be seen that all the dependence constraints are satisfied. Hence the schedule as defined is valid. Now, for every operation $\mathsf{ASAP}(t_i) \leq \mathsf{CP}(G)$. Therefore, this schedule has the minimum possible latency. Further, it can be seen that every operation is scheduled to the latest step possible without increasing the latency of the schedule and so this schedule satisfies the requirements specified. □

To see that the procedure to calculate the ASAP can in fact also be used to compute the ALAP schedule, one must examine the reverse graph of G.

Proposition 3.3 *If a valid schedule $\mathcal{S}$ for a graph $G = (T, E)$ satisfies some resource and/or time constraints, then the schedule $\mathcal{S}^r$ for G^r defined as $\mathcal{S}^r(t_i) = \mathsf{Latency}(\mathcal{S}) - \mathcal{S}(t_i) + 1$ is valid, and satisfies all the resource and/or time constraints satisfied by $\mathcal{S}$.*

PROOF. To show the validity of $\mathcal{S}^r$, we must show that all dependencies in G^r are satisfied. Let $\langle t_i, t_j \rangle \in E^r$. Then $\langle t_j, t_i \rangle \in E$, and by the validity of $\mathcal{S}$, we have $\mathcal{S}(t_i) > \mathcal{S}(t_j)$. Therefore, $\mathcal{S}^r(t_j) - \mathcal{S}^r(t_i) = \mathsf{Latency}(\mathcal{S}) - \mathcal{S}(t_j) + 1 - \mathsf{Latency}(\mathcal{S}) + \mathcal{S}(t_i) - 1$. Therefore, $\mathcal{S}^r(t_j) > \mathcal{S}^r(t_i)$. Hence $\mathcal{S}^r$ is a valid schedule.

To show that any resource constraint satisfied by $\mathcal{S}$ is also satisfied by $\mathcal{S}^r$, we simply need to show that the maximum number of simultaneously active resources of any one resource class is the same for both schedules. This is obviously true because $\mathcal{S}(t_i) = \mathcal{S}(t_j) \Leftrightarrow \mathcal{S}^r(t_i) = \mathcal{S}^r(t_j)$.

Finally, $\mathcal{S}^r(t_i) = \mathsf{Latency}(\mathcal{S}) + 1 - \mathcal{S}(t_i) \leq \mathsf{Latency}(\mathcal{S})$. Hence all latency constraints are also satisfied. □

Notice that the Latency(ASAP for G^r) = Latency(ALAP). Further, the ASAP schedule for G^r, the reverse graph of G is in fact the ALAP schedule for G. This leads to the fact that the ALAP schedule can be simply calculated as the ALAP schedule for the reverse graph.

3.2 Scheduling with Latency Constraints

It should be noted that the ASAP (ALAP) schedules represent the earliest (latest) possible time steps to which operations can be scheduled when it is required that the schedule complete in the minimum possible time. Suppose we define some different constraints on the set of schedules. Let ϕ be the set of schedules which satisfy these constraints. We would like to define

$$\mathsf{ASAP}_{ideal}(\phi, t_i) = \min_{t_i \in T, \mathcal{S} \in \phi} \mathcal{S}(t_i)$$

and

$$\mathsf{ALAP}_{ideal}(\phi, t_i) = \max_{t_i \in T, \mathcal{S} \in \phi} \mathcal{S}(t_i)$$

With this definition, our original ASAP and ALAP schedules can be represented as

$$\mathsf{ASAP}(t_i) = \mathsf{ASAP}_{ideal}(\phi_1, t_i)$$

and

$$\mathsf{ASAP}(t_i) = \mathsf{ALAP}_{ideal}(\phi_1, t_i)$$

where ϕ_1 represents the set of valid schedules (i.e. the set of schedules which respect all the dependences). In the presence of a $\mathsf{Latency}$ constraint requiring all schedules to complete before UB, let us define $\mathsf{ASAP}(UB, t_i) = \mathsf{ASAP}(t_i)$, and $\mathsf{ALAP}(UB, t_i) = \mathsf{ALAP}_{ideal}(\phi_2, t_i)$ where ϕ_2 is the set of valid schedules which have latencies less than or equal to UB. Further, $\mathsf{ALAP}(UB, t_i) = \mathsf{ALAP}(t_i) + UB - \mathsf{Latency}(\mathsf{ALAP})$. Notice that this reflects the fact the $\mathsf{ALAP}(t_i, UB)$ does not exist for $UB < \mathsf{Latency}(\mathsf{ASAP})$. These schedules are denoted by $\mathsf{ASAP}(UB)$ and $\mathsf{ALAP}(UB)$. So, $\mathsf{ASAP}(UB, t_i)$ and $\mathsf{ALAP}(UB, t_i)$ represent the earliest and the latest possible time steps to which t_i may be scheduled in any valid schedule whose latency is less than or equal to UB. This range of steps is called the $\mathsf{TimeFrame}$ of the operation. So, $\mathsf{TimeFrame}(t_i) = \{\mathsf{ASAP}(UB, t_i), \ldots, \mathsf{ALAP}(UB, t_i)\}$. The $\mathsf{TimeFrame}$ of an operation represents the set of steps to which it may be scheduled to in a valid schedule whose $\mathsf{Latency}$ is $\leq UB$. It should be pointed out that calculating the $\mathsf{ALAP}_{ideal}(\phi)$ and the $\mathsf{ASAP}_{ideal}(\phi)$ schedules is as difficult as computing the schedule in ϕ with the minimum $\mathsf{Latency}$. For an arbitrary set ϕ, this problem can easily be seen to be NP-complete. Calculation of the $\mathsf{TimeFrames}$ of operations is required for many heuristics and exact solution techniques. We therefore, allow approximations to these schedules, and define $\mathsf{ASAP}_{approx}(\phi)$ and $\mathsf{ALAP}_{approx}(\phi)$ as

$$\mathsf{ASAP}_{approx}(\phi, t_i) = \text{ Lower bound on } \min_{t_i \in T, \mathcal{S} \in \phi} \mathcal{S}(t_i)$$

and

$$\mathsf{ALAP}_{approx}(\phi, t_i) = \text{ Upper bound on } \max_{t_i \in T, \mathcal{S} \in \phi} \mathcal{S}(t_i)$$

and we can redefine

$$\mathsf{TimeFrame}(t_i) = \{\mathsf{ASAP}_{approx}(UB, t_i), \ldots, \mathsf{ALAP}_{approx}(UB, t_i)\}$$

Now, we can be sure that operation t_i cannot be scheduled to some step not in $\mathsf{TimeFrame}(t_i)$ in any valid and optimal schedule. Most scheduling algorithms simply use the first definition of $\mathsf{TimeFrame}$. In Chapter 4, we will show how to calculate $\mathsf{ASAP}_{approx}(\phi, t_i)$ and $\mathsf{ALAP}_{approx}(\phi, t_j)$.

3.2.1 Minimizing Resource Utilization

For the TCS problem, we require a schedule which minimizes the resource utilization, given some latency constraint. Clearly the $\mathsf{ASAP}(UB)$ and $\mathsf{ALAP}(UB)$ represent upper bounds on the resource utilization of all such schedules, as does any valid schedule whose latency is $\leq UB$. What we want to do is to reduce the resource utilization by as much as possible, while ensuring that the $\mathsf{Latency}$ constraint is not violated. That is, we wish to reduce the (maximum) concurrency of the schedule by as much as possible, with the constraint that the schedule be completed within the specified latency. In this section we will discuss two commonly used heuristics for the TCS problem, namely, List scheduling and Force-directed scheduling [PK87, PK89]. For the remainder of this section, we will be attempting to find the optimal (minimal) resource utilization of schedules whose latency is $\leq UB$.

Procedure List-Scheduler(T, E, $\mathsf{Priority}$, $\mathsf{BestStep}$)
for $i \leftarrow 1$ **to** $|T|$ **do**
  $t_i \leftarrow$ unscheduled operation with the highest $\mathsf{Priority}$.
  $\mathcal{S}(t_i) \leftarrow \mathsf{BestStep}(t_i)$
  Update $\mathsf{Priority}$ Function if required.
enddo

Figure 3.2: Generic List Scheduling Algorithm

3.2.2 List Scheduling

A list scheduling algorithm is simply a scheduler which schedules operations sequentially according to a priority list. In general, the priority list is dynamic, and has to be recalculated after each operations is scheduled. Every operations is scheduled to some "best step", which is computed based on the previously scheduled operations, and possibly some other factors. A simple list scheduler is shown in Figure 3.2.

A list scheduling algorithm is therefore defined by its priority function and its "best step" selection strategy.

A commonly used priority function is based on the $\mathsf{TimeFrame}$ of the operations. At every step, we pick an operation $t_i \in T$ for which $|\mathsf{TimeFrame}(t_i)|$ is the least. It is then scheduled to a step s for which the number of operation of belonging to resource class $\mathsf{Resource}(t_i)$ already scheduled is minimum. If $\mathcal{S}(t_i) = s$, then for all $t_j \in T_{succ}(t_i)$, we would like every element of $\mathsf{TimeFrame}(t_j)$ to be $\geq \Gamma(t_i, t_j) + \mathcal{S}(t_i)$ and for all $t_j \in T_{pred}(t_i)$, we would like every element of $\mathsf{TimeFrame}(t_j)$ to be $\leq \mathcal{S}(t_i) - \Gamma(t_j, t_i)$. By doing this, we can be sure that the $\mathsf{TimeFrame}$ of all unscheduled operations is such that the the existence of a feasible schedule for any possible assignment of any *one* of the remaining operations to any step in their updated $\mathsf{TimeFrame}$ is guaranteed.

3.2.3 Force Directed Scheduling

A major problem with the $\mathsf{TimeFrame}$-based list scheduler is that it does not distinguish between operations whose $\mathsf{TimeFrames}$ only contain steps to which a lot of other operations must be scheduled to and operations whose $\mathsf{TimeFrames}$ contain steps which to which (relatively) few other operations must be scheduled to. Force directed scheduling is an attempt to make the decision of scheduling a operations to a step depend on other the $\mathsf{TimeFrames}$ of other operations as well. The basic idea is as follows. If we assume that an operation t_i is equally likely to be scheduled to any of the steps in its $\mathsf{TimeFrame}$, then, $\frac{1}{|\mathsf{TimeFrame}(t_i)|}$ represents the probability that it will be scheduled to any one particular step. A similar probability can be calculated for each operations. Now, since we have the probability that any operations will be assigned to a particular step, we would like to calculate the expected value of the number of operations assigned to that step. Now, while it is true that scheduling the operations are not really independent events, it is extremely difficult to calculate this expectation if we do not make this assumption. Under this (possibly invalid) assumption, we can calculate the expectation of the number of operations belonging to resource class r

Procedure TimeFrameBasedScheduler(T, E)
Calculate $\mathsf{TimeFrames}$ for all the operations in T
for $i \leftarrow 1$ **to** $|T|$ **do**
  $t_i \leftarrow$ unscheduled operation with the least $|\mathsf{TimeFrame}|$.
  $s \leftarrow$ step to which the number of operations of class
    $\mathsf{Resource}(t_i)$ already scheduled is least.
  $\mathcal{S}(t_i) \leftarrow s$
  For all t_j in $T_{pred}(t_i)$ **do**
    $\mathsf{TimeFrame}(t_j) \leftarrow \mathsf{TimeFrame}(t_j) \cap \{1, 2, \ldots s - \Gamma(t_j, t_i)\}$
  enddo
  For all t_j in $T_{succ}(t_i)$ **do**
    $\mathsf{TimeFrame}(t_j) \leftarrow \mathsf{TimeFrame}(t_j) \cap \{s + \Gamma(t_j, t_i), \ldots \infty\}$
  enddo
enddo

Figure 3.3: TimeFrame Based List Scheduling Algorithm

Procedure ForceDirectedSchedule(T, E)
Calculate $\mathsf{TimeFrame}$'s for all the operations in T.
while (not all operations scheduled) **do**
  Calculate $NUM(s)\ \forall s \in \{1, 2, \ldots, UB\}$
  Calculate $\mathsf{force}(t_i, s)\ \forall t_i \in T\ \forall s \in \mathsf{TimeFrame}(t_i)$
  Pick (t_i, s) such that $\mathsf{force}(t_i, s) \leq \mathsf{force}(t_j, s')$
    $\forall \quad t_j \in T, s' \in TimeFrame(t_j)$
  $\mathcal{S}(t_i) \leftarrow s$
  $\mathsf{TimeFrame}(t_i) \leftarrow \{s\}$
  For all $t_j \in T_{pred}(t_i)$ **do**
    $\mathsf{TimeFrame}(t_j) \leftarrow \mathsf{TimeFrame}(t_j) \cap \{1, 2, \ldots s - \Gamma(t_j, t_i)\}$
  enddo
  For all $t_j \in T_{succ}(t_i)$ **do**
    $\mathsf{TimeFrame}(t_j) \leftarrow \mathsf{TimeFrame}(t_j) \cap \{s + \Gamma(t_j, t_i), \ldots, \infty\}$
  enddo
enddo

Figure 3.4: Force Directed Scheduling

being scheduled to a step s as $NUM(r,s) = \sum_{\substack{s \in \mathsf{TimeFrame}(t_i) \\ \mathsf{Resource}(t_i)=r}} \frac{1}{|\mathsf{TimeFrame}(t_i)|}$. Now, our goal is to minimize the maximum concurrency over all the steps. The average concurrency of the entire algorithm is the same for all schedules, since the number of operations is fixed as is the latency of the schedule. The maximum concurrency over all the steps would be minimized if all the steps had the same number of operations assigned. This can be understood by seeing that we would like to (as far as possible) distribute the work load evenly to all steps, because this would cause the best utilization of resources. If the number of resources required at one particular step is a lot more than the remaining steps, then this would result in inefficient utilization of the resources available. So, we are interested (in a sense) in reducing the work load on steps s with large $NUM(s)$ and in increasing the workload in steps s' with a small $NUM(s')$. So, if we are scheduling some operation t_i, we would obviously like to schedule it to a step s such that $NUM(s)$ is minimum of all steps in $\mathsf{TimeFrame}(t_i)$. What we would like do is to quantify the reduction in the maximum concurrency that scheduling an operation to a particular step has. Suppose that some operation t_i is scheduled to a step s. Then the increase in concurrency at step s' is (probabilistically speaking)

$$\delta_s(t_i, s') = \begin{cases} 1 - \frac{1}{|\mathsf{TimeFrame}(t_i)|} & \text{if } s = s' \\ \frac{-1}{|\mathsf{TimeFrame}(t_i)|} & \text{if } s \neq s' \end{cases}$$

The "self-force" of an operation is a measure of the reduction in the maximum concurrency (or a measure of the equalization of the work load) as a result of scheduling t_i to a step s, and we define

$$\mathsf{force}(t_i, s) = \sum_{s' \in \mathsf{TimeFrame}(t_i)} \delta_s(t_i, s') \cdot NUM(\mathsf{Resource}(t_i), s').$$

Notice that this penalizes scheduling operations to heavily populated steps, and rewards scheduling operations to lightly populated steps. The algorithm is shown in Figure 3.4. In later chapters, we present techniques to tighten the TimeFrames of the operations. Using these tightened TimeFrames can sometimes result in better schedules as shown in Tables 3.1, 3.2 and 3.3.

3.3 Scheduling with Resource Constraints

Since we have a constraint on the latency of schedules for the TCS problem, we can compute the TimeFrame as specified. For the RCS problem, we can define the $\mathsf{ASAP}_{ideal}(\phi)$ as before, where ϕ is the set of valid schedules which satisfy the resource constraints. As before, since this schedule cannot always be computed fast, we can approximate it with $\mathsf{ASAP}_{approx}(\phi)$. However, $\mathsf{ALAP}_{ideal}(\phi)$ schedule poses a problem. If we use the same definition, then we get $\mathsf{ALAP}_{ideal}(\phi, t_i) = \infty \quad \forall t_i \in T$. So, we modify the ϕ we are considering. We are interested in finding a schedule in ϕ with minimum Latency. Note that any schedule in ϕ represents an upper bound on this latency. The schedule $\mathcal{S} \in \phi$ can be generated by a heuristic. Let

$$\phi_1 = \{\mathcal{S}' \in \phi : \mathsf{Latency}(\mathcal{S}') \leq \mathsf{Latency}(\mathcal{S})\}$$

Table 3.1: Upper Bounds for the *FDCT* benchmark

Latency	FDS	FDS with tight TimeFrames
8	52	52
9	63	63
10	33	33
11	28	28
12	58	33
13	28	28
14	28	28
15	23	23
16	23	23
17	23	23
18	23	23
19	23	23
20	23	23
21	23	19
22	14	14
23	19	14
24	16	16
25	14	14
26	19	14
27	14	14
28	14	14
29	14	14
30	14	14
31	14	14

Table 3.2: Upper Bounds for the AR benchmark

Latency	FDS	FDS with tight TimeFrames
11	34	34
12	44	39
13	46	32
14	24	24
15	34	24
16	44	24
17	24	24
18	24	24
19	24	19
20	24	19
21	19	19
22	19	19
23	19	19
24	19	19
25	19	19
26	14	14
27	19	14
28	22	17
29	17	17
30	12	12
31	12	12
32	12	12
33	12	12
34	12	12

Table 3.3: Upper Bounds for the *EWF* benchmark

Latency	FDS	FDS with tight TimeFrames
17	21	21
18	26	26
19	19	19
20	14	14
21	14	14
22	19	19
23	14	14
24	9	9
25	9	9
26	9	9
27	14	14
28	14	9

Now,

$$\min_{\mathcal{S}' \in \phi}\{Latency(\mathcal{S}')\} = \min_{\mathcal{S}' \in \phi_1}\{Latency(\mathcal{S}')\}$$

Further, $\mathsf{ALAP}_{ideal}(\phi_1)$ has a meaningful definition, and so $\mathsf{ALAP}_{approx}(\phi_1)$ and TimeFrame can be given meaningful definitions.

We will only discuss list scheduling algorithms for this problem. One of the few polynomial time algorithms known for latency minimization involves scheduling a tree-structured graph [Hu61]. Hu showed that scheduling using the ALAP value as a priority function results in a minimum length schedule. Since this priority function produces optimal results for trees, it seems reasonable to expect that this will also perform well for generic DAGs, and this is indeed a very commonly used priority function. One advantage of scheduling using this priority function is that it is guaranteed that all the predecessors of an operation will be scheduled before the operation is scheduled. This is especially important because, for this problem, as the number of resource instances is fixed, we wish to ensure that after every scheduling decision we make, we can always complete the schedule, without violating any of the resource constraints or the precedence constraints. If both successors and predecessors of some operation t_i were scheduled before t_i is scheduled, then it can be seen that we might be left with no freedom about which step to schedule t_i to. This naturally leads to the possibility of using other topological orderings as the priority function. An algorithm using a priority function which is also a topological ordering is shown in Figure 3.5. As such, it is very difficult to theoretically compare different priority functions. However, we show the results of using two different priority functions (which will be generated using techniques presented in later chapters) and the values of the optimal solution in Tables 3.4, 3.5 and 3.6

Procedure RCSListScheduler(T, E, Priority)
for $i \leftarrow 1$ **to** $|T|$ **do**
  $t_i \leftarrow$ unscheduled operation with the highest Priority.
  $s \leftarrow$ smallest step in Timeframe(t_i)
  $\mathcal{S}(t_i) \leftarrow s$
  For all $t_j \in T_{pred}(t_i)$ **do**
    TimeFrame$(t_j) \leftarrow$ TimeFrame$(t_j) \cap \{1, 2, \ldots s - \Gamma(t_j, t_i)\}$
  enddo
  For all $t_j \in T_{succ}(t_i)$ **do**
    TimeFrame$(t_j) \leftarrow$ TimeFrame$(t_j) \cap \{s + \Gamma(t_j, t_i), \ldots, \infty\}$
  enddo
enddo

Figure 3.5: Resource Constrained Scheduling Algorithm

Table 3.4: Heuristic Solutions (RCS) for the $FDCT$ benchmark

$*_p$	$*_m$	ALU	Optimal	Priority Function 1	Priority Function 2
0	1	1	34	34	34
0	1	2	34	34	34
0	2	1	26	26	26
0	2	2	18	18	18
0	2	3	18	18	18
0	3	2	14	14	14
0	4	2	13	13	13
0	4	3	11	11	11
0	4	4	11	11	11
0	4	5	11	11	11
0	5	4	10	10	10
0	5	5	10	10	10
0	8	4	08	08	08
1	0	1	26	26	26
2	0	1	26	26	26
1	0	2	19	19	19
2	0	2	13	13	13
3	0	2	13	13	13
2	0	3	12	12	12
3	0	3	10	10	10
4	0	4	09	09	09
4	0	8	08	08	08

Table 3.5: Heuristic Solutions(RCS) for the *ARF* benchmark

$*_p$	$*_m$	ALU	Optimal	Priority Function 1	Priority Function 2
0	1	1	34	34	34
0	1	2	34	34	34
0	2	1	18	18	18
0	2	2	18	18	18
0	2	3	15	15	15
0	3	1	16	16	16
0	4	1	16	16	16
0	4	2	11	11	11
1	0	1	19	19	19
1	0	2	19	19	19
1	0	3	19	19	19
2	0	2	13	13	13
2	0	3	13	13	13
3	0	2	13	13	13
4	0	2	11	11	11

Table 3.6: Heuristic Solutions(RCS) for the *EWF* benchmark

$*_p$	$*_m$	ALU	Optimal	Priority Function 1	Priority Function 2
0	1	1	28	28	28
0	1	2	21	22	21
0	2	1	28	28	28
0	2	2	18	18	18
0	2	3	18	18	18
0	3	2	18	18	18
0	3	3	17	17	17
1	0	1	28	28	28
1	0	2	19	19	19
1	0	3	18	18	18
2	0	1	28	28	28
2	0	2	18	18	18
2	0	3	18	17	17

3.4 Estimation of the quality of schedules

The algorithms described in this chapter are essentially heuristics, and are not guaranteed to return optimal solutions. However, they do tend to return solutions that are optimal or close to optimal most of the time. This, however, does not imply that we cannot use these algorithms if we require a solution that is optimal. We can use these techniques, along with some other information to generate guarantees on the quality of the solution. Typically, guarantees on the quality of the solution can be obtained in one of three ways

- The schedule generating process provides a guarantee of the optimality of the schedule; clearly none of the algorithms described above do so. Algorithms which do so will be discussed in greater detail in Chapter 5.
- Quality bound can be derived for the schedule generated For example, if the schedule generated takes no more time than the critical path of the graph for a RCS problem, or if a particular resource is used to its capacity at all the time steps, then it is clear that such a schedule would be exact, if it satisfied all the other constraints.
- The schedule generated has latency equal to a lower bound on the latency of the schedule generated through other techniques.

We will describe these techniques in greater detail in the rest of the chapters.

Chapter 4

Lower Bounding Techniques

In this chapter, some techniques for computing a lower bound on the latency of the optimal schedule for the RCS problem, and a lower bound on the number of resources required for the TCS problem will be presented. A comparison between the quality of the bounds generated through the various techniques will also be presented. This is especially important because, the quality of the bounds generated determine the time taken by the exact schedulers present in Chapter 5.

4.1 Previous Work

4.1.1 Bounds for the RCS Problem

In this section, we will assume that the number of instances of resource class r available is M_r. We will let ϕ be the set of valid schedules which satisfy all the resource constraints. Further, we will let Φ represent the set of schedules

$$\Phi = \{\mathcal{S}' \in \phi : \mathsf{Latency}(\mathcal{S}') \leq \mathsf{Latency}(\mathcal{S})\}$$

where $\mathcal{S}$ is some element of ϕ.

Suppose we associate a step called the *release step* with each operation $t \in T$. The release step represents a lower bound on the step at which an operation can be scheduled. One possible set of values could be the ASAP times (i.e., the release time of operation t_i is taken to be $\mathsf{ASAP}(t_i)$). We will let $\mathsf{Release} : T \rightarrow \mathcal{Z}^+$ be the function mapping operations to their release steps of the operations. Let Θ represent the set of valid schedules for which $\mathcal{S}(t_i) \geq \mathsf{Release}(t_i)\ \forall t_i \in T$ and which satisfy all resource constraints. Notice that a schedule in Θ would also be in Φ if it also satisfied all dependency constraints, i.e., if $\mathcal{S}(t_i) \geq \mathcal{S}(t_j)\ \forall \langle t_j, t_i \rangle \in E$. Clearly $\Phi \subseteq \Theta$. Therefore,

$$\min_{\mathcal{S} \in \Theta} \{\mathsf{Latency}(\mathcal{S})\} \leq \min_{\mathcal{S} \in \Phi} \{\mathsf{Latency}(\mathcal{S})\}.$$

So, $\min_{\mathcal{S} \in \Theta} \{\mathsf{Latency}(\mathcal{S})\}$ is a lower bound on the latency of any schedule $\mathcal{S} \in \Phi$. The algorithm given in Figure 4.1 is from [RJ94] and it computes this lower bound in $O(\mid T \mid)$ time.

We had earlier defined the $\mathsf{ASAP}_{ideal}(\Phi)$ ($\mathsf{ALAP}_{ideal}(\Phi)$) value of an operation $t \in T$ to be the earliest (latest) possible time step to which the operation can be scheduled. What

Sort the operations in order of increasing ALAP values.
$latency\_estimate \leftarrow 0$;
AVAILABLE-RESOURCES(l, r) $\leftarrow M_r\ \forall\, 1 \leq r \leq R \quad \forall\, l$
for $i \leftarrow 1$ **to** $|T|$ **do**
  $t \leftarrow i^{th}$ operation in T in order of increasing ALAP value;
  $k \leftarrow$ ASAP value of operation t;
  while (AVAILABLE-RESOURCES(k, $\mathsf{Resource}(t)$) $= 0$) **do**
    $k \leftarrow k + 1$;
  enddo
  Decrement AVAILABLE-RESOURCES(k, $\mathsf{Resource}(t)$);
  $latency\_estimate \leftarrow \max\left(latency\_estimate, k + \max_{t_i \in T_{succ}(t)} \{\Gamma(t, t_i)\}\right)$;
enddo

Figure 4.1: Lower Bound Estimation

this means is that for any valid schedule $\mathcal{S}$, $\mathsf{ASAP}_{ideal}(t) \leq \mathcal{S}(t) \leq \mathsf{ALAP}_{ideal}(t)$. In practice, we normally use ASAP, which is generated by scheduling the graph, after ignoring all the resource constraints. So the actual values of ASAP are lower bounds on the actual ASAP_{ideal} values.

So, for any $\mathcal{S} \in \Phi$ and $t \in T$, $\mathsf{ASAP}(t) \leq S(t)$. However, if for some $t \in T$, $\mathsf{ASAP}(t) < \mathcal{S}(t)\ \forall \mathcal{S} \in \Phi$, then the ASAP value is not as tight as it could be.

An operation cannot begin execution before all of its predecessors (direct and indirect) have completed executing. Therefore, another lower bound on the execution step of an operation is the minimum time taken to completely execute $G_{pred}(t)$. So, if the algorithm in [RJ94] returns l as a lower bound on the latency of $G(t)$ then, for any $\mathcal{S} \in \Phi$ and $t \in T$, $l \leq \mathcal{S}(t)$. This idea is used to develop tighter bounds in [LC96]. The algorithm in [RJ94] is called recursively to compute (possibly) *better* values of ASAP for all the operations, and then the [RJ94] algorithm can be used with the *new and improved* ASAP as the release step.

4.1.2 Bounds for the TCS problem

In this section, we will assume that the maximum allowable latency is UB. Let ϕ be the set of schedules $\{\mathcal{S} : \mathsf{Latency}(\mathcal{S}) \leq UB\}$. Now,

$$\mathsf{TimeFrame}(t_i) = \{\mathsf{ASAP}(UB, t_i), \ldots, \mathsf{ALAP}(UB, t_i)\}$$

represents the set of steps to which $t_i \in T$ can be scheduled to any schedule in ϕ. We will call a set of consecutive steps $\{l, l+1, \ldots, u-1, u\}$ an interval, and denote it by $[l, u]$. Therefore, $\mathsf{TimeFrame}(t_i) = [\mathsf{ASAP}(UB, t_i), \mathsf{ALAP}(UB, t_i)]$. Now, consider some interval $[l, u] \subseteq [1, UB]$. Define

$$T_{[l,u]} = \{t \in T : \mathsf{TimeFrame}(t) \subseteq [l, u,]\}$$

So, $T_{[l,u]}$ is the set of operations that *must* be scheduled in the interval $[l, u]$. Define $T^r_{[l,u]} = \left\{t \in T_{[l,u]} : \mathsf{Resource}(t) = r\right\}$. Clearly, a lower bound on the number of instances of resource class r required would be $\left\lceil \frac{|T^r_{[l,u]}|}{|[l,u]|} \right\rceil$. This is the idea behind the bounding schemes presented in [FB73a] and in [SJ93]. These schemes return

$$\max_{[l,u] \subseteq [1,UB]} \left\lceil \frac{|T^r_{[l,u]}|}{|\,[l,u]\,|} \right\rceil$$

as the lower bound for the number of instances of resource class r.

4.2 Bounds on Resource-Constrained Scheduling (RCS)

In this section, we will obtain bounds on the time taken to schedule all the tasks in T, given constraints on the number of resources of each type available. Our strategy for obtaining bounds will be to discard all the precedence constraints, and then calculate the optimal value of the latency of the schedule; this will be a lower bound on the optimal latency of schedules that preserve the precedence constraints.

Before we describe how to compute this bound, we will formulate a simpler problem and then describe a technique to compute the optimal solution to this simpler problem, and then show how this procedure can be applied to solve the problem at hand.

4.2.1 Scheduling Independent Operations with Release Times and Resource Constraints

In this section, we will consider the problem of scheduling independent tasks with release times. Let $T = \{t_1, t_2, \ldots, t_N\}$ be the set of operations to be scheduled. Each task $t_i \in T$ has an associated release time $\mathsf{Release}(t_i)$, and so the operation cannot begin executing before time step $\mathsf{Release}(t_i)$. We initially assume that all operations execute on the same type of resource, i.e., there is only one resource class, and that each resource of this class has unit delay. The number of instances of this resource available is M_1. Further, we assume that we are given another function $\theta : T \rightarrow \mathcal{Z}^+$, and we seek a schedule $\mathcal{S} : T \rightarrow \mathcal{Z}^+$ such that the value of

$$\mathsf{Deficiency}(\mathcal{S}) = \max_{t_i \in T} \left(\mathcal{S}(t_i) + \theta(t_i)\right)$$

is minimized.

Without loss of generality, we assume that the operations in T have been ordered in decreasing order of $\theta(t_i)$. So,

$$\theta(t_1) \geq \theta(t_2) \geq \theta(t_3) \geq \ldots \geq \theta(t_N)$$

```
Procedure SimpleBound(T, Release, θ, M_1)
NumberResAvailable[step] ← M_1            ∀step ∈ Z^+
Deficiency ← 0

for i ← 1  to N do
    step ← Release(t_i)
    while (NumberResAvailable[step] = 0) do
        step ← step + 1
    enddo
    S(t_i) ← step
    NumResAvailable[step] ← NumResAvailable[step] − 1
    Deficiency ← max(Deficiency, step + θ(t_i))
enddo
return Deficiency
```

Figure 4.2: Algorithm to minimize Deficiency

Theorem 4.1 *The algorithm shown in Figure 4.2 produces a schedule $\mathcal{S}$ such that the value of* Deficiency($\mathcal{S}$) *is minimum.*

PROOF. Let $\mathcal{S}(t_i)$ be the time step that operation t_i is assigned to by the algorithm in Figure 4.2. Define FirstDiffer(A, B) where A and B represent schedules as follows.

$$\mathsf{FirstDiffer}(A,B) = \begin{cases} 1 & \text{if } A(t_1) \neq B(t_1) \\ 1 + \left(\max\limits_{\forall 1 \leq j \leq i \; A(t_j) = B(t_j)} i \right) & \text{otherwise.} \end{cases}$$

Let $\mathcal{OPT}$ be the set of all schedules $\mathcal{S}$ for which Deficiency($\mathcal{S}$) is minimized. Pick $\mathcal{O} \in \mathcal{OPT}$ such that $\forall \mathcal{O}' \in \mathcal{OPT}$ we have

$$\mathsf{FirstDiffer}(\mathcal{S}, \mathcal{O}) \geq \mathsf{FirstDiffer}(\mathcal{S}, \mathcal{O}')$$

The outline of our proof is as follows. We will argue by contradiction, showing that if $\mathcal{O} \neq \mathcal{S}$ (i.e., FirstDiffer($\mathcal{O}, \mathcal{S}$) $< N + 1$) then we can always construct another optimal schedule (in terms of minimizing Deficiency) $\mathcal{O}'$ from $\mathcal{O}$ with FirstDiffer($\mathcal{O}, \mathcal{S}$) $<$ FirstDiffer($\mathcal{O}', \mathcal{S}$), thereby contradicting the way $\mathcal{O}$ was chosen.

Let $m =$ FirstDiffer($\mathcal{S}, \mathcal{O}$). Let $Oper(\mathcal{O}, step)$ be the set of operations assigned to time *step*. So

$$Oper(\mathcal{O}, step) = \{t_j | \mathcal{O}(t_j) = step\}.$$

We must have $\mathcal{S}(t_m) < \mathcal{O}(t_m)$. There are two cases.

Case (a): The number of operations scheduled at the same step as t_m is less than the number of available resources, i.e., $|Oper(\mathcal{O}, \mathcal{S}(t_m))| < M_1$.

Since $\mathcal{S}(t_m) \geq \mathsf{Release}(t_m)$, operation t_m can be rescheduled to step $\mathcal{S}(t_m)$. Modify schedule $\mathcal{O}$ by rescheduling operation t_m to step $\mathcal{S}(t_m)$ to obtain schedule $\mathcal{O}'$. Clearly $\mathsf{Deficiency}(\mathcal{O}') \leq \mathsf{Deficiency}(\mathcal{O})$. Therefore, $\mathcal{O}' \in \mathcal{OPT}$. Further, $\mathsf{FirstDiffer}(\mathcal{S}, \mathcal{O}) < \mathsf{FirstDiffer}(\mathcal{S}, \mathcal{O}')$. This violates our assumption and hence, this case is not possible.

Case (b): All units of the resource are allocated to operations that are scheduled to the same step as t_m, i.e., $|Oper(\mathcal{O}, \mathcal{S}(t_m))| = M_1$.

Then, there must be at least one $t_k \in \{t_{m+1}, t_{m+2}, \ldots, t_N\}$ for which $\mathcal{O}(t_k) = \mathcal{S}(t_m)$. Since $\theta(t_m) \geq \theta(t_m + 1) \geq \ldots \geq \theta(t_N)$, we must have $\theta(t_m) \geq \theta(t_k)$. Operation t_k can be scheduled to time step $\mathcal{O}(t_m)$ because

$$\mathsf{Release}(t_k) \leq \mathcal{O}(t_k) = \mathcal{S}(t_m) \leq \mathcal{O}(t_m)$$

Further, operation t_m can be scheduled to time step $\mathcal{O}(t_k) = \mathcal{S}(t_m)$. Create a new schedule $\mathcal{O}'$ with

$$\mathcal{O}'(t_i) = \begin{cases} \mathcal{O}(t_i) & \text{if } i \neq m \text{ and } i \neq k \\ \mathcal{O}(t_m) & \text{if } i = k \\ \mathcal{O}(t_k) & \text{if } i = m \end{cases}$$

In other words, $\mathcal{O}'$ is the same as $\mathcal{O}$ with the exception that tasks m and k have their positions swapped. Note that $\mathcal{O}'$ is a correct schedule in that all operations are scheduled only after their Release time and no resource constraints are violated.

$$\mathcal{O}'(t_k) + \theta(t_k) = \mathcal{O}(t_m) + \theta(t_k) \leq \mathcal{O}(t_m) + \theta(t_m)$$

$$\mathcal{O}'(t_m) + \theta(t_m) = \mathcal{O}(t_k) + \theta(t_m) \leq \mathcal{O}(t_m) + \theta(t_m)$$

Therefore, $\mathsf{Deficiency}(\mathcal{O}') \leq \mathsf{Deficiency}(\mathcal{O})$ and so $\mathcal{O}'$ is an optimal schedule with $\mathsf{FirstDiffer}(\mathcal{S}, \mathcal{O}) < \mathsf{FirstDiffer}(\mathcal{S}, \mathcal{O}')$, which is a contradiction.

Therefore the algorithm in Figure 4.2 always computes the optimal value of the function specified. $\square$

Lemma 4.2 *If* $\mathsf{Release}_1(t_i) \geq \mathsf{Release}_2(t_i)\ \forall t_i \in T$, *then*

$$\mathsf{SimpleBound}(T, \mathsf{Release}_1, \theta, M_1) \geq \mathsf{SimpleBound}(T, \mathsf{Release}_2, \theta, M_1)$$

PROOF. $\mathsf{SimpleBound}(T, \mathsf{Release}, \theta, M_1) =$

$$\mathsf{Deficiency}(\mathcal{S}) = \min_{\mathcal{S}:\, \forall t_i, \mathcal{S}(t_i) \geq \mathsf{Release}(t_i)} \left(\max_{t_i} (\mathcal{S}(t_i) + \theta(t_i)) \right)$$

So if $\mathsf{Release}_1(t_i) \geq \mathsf{Release}_2(t_i)\ \forall t_i \in T$, then since

$$\{\mathcal{S} :\ \forall t_i, \mathcal{S}(t_i) \geq \mathsf{Release}_1(t_i)\} \subseteq \{\mathcal{S} :\ \forall t_i, \mathcal{S}(t_i) \geq \mathsf{Release}_1(t_i)\}$$

we must have $\mathsf{SimpleBound}(T, \mathsf{Release}_1, \theta, M_1) \geq \mathsf{SimpleBound}(T, \mathsf{Release}_2, \theta, M_1)$ $\square$

Lemma 4.3 *If* $\theta_1(t_i) \geq \theta_2(t_i)\ \forall t_i \in T$, *then*

$$\mathsf{SimpleBound}(T, \mathsf{Release}, \theta_1, M_1) \geq \mathsf{SimpleBound}(T, \mathsf{Release}, \theta_2, M_1)$$

PROOF. For any schedule $\mathcal{S}$ and for all $t_i \in T$, we must have

$$\mathcal{S}(t_i) + \theta_1(t_i) \geq \mathcal{S}(t_i) + \theta_2(t_i)$$

Therefore, $\mathsf{SimpleBound}(T, \mathsf{Release}, \theta_1, M_1) \geq \mathsf{SimpleBound}(T, \mathsf{Release}, \theta_2, M_1)$ $\square$

The bounding procedure SimpleBound has been defined only for sets of operations that are all mapped onto the same resource class. Now, suppose the operations in T are mapped to R different resource classes. In this case, we partition T into R subsets $\{T_1, T_2, \ldots, T_R\}$ such that $T = T_1 \cup T_2 \cup \ldots \cup T_R$ and each partition T_j consists of all tasks mapped to resource class j, i.e., $\mathsf{Resource}(t_i) = j \quad \forall t_i \in T_j$. Given any function f, we will denote by $f|_{T_j}$ the restriction of f to the set T_j. We define the bound as

$$\mathsf{SimpleBound}(T, \mathsf{Release}, \theta) \quad = \quad \max_{1 \leq j \leq R} \mathsf{SimpleBound}(T_j, \mathsf{Release}|_{T_j}, \theta|_{T_j}, M_r)$$

4.2.2 Application to the original problem

Since we are considering bounds that can be obtained by relaxing precedence constraints, it is useful to (for this section alone) extend our definition of schedules. We will refer to all functions $\mathcal{S} : T \rightarrow \mathcal{Z}^+$ that satisfy the resource constraints, i.e.,

$$\left(\sum_{t_i \in T, type(t_i) = r, \mathcal{S}(t_i) = s} 1 \right) \leq M_r \qquad \forall s \in \mathcal{Z}^+$$

as schedules. We will refer to schedules that do not violate precedence constraints as valid schedules. We will also extend the definition of schedules to subsets of the set of tasks T. We will refer to $\mathcal{S}'$ as a valid schedule of $T' \subseteq T$ if it satisfies all precedences E' induced by T'. For any $T' \subseteq T$, we define

$$Opt(T') = \min_{\text{schedules } \mathcal{S} \text{ valid for } T'} \left(\max_{t_i \in T'} \mathcal{S}(t_i) \right).$$

Let us assume that UB is an upper bound on the latency of the optimal valid schedule, i.e., the schedule does not violate any precedence constraints. Any valid schedule is an upper bound on the latency of an optimal valid schedule, and so this value can be easily obtained as the latency of a heuristic solution.

We will derive results for two functions $\mathcal{H}_1$ and $\mathcal{H}_2$. Suppose we are given functions $\mathcal{H}_1$ and $\mathcal{H}_2$ which satisfy the following properties.

- $UB - Opt(G_{succ}(t_i)) \leq \mathcal{H}_2(t_i) \leq UB$

- $0 \leq \mathcal{H}_1(t_i) \leq Opt(G_{pred}(t_i))$

Note that the ASAP and the ALAP functions as we defined before satisfy these conditions for $\mathcal{H}_1$ and $\mathcal{H}_2$ respectively. Also notice that $\mathcal{H}_1$ and $\mathcal{H}_2$ need not be schedules as they need not satisfy the resource constraints.

Let $\Lambda(\mathcal{S}) = \max_{t_i \in T}(\mathcal{S}(t_i) + UB - \mathcal{H}_2(t_i))$ and $\lambda(\mathcal{S}) = \max_{t_i \in T} \mathcal{S}(t_i)$.

Theorem 4.4 *For any valid schedule $\mathcal{S}$, $\lambda(\mathcal{S}) = \Lambda(\mathcal{S})$.*

Proof. It is clear that for any schedule $\mathcal{S}$ of T, $\lambda(\mathcal{S}) \leq \Lambda(\mathcal{S})$. So we need only show the reverse inequality.

For any $t_i \in T$

$$\begin{array}{rcccc} & UB - Opt(G_{succ}(t_i)) & \leq & \mathcal{H}_2(t_i) & \leq & UB \\ \Rightarrow & Opt(G_{succ}(t_i)) - UB & \geq & -\mathcal{H}_2(t_i) & \geq & -UB \\ \Rightarrow & Opt(G_{succ}(t_i)) & \geq & UB - \mathcal{H}_2(t_i) & \geq & 0 \end{array}$$

Therefore, if t_i has no successors $\mathcal{H}_2(t_i) = UB$. Therefore, for all $t_i \in T$ with no successors, $\mathcal{S}(t_i) = \mathcal{S}(t_i) + UB - \mathcal{H}_2(t_i)$ for all schedules $\mathcal{S}$ of T.

Now, Suppose t_i has at least one successor.

$$\begin{array}{rcccc} & Opt(G_{succ}(t_i)) & \geq & UB - \mathcal{H}_2(t_i) & \geq & 0 \\ \Rightarrow & \mathcal{S}(t_i) + Opt(G_{succ}(t_i)) & \geq & \mathcal{S}(t_i) + UB - \mathcal{H}_2(t_i) & \geq & \mathcal{S}(t_i) \end{array}$$

In no schedule $\mathcal{S}$ of T can $\mathcal{S}(t_i) - \mathcal{S}(t_j) < Opt(G_{succ}(t_i)) \quad \forall t_j \in G_{succ}(t_i)$. So there must exist some t_j with no successors such that $\mathcal{S}(t_j) \geq \mathcal{S}(t_i) + Opt(G_{succ}(t_i))$. Let $\mathsf{NoSucc}(T) = \{t_i \in T : t_i \text{ has no successor }\}$. Therefore,

$$\begin{array}{rcl} \lambda(\mathcal{S}) & = & \max_{t_i \in \mathsf{NoSucc(T)}} (\mathcal{S}(t_i)) \\ & = & \max_{t_i \in \mathsf{NoSucc(T)}} (\mathcal{S}(t_i) + Opt(G_{succ}(t_i))) \\ & \geq & \max_{t_i \in T - \mathsf{NoSucc(T)}} (\mathcal{S}(t_i) + Opt(G_{succ}(t_i))) \end{array}$$

Therefore, for any schedule $\mathcal{S}$, $\Lambda(\mathcal{S}) \leq \lambda(\mathcal{S})$.

Therefore $\lambda(\mathcal{S}) = \Lambda(\mathcal{S})$. □

As a result, it does not really matter whether we optimize for $\lambda(\mathcal{S})$ or for $\Lambda(\mathcal{S})$ when computing exact solutions. However, the choice of the optimizing function does make a difference when we consider a relaxation where all the precedences are discarded. It can be seen that the inequality $\Lambda(\mathcal{S}) \geq \lambda(\mathcal{S})$ holds irrespective of the validity (in terms of precedence constraints) of $\mathcal{S}$. However, the two need not be equal if any precedences are violated in $\mathcal{S}$ as the $UB - \mathcal{H}_2(t_i)$ term is effectively a penalty associated with precedence violations. Therefore, when we relax all precedence constraints, the optimal value of Λ is a tighter bound than the optimal value of λ. Solving for an optimal value of Λ can be computed by calling the routine in Figure 4.3 with $\mathcal{H}_1$ as the Release function and $UB - \mathcal{H}_2$ as the θ function. The complete algorithm for calculating a lower bound on the latency of the optimal valid schedule given the functions $\mathcal{H}_1$ and $\mathcal{H}_2$ is shown in Figure 4.3. Table 4.1

Table 4.1: Bounds on the latency

Benchmark	$*_m$	ALU	Optimal Latency	Optimal λ	Optimal Λ
EWF	1	1	28	27	28
EWF	2	1	28	27	28
EWF	1	2	21	20	21
EWF	2	2	18	17	18
FDCT	1	1	34	33	34
FDCT	2	2	18	17	18
FDCT	2	3	18	17	18
FDCT	3	3	14	12	13
ARF	1	1	34	32	34
ARF	1	2	34	32	34
ARF	2	1	18	16	18
ARF	2	2	18	16	18
HAL	1	1	13	12	13
HAL	2	1	8	6	7
HAL	3	2	6	6	6

Procedure $\mathsf{LowerBound}(T,\ \mathcal{H}_1,\ \mathcal{H}_2,\ UB)$
for $i \leftarrow 1$ **to** $|T|$ **do**
  $\mathsf{Release}(t_i) \leftarrow \mathcal{H}_1(t_i)$
  $\theta(t_i) \leftarrow UB - \mathcal{H}_2(t_i)$
enddo
return $\mathsf{SimpleBound}(T,\ \mathsf{Release},\ \theta)$

Figure 4.3: Lower Bounding Routine

shows the results of solving for the optimal value of both λ and Λ for various benchmarks. It can be seen that solving for optimal Λ produces strictly tighter bounds in all of the cases.

4.2.3 Obtaining $\mathcal{H}_1$ and $\mathcal{H}_2$

As stated in Lemmas 3.1 and 3.2, the tightest bounds on the latency of optimal schedules of T can be obtained when the tightest values of the $\mathcal{H}_1$ and $\mathcal{H}_2$ functions that satisfy the requirements are used. The values of $\mathcal{H}_1(t_i)$ and $\mathcal{H}_2(t_i)$ are actually lower and upper bounds on the value of $\mathsf{ASAP}_{ideal}(\phi, t_i)$ and $\mathsf{ALAP}_{ideal}(\phi, t_i)$ respectively, where ϕ is the set of all schedules that satisfy all the precedence and all the resource constraints and $\mathsf{Latency}(\mathcal{S}) \leq UB$.

$\mathcal{H}_1 \leftarrow \mathsf{ASAP}_G$

for $i \leftarrow 1$ **to** $|T|$ **do**
  $t_i \leftarrow i^{th}$ operation in topologically sorted order
  $\mathcal{H}_1(t_i) \leftarrow \mathsf{LowerBound}\left(G_{pred}(t_i), \mathcal{H}_1|_{G_{pred}(t_i)}, \mathsf{ALAP}|_{G_{pred}(t_i)}, UB\right)$
enddo

Figure 4.4: Code fragment to calculate $\mathcal{H}_1$ (and $\mathcal{H}_2$)

Define $T(t_i, t_j) = T_{succ}(t_i) \cap T_{pred}(t_j)$ and a family of functions $\mathcal{H}_{2_j} : T_{pred}(t_j) \rightarrow \mathcal{Z}^+$ where $\mathcal{H}_{2_j}(t_i) = \mathsf{CP}(T(t_i, t_j))$.

Although we do not show it, it can be shown that tighter results can be obtained by defining $\mathcal{H}_{2_j}$ as

$$\mathcal{H}_{2_j}(t_i) = \mathsf{LowerBound}(T(t_i, t_j), \mathsf{ASAP} - \mathsf{ASAP}(t_i), \mathsf{ALAP}(t_j, UB) - \mathsf{ALAP}(UB), UB)$$

Now, we use the fact that operation t_i cannot be scheduled until all the operations in $G_{pred}(t_i)$ have been completed. It can be shown that

$$\mathsf{ASAP}_{ideal}(\phi, t_i) \geq \mathsf{LowerBound}(T_{pred}(t_i), \mathsf{ASAP}, \mathcal{H}_{2_i}, UB) \geq ASAP(t_i)$$

So we use the routine shown in Figure 4.4 to compute $\mathcal{H}_1$, which is a better bound for ASAP_{ideal} than is ASAP.

To compute the value of $\mathcal{H}_2$ we make use of the following lemma, which we proved in Chapter 3.

Lemma 4.5 *A lower bound on the* ASAP *function for a graph G is also an upper bound on the* ALAP *function for the graph G^r obtained from G by reversing all the edge directions (dependencies) and vice-versa.*

Proof. See Chapter 3. □

So we can use the same routine to compute the values of $\mathcal{H}_2$.

Lemma 4.6 *For any graph $G = (T, E)$ and any $\mathcal{H}_1 \geq$* ASAP*,*

$$\mathsf{LowerBound}(T, \mathcal{H}_1, \mathcal{H}_2, UB) \geq \mathsf{CP}(G)$$

PROOF. This result can be seen immediately from the fact that if $\mathcal{H}_1 \geq \mathsf{ASAP}$, then there is at least one operation t_i whose Release Time ($\mathcal{H}_1$) is equal to the Critical Path Length of T. □

Lemma 4.7 *The code fragment in Figure 4.4 calculates values of $\mathcal{H}_1$ and $\mathcal{H}_2$ which satisfy the required properties, and which are tighter than the* ASAP *and the* ALAP *functions.*

PROOF. From the code in Figure 4.4, it can be seen that $\mathcal{H}_1(t_i)$ is calculated only after $\mathcal{H}_1(t_j)$ has been calculated for all predecessors t_j of t_i. If t_i, has no predecessors, then the statement $\mathcal{H}_1(t_j) \geq \mathsf{ASAP}(t_j) \quad \forall t_j \in G_{pred}(t_i)$ holds vacuously. Therefore, the value calculated for $\mathcal{H}_1(t_i) \geq \mathsf{ASAP}(t_i)$ by Lemma 3.4. Now, suppose that for some t_i (with predecessors), $\mathcal{H}_1(t_j) \geq \mathsf{ASAP}(t_j) \quad \forall t_j \in G_{pred}(t_i)$ holds. Then again by Lemma 3.4, we have $\mathcal{H}_1(t_i) \geq \mathsf{ASAP}(t_i)$. Hence, by induction, $\mathcal{H}_1(t_i) \geq \mathsf{ASAP}(t_i) \quad \forall t_i \in T$.

The proof that $\mathcal{H}_2(t_i) \leq \mathsf{ALAP}(t_i) \quad \forall t_i \in T$ is similar to the one above. □

The bounds calculation is done in topological order so that any improvement in the value of either of these functions will be propagated to their successors or predecessors.

It is easy to see that the bounds derived by Rim and Jain [RJ94] are the same as the bounds obtained from $\mathsf{LowerBound}(T, \mathsf{ASAP}, UB - \mathsf{ALAP}, UB)$, and that the bounds computed by Langevin and Cerny [LC96] is the same as the bounds obtained from $\mathsf{LowerBound}(T, \mathcal{H}_1, UB - \mathsf{ALAP})$. Therefore, the bounds obtained from the procedure shown above are tighter than the bounds obtained by Langevin and Cerny [LC96] which in turn are tighter than those obtained by Rim and Jain [RJ94].

The algorithm shown has $O(N)$ time complexity, once $\mathcal{H}_1$ and $\mathcal{H}_2$ have been calculated; the functions $\mathcal{H}_1$ and $\mathcal{H}_2$ can be calculated in $O(N^2)$ time, and so it is advantageous to use our algorithm both for initial bound estimation and as a part of a branch-and-bound scheme. Tables 4.2, 4.3 and 4.4 show the results of running our algorithm on various benchmarks.

4.3 Bounds on Time-Constrained Scheduling (TCS)

In this section, we consider the time-constrained scheduling problem. As before, we will consider the bounds that can be obtained by relaxing all the precedence constraints. We will show how the optimal solution can be calculated in the absence of resource constraints.

4.3.1 Scheduling Independent Operations with Release Times and Deadlines (SIRD)

In this section we consider the following problem: Given a set of independent tasks T, with each task $t_i \in T$ having a release step $\mathsf{Release}(t_i)$ and a deadline $\mathsf{Deadline}(t_i)$, find the minimum number of resources required for executing all the tasks within the ranges specified. As before, we initially start by restricting all the operations to execute on the same resource class.

This problem can be modeled and solved using Integer Linear Programming. In this section, we will show that the polyhedra corresponding to this problem is (almost) integral,

Table 4.2: Bounds for the RCS problem for the *EWF* benchmark

$*_p$	$*_m$	ALU	#steps	LB
0	1	1	28	28
0	1	2	21	21
0	2	1	28	28
0	2	2	18	18
0	2	3	18	18
0	3	2	18	18
0	3	3	17	17
1	0	1	28	28
1	0	2	19	18
1	0	3	18	18
2	0	1	28	28
2	0	2	18	18
2	0	3	18	17

Table 4.3: Bounds for the RCS problem for the *ARF* benchmark

$*_p$	$*_m$	ALU	#steps	LB
0	1	1	34	34
0	1	2	34	34
0	2	1	18	18
0	2	2	18	18
0	2	3	15	15*
0	3	1	16	16
0	3	2	15	13
0	4	1	16	16
0	4	2	11	11
1	0	1	19	19
1	0	2	19	19
1	0	3	19	19
2	0	2	13	13
2	0	3	13	13
3	0	2	13	13
4	0	2	11	11

Table 4.4: Bounds for the RCS problem for the *FDCT* benchmark

$*_p$	$*_m$	ALU	#steps	LB
0	1	1	34	34
0	1	2	34	34
0	2	1	26	26
0	2	2	18	18
0	2	3	18	18
0	3	2	14	13
0	4	2	13	13
0	4	3	11	11
0	4	4	11	11
0	4	5	11	11
0	5	4	10	10
0	5	5	10	10
0	8	4	08	08
1	0	1	26	26
2	0	1	26	26
1	0	2	19	19
2	0	2	13	13
3	0	2	13	13
2	0	3	12	11
3	0	3	10	10
4	0	4	09	09
4	0	8	08	08

and that the ceiling of the optimal solution to the linear relaxation of this problem is the optimal solution to this problem; therefore, it is sufficient to solve the linear relaxation problem to find optimal integral solutions to the original problem.

4.3.2 ILP formulation for this (SIRD) problem

We use a linear programming formulation similar to the one used by Chaudhuri and Walker [CW96] and Gebotys and Elmasry [GE93]; this is a restriction of their formulation.

Each task t_i has to be scheduled at some step between $\mathsf{Release}(t_i)$ and $\mathsf{Deadline}(t_i)$, and we have to decide which step to schedule the task in. This is modeled by a set of decision variables $\{x_{i,n_1}, x_{i,n_1+1}, \ldots, x_{i,n_2}\}$, where $n_1 = \mathsf{Release}(t_i)$ and $n_2 = \mathsf{Deadline}(t_i)$. Since the task should only be executed on one of these steps, we have

$$\sum_{\mathsf{Release}(t_i) \leq j \leq \mathsf{Deadline}(t_i)} x_{i,j} = 1 \tag{4.1}$$

This is equivalent to the two constraints

$$\sum_{\mathsf{Release}(t_i) \leq j \leq \mathsf{Deadline}(t_i)} -x_{i,j} \leq -1 \tag{4.2}$$

and

$$\sum_{\mathsf{Release}(t_i) \leq j \leq \mathsf{Deadline}(t_i)} x_{i,j} \leq 1 \tag{4.3}$$

The number of resources required is the maximum of the number of resources used at any of the steps. If R is the number of resources required, then, we require that the number of resources used at each step j does not exceed R:

$$\forall \text{steps } j \quad \sum_{\{i \mid \mathsf{Release}(t_i) \leq j \leq \mathsf{Deadline}(t_i)\}} x_{i,j} \leq R. \tag{4.4}$$

Since the number of resources is a variable whose value has to be determined, we will express Equation 4.4 as

$$\left(\sum_{\{i \mid \mathsf{Release}(t_i) \leq j \leq \mathsf{Deadline}(t_i)\}} x_{i,j} \right) - R \leq 0 \tag{4.5}$$

Of course, we require that all the decision variables be either 0 or 1; also, we require that the value of R be integral.

We can represent the conditions, i.e., Equations 4.2, 4.3 and 4.5, as a matrix inequality of the form $A'\vec{x} \leq \vec{b}$, where A' is the coefficient matrix and $\vec{x}$ is the column vector corresponding to the decision variables and R. The first $|T|$ rows of A' represent the coefficients of Equation 4.2, and the next $|T|$ rows represent the coefficients of Equation 4.3. We will let $N = 2 \cdot |T|$. The remaining *Latency* rows represent the coefficients of Equation 4.5 for the various steps. Let the number of variables (=the number of decision variables+1) be M. Therefore A' is a $(N + Latency) \times M$ matrix and $\vec{x}$ a $M \times 1$ matrix(vector). Without loss of generality, we will assume that the first $M - 1$ elements of vector $\vec{x}$ represent the decision variables, and the last element represents R. Then

$$A' = \left(\begin{array}{c:c} A_1 & \vec{0} \\ A_2 & -\vec{1} \end{array} \right) = \left(\begin{array}{c:c} A & B \end{array} \right)$$

where A_1 is an $N \times (M-1)$ matrix and A_2 is a $Latency \times (M-1)$ matrix and A is the $(N + Latency) \times (M-1)$ matrix obtained by deleting the last (i.e., Mth) column from A. Let $\vec{b_1}$ be the column vector of $|T|$ 1's. Let $\vec{b_2}$ the column vector of *Latency* 0's. We denote by $\vec{b}$ the column vector

$$\begin{pmatrix} -\vec{b_1} \\ \vec{b_1} \\ \vec{b_2} \end{pmatrix}$$

We denote by X the set of decision variables.

We will now prove that A is a totally unimodular matrix. To do so, we use Lemma 2.7 (Section III.1) in [NW88], which we state below.

Lemma 4.8 *The following statements are equivalent*

1. *A is totally unimodular (TU)*

2. *For every $J \subseteq \{1, 2, \ldots, N + Latency\}$, there exists a partition J_1, J_2 of J such that*

$$\left| \sum_{j \in J_1} a_{ij} - \sum_{j \in J_2} a_{ij} \right| \leq 1 \qquad \forall i \in \{1, \ldots M\}$$

where a_{ij} represents the element in row j and column i.

Lemma 4.9 *The matrix A is totally unimodular.*

Proof. In this proof, we will assume that the first $N/2 = |T|$ rows of A are obtained from Equation 4.2, with the elements of row i being the coefficients of Equation 4.2 for task t_i. Similarly the elements of row $N/2 + i$ are the coefficients of Equation 4.2 for task t_i.

Each column of A will have exactly 3 non zero entries. There will be a -1 in one of the rows $\{1, \ldots, N/2\}$, a 1 in one of the rows $\{N/2 + 1, \ldots, N\}$ and a 1 in one of the remaining rows. This is because each decision variable x_{ij} appears once in a inequality of the form of

Equation 4.2, once in a inequality of the form Equation 4.3 and once in a inequality of the form Equation 4.5, and its coefficients are as stated above.

Let J be some arbitrary set of rows. Now, let $J_0 = J \cap \{1, \ldots, N/2\}$, $J_1 = J \cap \{N/2 + 1, \ldots, N\}$ and $J_2 = J \cap \{N+1, \ldots, N + \mathit{Latency}\}$. Notice that each column in the rows of J_1 and J_2 have at most one non zero element (which must be a 1) in any column. For every $i \in J_0$, if $i + N/2 \in J_1$, then add i to J_1, else add it to J_2. Now, $J_1 \cup J_2 = J$.

Clearly $a_{ij} \in \{-1, 0, 1\} \quad \forall j \in J_2$. Suppose that for some $j \in J_2$, $a_{i_1 j} = -1$. Then $a_{i_1 j} = 0$ for all $j \in J_1$. If $a_{i_1 j} \neq -1 \quad \forall j \in J_2$, then $\left|\sum_{j \in J_2} a_{i_1 j} - \sum_{j \in J_1} a_{i_2 j}\right| \leq 1$. Therefore, we have,

$$\left|\sum_{j \in J_2} a_{ij} - \sum_{j \in J_1} a_{ij}\right| \leq 1 \qquad \forall i \in \{1, \ldots M\}$$

□

Although we have shown that A is TU, A' need not be TU. In the next section, we show that the optimal solution to the SIRD problem can be obtained by taking the ceiling of the value returned by the LP relaxation of the ILP formulation.

4.3.3 Obtaining an optimal solution to the SIRD problem

The problem of finding the optimal (minimum) number of resources is equivalent to solving the ILP model

$$\text{(I)} \quad \text{Minimize} \quad R \quad \text{Subject to} \quad \left(A \,\vdots\, B \right) \vec{x} \leq \vec{b} \quad R, x \text{ integral } \forall x \in X$$

We denote this problem as (I). The linear relaxation of this problem (LR) is then

$$\text{(LR)} \quad \text{Minimize} \quad R \quad \text{Subject to} \quad \left(A \,\vdots\, B \right) \vec{x} \leq \vec{b}$$

Lemma 4.10 *Let r be the solution to (LR). Then $\lceil r \rceil$ is the solution to (I).*

Proof. Since the solution to (I) must be integral, we know that $\lceil r \rceil$ is a lower bound on the solution to (I). Let b_3 be the vector of *Latency* $\lceil r \rceil$'s, and $\vec{b_r}$ be the vector $\begin{pmatrix} -\vec{b_1} \\ \vec{b_1} \\ \vec{b_3} \end{pmatrix}$. Then $\lceil r \rceil$ is the solution to (I) if and only if

$$A\vec{x} \leq \vec{b_r}$$

has at least one integral solution in it. Since the matrix A is totally unimodular, $\lceil r \rceil$ is the solution to (I) iff the space has any feasible (possibly non-integral) solution. It is clear that this space must have a feasible solution as the solution returned by (LR) is certainly a feasible solution for this space. Hence $\lceil r \rceil$ is the solution to (I). □

Table 4.5: Bounds (TCS) for the *HAL* benchmark (Pipelined Multiplier)

	F-B Lower Bounds		LP Lower Bounds		Upper Bounds	
#steps	$*_p$	ALU	$*_p$	ALU	$*_p$	ALU
6	2	1	2	1	2	1
7	2	1	2	1	2	1
8	1	1	1	1	1	1
9	1	1	1	1	1	1
10	1	1	1	1	1	1
11	1	1	1	1	1	1
12	1	1	1	1	1	1
13	1	1	1	1	1	1
14	1	1	1	1	1	1

Hence the values returned by the LP solution to this problems must be no less than the values returned by those bounding techniques. Consider the case when the precedence constraints are added. Then the optimal solution must certainly be larger, because the solution space has reduced.

Let P denote matrix such that $P\vec{x} \leq \vec{q}$ represents the set of precedence constraints (see [CW94, GE93]). We define the following set of bounds:

$$\begin{aligned}
LB_0 &= \text{Bounds using [SJ93, FB73b]} \\
LB_1 &= \left\lceil \min_{A\vec{x}\leq\vec{c}} R \right\rceil \\
LB_2 &= \min_{A\vec{x}\leq\vec{c},\vec{x}\text{ integral}} R \\
LB_3 &= \left\lceil \min_{A\vec{x}\leq\vec{c},P\vec{x}\leq\vec{q}} R \right\rceil \\
Solution &= \min_{A\vec{x}\leq\vec{c},P\vec{x}\leq\vec{q},\vec{x}\text{ integral}} R
\end{aligned}$$

It is clear that $LB_1 \leq LB_2 \leq Solution$ and $LB_1 \leq LB_3$. We have shown that $LB_1 = LB_2$. It can be shown that the $LB_0 \leq LB_2$. Hence, we must have $LB_0 \leq (LB_1 = LB_2) \leq LB_3 \leq Solution$. Hence we have shown that the ceiling of the solution to the linear relaxation of the ILP formulation is tighter than the bounds obtained using [SJ93, FB73b]. The bounding technique used in [CW96] is provably tighter than LP-relaxation since they solve two LP instances and use the better of the two solutions. Therefore, it is most advantageous to use Chaudhuri and Walker's [CW96] technique for initial bound estimation. Tables 4.5-4.12 compare these techniques.

Table 4.6: Bounds (TCS) for the *EWF* benchmark (Pipelined Multiplier)

#steps	F-B Lower Bounds		LP Lower Bounds		Upper Bounds	
	$*_p$	ALU	$*_p$	ALU	$*_p$	ALU
17	2	3	2	3	2	3
18	1	2	1	2	1	3
19	1	2	1	2	1	2
20	1	2	1	2	1	2
21	1	2	1	2	1	2
22	1	2	1	2	1	2
23	1	2	1	2	1	2
24	1	2	1	2	1	2
25	1	2	1	2	1	2
26	1	2	1	2	1	2
27	1	1	1	1	1	2
28	1	1	1	1	1	1

Table 4.7: Bounds (TCS) for the *ARF* benchmark (Pipelined Multiplier)

#steps	F-B Lower Bounds		LP Lower Bounds		Upper Bounds	
	$*_p$	ALU	$*_p$	ALU	$*_p$	ALU
11	2	2	4	2	4	2
12	2	2	2	2	3	2
13	2	2	2	2	2	2
14	2	2	2	2	2	1
15	2	1	2	1	2	1
16	2	1	2	1	2	1
17	2	1	2	1	2	1
18	2	1	2	1	2	1
19	1	1	1	1	1	1
20	1	1	1	1	1	1
21	1	1	1	1	1	1
22	1	1	1	1	1	1
23	1	1	1	1	1	1
24	1	1	1	1	1	1
25	1	1	1	1	1	1
26	1	1	1	1	1	1
27	1	1	1	1	1	1
28	1	1	1	1	1	1
29	1	1	1	1	1	1
30	1	1	1	1	1	1
31	1	1	1	1	1	1
32	1	1	1	1	1	1
33	1	1	1	1	1	1
34	1	1	1	1	1	1

Table 4.8: Bounds (TCS) for the *FDCT* benchmark (Pipelined Multiplier)

#steps	F-B Lower Bounds		LP Lower Bounds		Upper Bounds	
	$*_p$	ALU	$*_p$	ALU	$*_p$	ALU
8	4	4	8	4	4	8
9	4	3	4	3	4	4
10	3	3	3	3	3	3
11	2	3	2	3	2	4
12	2	3	2	3	2	3
13	2	2	2	2	2	2
14	2	2	2	2	2	2
15	2	2	2	2	2	2
16	2	2	2	2	2	2
17	2	2	2	2	2	2
18	2	2	2	2	2	2
19	1	2	1	2		
20	1	2	1	2	1	2
21	1	2	1	2	1	2
22	1	2	1	2	1	2
23	1	2	1	2	1	2
24	1	2	1	2	1	2
25	1	2	1	2	1	2
26	1	1	1	1	1	1
27	1	1	1	1	1	1
28	1	1	1	1	1	1
29	1	1	1	1	1	1
30	1	1	1	1	1	1
31	1	1	1	1	1	1
32	1	1	1	1	1	1
33	1	1	1	1	1	1
34	1	1	1	1	1	1

Table 4.9: Bounds (TCS) for the *HAL* benchmark (Multicycle Multiplier)

#steps	F-B Lower Bounds		LP Lower Bounds		Upper Bounds	
	$*_m$	ALU	$*_m$	ALU	$*_m$	ALU
6	3	1	3	1	3	2
7	2	1	2	1	2	2
8	2	1	2	1	2	1
9	2	1	2	1	2	1
10	2	1	2	1	2	1
11	2	1	2	1	2	1
12	2	1	2	1	2	1
13	1	1	1	1	2	1
14	1	1	1	1	1	1

Table 4.10: Bounds (TCS) for the *EWF* benchmark (Multicycle Multiplier)

#steps	F-B Lower Bounds		LP Lower Bounds		Upper Bounds	
	$*_m$	ALU	$*_m$	ALU	$*_m$	ALU
17	3	3	3	3	3	2
18	2	2	2	2	2	2
19	2	2	2	2	2	2
20	2	2	2	2	2	2
21	1	2	1	2	1	2
22	1	2	1	2	1	2
23	1	2	1	2	1	2
24	1	2	1	2	1	2
25	1	2	1	2	1	2
26	1	2	1	2	1	2
27	1	1	1	1	1	2
28	1	1	1	1	1	1

Table 4.11: Bounds (TCS) for the *ARF* benchmark (Multicycle Multiplier)

#steps	F-B Lower Bounds		LP Lower Bounds		Upper Bounds	
	$*_m$	ALU	$*_m$	ALU	$*_m$	ALU
11	4	2	4	2	4	2
12	4	2	4	2	4	2
13	4	2	4	2	4	2
14	4	2	4	2	4	2
15	3	2	3	2	3	2
16	3	1	3	1	3	3
17	3	1	3	1	2	3
18	2	1	2	1	2	1
19	2	1	2	1	2	1
20	2	1	2	1	2	1
21	2	1	2	1	2	1
22	2	1	2	1	2	1
23	2	1	2	1	2	1
24	2	1	2	1	2	1
25	2	1	2	1	2	1
26	2	1	2	1	2	1
27	2	1	2	1	2	1
28	2	1	2	1	2	1
29	2	1	2	1	2	1
30	2	1	2	1	2	1
31	2	1	2	1	2	1
32	2	1	2	1	2	1
33	2	1	2	1	2	1
34	1	1	1	1	1	1

Table 4.12: Bounds (TCS) for the *FDCT* benchmark (Multicycle Multiplier)

#steps	F-B Lower Bounds		LP Lower Bounds		Upper Bounds	
	$*_m$	ALU	$*_m$	ALU	$*_m$	ALU
8	8	4	8	4	8	4
9	6	3	8	3	8	3
10	5	3	5	3	5	3
11	4	3	4	3	4	3
12	4	3	4	3	4	3
13	3	2	3	2	3	2
14	3	2	3	2	3	2
15	3	2	3	2	3	2
16	3	2	3	2	3	2
17	3	2	3	2	3	2
18	2	2	2	2	2	2
19	2	2	2	2	2	2
20	2	2	2	2	2	2
21	2	2	2	2	2	2
22	2	2	2	2	2	2
23	2	2	2	2	2	2
24	2	2	2	2	2	2
25	2	2	2	2	2	2
26	2	1	2	1	2	1
27	2	1	2	1	2	1
28	2	1	2	1	2	1
29	2	1	2	1	2	1
30	2	1	2	1	2	1
31	2	1	2	1	2	1
32	2	1	2	1	2	1
33	2	1	2	1	2	1
34	1	1	1	1	1	1

Chapter 5

Exact Scheduling Techniques

This chapter is concerned with the generation of exact schedules. Although there have been many techniques proposed to find the optimal solution to the scheduling problem, the ILP based techniques have been the most successful so far. In this chapter, some techniques which can be applied to the ILP solution process in order to improve its efficiency will be presented. A non-ILP based technique will also be presented which is shown to be superior to ILP based methods in terms of execution time required. This algorithm, a branch and bound algorithm, uses the bounding techniques presented in Chapters 3 and 4 to quickly determine the optimal solution.

5.1 Improving the efficiency of ILP

The ILP model for the scheduling problem has already been presented in Chapter 2. However simply using a generic ILP solver to solve the problem is perhaps not the most efficient way to solve the problem. We can build a better branch and bound algorithm by exploiting our knowledge of the "original" problem (as opposed to the ILP problem). This is especially true in cases when most of the variables are 0-1 variables and represent decision variables. In such a problem, if the solver branches on a decision variable y_{ij}, then it creates two branches, $y_{ij} = 0$ and $y_{ij} = 1$. It should be noted that setting the decision variable to 0 or to 1 has an interpretation in the original problem. It is possible that this decision has other ramifications, and so by using a problem specific ILP solver, we can improve the efficiency of the ILP solution process by exploiting this additional information. Another area in which the knowledge of the problem can be exploited is in the improvement of the lower bound and the selection of the next branch. These will be described in detail in the sections that follow. For the rest of the chapter, we will assume that the ILP formulation is

$$Z_{ILP} = \min \{cx : Ax \leq b, \quad x \in \mathcal{Z}_+^m\}$$

Since we are considering branch and bound techniques, we will be partitioning the polyhedron described by $Ax \leq b$, i.e., (A, b), into disjoint polyhedra $(A_1, b_1), (A_2, b_2), \ldots, (A_n, b_n)$ whose union contain all the elements of $integer(A, b)$. We will use $ILP\left((A, b), c\right)$ to denote the vector

$$ILP\left(\left(A,b\right),c\right)=\{x\in integer(A,b):cx\leq cx'\quad\forall x'\in integer(A,b)\}$$

and $LR\left(\left(A,b\right),c\right)$ to denote

$$LR\left(\left(A,b\right),c\right)=\{x:Ax\leq b\text{ and }cx\leq cx'\quad\forall Ax'\leq b\}.$$

We denote by $(A,b)|_{y_{i_1j_1}=k_1,y_{i_2j_2}=k_2,\ldots,y_{i_lj_l}=k_l}$ the polyhedron

$$\{x:Ax\leq b\text{ and }y_{i_1j_1}=k_1,y_{i_2j_2}=k_2,\ldots,y_{i_lj_l}=k_l\}$$

Note that

$$integer(A,b)\subseteq(A,b)|_{y_{ij}=1}\cup(A,b)|_{y_{ij}=0}\qquad\forall\text{ decision variables }y_{ij}$$

and so

$$ILP\left(\left(A,b\right),c\right)=\min\left\{ILP\left(\left(A,b\right)|_{y_{i,j}=1},c\right),ILP\left(\left(A,b\right)|_{y_{i,j}=1},c\right)\right\}$$

5.1.1 Better Next Branch Selection

Clearly if $LR\left(\left(A,b\right),c\right)$ is an integral vector, then we are done. Suppose that this is not the case. Then there is at least one variable in $LR\left(\left(A,b\right),c\right)$ which is non-integral. Normally, many of the variables are non-integral, and so the solver picks one of the variables (say y_{ij}) to branch on. We will assume that the variable picked is a decision variable (i.e. takes on only the values 0 or 1) and so

$$integer\ (A,b)\subseteq(A,b)\,|_{y_{ij}=0}\cup(A,b)\,|_{y_{ij}=0}.$$

Let $(A_1,b_1)=(A,b)\,|_{y_{ij}=0}$ and $(A_2,b_2)=(A,b)\,|_{y_{ij}=1}$. The solver now has to decide whether it will attempt to find the optimal solution in (A_1,b_1) first or in (A_2,b_2) first. So, the solver has to make two decisions here. First it has to pick the variable to branch on. Then it has to decide which one of the smaller polyhedra resulting from this branch to explore first. A particularly naive solver might just make random choices for both these decisions. More sophisticated solvers might attempt to pick a variable to branch on based on the values of the variables, or branch on a set of variables simultaneously. A better branching scheme can be derived by exploiting the fact that the variables $\{y_{ij}:j\in\mathsf{TimeFrame}(t_i)\}$ are the set of variables deciding the step to which t_i is assigned. Notice that only one of these variables can be set to 1 at a time. Therefore,

$$integer(A,b)\subseteq\bigcup_{j\in\mathsf{TimeFrame}(t_i)}(A,b)|_{y_{ij}=1}.$$

Therefore, a good branching scheme would be to pick an operation t_i to branch on, and then branch on this set of variables.

We can use many heuristics to pick the next operation to branch on. If we look at the value of the variable y_{ij} in the vector $LR((A,b),c)$ as the probability of scheduling t_i in time step j, then we might decide to pick the operation t_i corresponding to the variable y_{ij} with the maximum value (probability). Then, this scheme would also result in an order in which the branches would be considered. Another scheme which yielded better results involves picking the operation t_i which has not been scheduled or branched on and all of whose predecessors have been scheduled. Once the operation is picked, the branches can be explored in order of the values of y_{ij}. This scheme considerably reduced the number of times the the linear programming solver had to be called.

5.1.2 Addition of Valid Inequalities

As we already mentioned, branching on a node (set of nodes) has an interpretation in the original problem. We use this information to generate additional valid inequalities, i.e., inequalities which will always be satisfied by any integer solution, but need not be satisfied by non-integer solutions. Suppose that as a result of a branch, a variable $y_{i_1 j_1}$ is set to 1. What this means is that operation t_{i_1} has been scheduled to step j_1. Clearly this means that all none of the operations $t_{i_2} \in T_{succ}(t_{i_1})$ can be scheduled to a time step before $\Gamma(t_{i_1}, t_{i_2}) + j_1$[1]. Therefore, we can introduce another inequality

$$\sum_{\substack{j_2 \in \mathsf{TimeFrame}(t_{i_2}) \\ j_2 < \Gamma(t_{i_1}, t_{i_2}) + j_1}} y_{i_2 j_2} = 0.$$

for every operation $t_{i_2} \in T_{succ}(t_{i_1})$. We can derive this inequality based on our knowledge of the "original" problem. In practice, we add up all the inequalities of the form of the inequality shown, and create a single inequality

$$\sum_{\substack{j_2 \in \mathsf{TimeFrame}(t_{i_2}) \\ j_2 < \Gamma(t_{i_1}, t_{i_2}) + j_1 \\ t_{i_2} \in T_{succ}(t_{i_1})}} y_{i_2 j_2} = 0$$

It should be noted that while in general replacing many inequalities with their sum results in a weakening of the inequalities, in this case there is no weakening. Adding such an inequality tightens the formulation, and results in faster solution times. This technique of adding cutting planes (inequalities) in a branch and bound setting is called branch and cut, and has recently been used to solve some extremely large ILP formulations.

[1] $\Gamma(t_i, t_j)$ represents the length of the longest path from t_i to t_j

5.1.3 Better Bounding

Tightening the bounding procedure can dramatically reduce the time taken for computing the optimal solution. In most of the cases, the ILP solver finds an optimal solution early on, but spends most of the time just confirming that the solution found is indeed the best possible. Therefore, tightening the bounds produced can reduce the problem solving time. Knowledge of the objective function can sometimes be used to improve the actual lower bound Z_{LR} generated by linear relaxation. For the ILP formulation of the RCS problem, we know that the latency will always be integral. Therefore, the objective function must always be integral, and hence, we can use the ceiling of the linear relaxation $\lceil Z_{LR} \rceil$ as a tighter lower bound. The same can be done for the TCS problem. If we are optimizing for area, and if the ratios of the areas of the resources can be expressed (or at least reasonably approximated) as an integer ratio (and this is almost always the case), we can do the same. Doing this is particularly important for the TCS problem because in this case $\lceil Z_{LR} \rceil$ is very often the tightest bound that can be obtained (see Chapter 4). Therefore, once the optimal solution has been found, just a few iterations of the linear program solver will confirm the optimality of the solution. A big advantage of this scheme is that the increase in computation cost (calculating the ceiling of a number compared to solving a LP formulation) is almost negligible. For the RCS problem, we can improve the lower bound by using an additional lower bounding scheme (like the ones discussed in Chapter 3). Since the computational complexity of the bounding procedures presented is $O(|T|)$, calling the routine is not likely to add significantly to the complexity of the algorithm.

5.1.4 Better Tests for Optimality

Another improvement would be to run a heuristic each time the solver is forced to branch, to see if the heuristic can come up with a schedule, whose objective function is the same as the solution to the linear relaxation of the problem at hand. The reason why this might prove useful is that there may be many solutions with the same objective function value. It is quite possible, therefore, for the (linear programming) solver to return a non-integral solution, even though there is actually an integral solution whose objective criteria is the same. On several occasions, running a simple heuristic confirms this, eliminating the need for further search. It is, of course crucial that the heuristic runs fast.

5.1.5 Reduction of the Search Space

The ILP model presented in Chapter 2 assumed that the $\mathsf{TimeFrames}$ of all the variables had been calculated. It was then mentioned that they could be calculated as $\mathsf{TimeFrame}(t_i) = \{\mathsf{ASAP}(UB, t_i), \ldots, \mathsf{ALAP}(UB, t_i)\}$. Being able to reduce the $\mathsf{TimeFrames}$ of the operations in T will improve the ILP model by reducing the size of the search space. The $\mathcal{H}_1$ and the $\mathcal{H}_2$ schedules have been shown to be better approximations of ASAP_{ideal} and ALAP_{ideal}. Hence using these to calculate the $\mathsf{TimeFrames}$ will lead to a ILP model with fewer variables. For the RCS problem, a further reduction may be obtained. We will let ϕ be the set of schedules which respect dependence and resource constraints. Now, let ξ be some arbitrary element of ϕ (possibly generated using a heuristic). As mentioned in Section 3.3, in order to have create a meaningful and useful ASAP schedule, we optimize over

$$\Phi = \{\theta \in \phi : \mathsf{Latency}(\theta) \leq \mathsf{Latency}(\xi)\}$$

this can be justified because a schedule with the least latency in ϕ is also a schedule with the least latency in Φ. However, suppose we set

$$\Phi' = \{\theta \in \phi : \mathsf{Latency}(\theta) < \mathsf{Latency}(\xi)\}.$$

Then the optimal schedule in Φ' is an optimal schedule in ϕ if and only if Φ' is non-empty. The ILP solver declare the ILP model corresponding to minimizing over Φ' as infeasible if Φ' has no integral vector in it. Suppose that the polyhedron representing the (schedule) search spaces Φ and Φ' are P_Φ and $P_{\Phi'}$ respectively. If Φ' is non-empty, and we assume that the ILP solver returns ∞ if the model is infeasible, then

$$ILP(\Phi, c) = \min\{\mathsf{Latency}(\xi), ILP(\Phi', c)\}.$$

The first point to note is that the dimension of P_Φ is less than the dimension $P_{\Phi'}$ because there are fewer variables required (exactly $|T|$ fewer) to model the polyhedron $P_{\Phi'}$. Therefore, it is likely that attempting a minimization over $P_{\Phi'}$ is likely to be quicker than a minimization over P_Φ both because solving each linear relaxation takes less time, and because the number of branches will be reduced. Further, if there Φ' is non-empty, then because the TimeFrames of the operations are more accurate, the optimal solution is likely to be found quicker.

Another point to be noted is that this same technique can actually be applied every time a solution which is better than the previous best solution is found. Therefore, finding a better solution will lead to both better bounding and a reduction in the number of branches that have yet to be explored.

5.2 An alternative to Integer Linear Programming

We have shown how and ILP formulation can solve the scheduling problem, and how the efficiency of such an approach can be improved. In this section, we will be exploring another branch and bound scheme. As we have shown in Chapter 4, there are a number of schemes to produce lower bounds for this problem [NR97, LC96, RJ94] . We have also demonstrated that some of these techniques can actually generate better (tighter) bounds than linear relaxation can. However, if another bounding scheme is to be used, then the criteria of optimality (i.e., if the lower bound is achievable) can be checked by also calculating an upper bound. If the values of the upper and lower bounds match, then we are guaranteed optimality. Therefore, we can embed an arbitrary lower bounding procedure in a branch and bound strategy as long as we also have a way to verify if there is an (actual) solution for which the objective function has the same value as the bound.

Linear programming (LP) is a time consuming procedure when the number of variables is large. In ILP problems involving a large number of variables, it is possible (even likely) that

Procedure $\mathsf{Enumerate}(i)$ /* Schedule nodes $\{t_i, t_{i+1}, \ldots t_{|T|}\}$ */
if $(i = N+1)$ /* Scheduled all the elements ? */
  if $(\mathsf{Feasible}(\mathcal{S}))$
    $best \leftarrow \min(best, \mathsf{Latency}(\mathcal{S}))$;
  endif
else
  for $j \leftarrow \mathsf{ASAP}(t_i)$ **to** $\mathsf{ALAP}(t_i)$ **do**
    $\mathcal{S}(t_i) \leftarrow j$; /* Schedule this node ... */
    $\mathsf{Enumerate}(i+1)$; /* and then schedule the rest */
  enddo
endif

Figure 5.1: Naive Complete Enumeration (Algorithm 1)

the LP routine is called a number of times. It is therefore likely that if we used another, faster technique than linear relaxation to bound the search space, we can (possibly) produce a solution much faster. The speed increase will be more pronounced if the bounding procedure used actually generates better (tighter) bounds, thereby reducing the search space that has to be completely explored. In this section, we present such a branch and bound algorithm for the RCS problem. It should be noted that this technique is not specific to this problem, and can also be used for any other problem.

An extremely naïve way of finding an optimal schedule is to enumerate every possible schedule and then pick the optimal schedule.

Let $\mathcal{F}$ be set of schedules $\{\mathcal{S} : T \rightarrow \mathcal{Z}^+ | \mathcal{S}(t_i) \in \mathsf{TimeFrame}(t_i)\ \forall t_i \in T\}$. Note that the elements in $\mathcal{F}$ need not be feasible schedules. Let ϕ be the set of feasible schedules (i.e., valid schedules that also respect the dependency constraints). Let $|T| = N$, and $\{t_1, t_2, \ldots, t_N\} = T$. Without loss of generality, let us assume that these elements are topologically ordered. So $\langle t_i, t_j \rangle \in E \Rightarrow i < j$.

Given this ordering, we can associate with each element $\mathcal{S} \in \mathcal{F}$ a N-tuple $\langle \mathcal{S}(t_1), \mathcal{S}(t_2), \mathcal{S}(t_3), \ldots, \mathcal{S}(t_N) \rangle$. It is very easy to enumerate all elements in $\mathcal{F}$. Given any element of $\mathcal{F}$, it is easy to check if it is indeed a feasible schedule. So, we can use the algorithm in Figure 5.1 to enumerate all schedules.

This algorithm is extremely inefficient. Part of its inefficiency stems from the fact that we are enumerating the elements of $\mathcal{F}$ instead of the elements of ϕ and $|\ \mathcal{F}\ | \gg |\ \phi\ |$. A schedule $\mathcal{S} \in \mathcal{F}$ is in ϕ unless at least one of the following occur:

- $\mathcal{S}$ violates one or more dependencies.
- $\mathcal{S}$ violates one or more resource constraints.

Suppose that there is a dependency violation in $\mathcal{S}$. Let us assume that the dependency $\langle t_i, t_j \rangle$ is violated. That means $\mathcal{S}(t_j) \leq \mathcal{S}(t_i)$. It is clear that any schedule with $\mathcal{S}(t_j) \leq \mathcal{S}(t_i)$ is not a feasible schedule. Since the operations are topologically ordered, we will be scheduling t_j (or assigning a value to $\mathcal{S}(t_j)$) only after we schedule t_i. If the check for the feasibility of

```
Procedure Enumerate(i)                          /* Schedule nodes {t_i, t_{i+1}, ... t_|T|} */
if (i = N + 1)                                  /* Scheduled all the elements ? */
   if (Feasible(S))
      best ← min(best, Latency(S));
   endif
else
   for j ← ASAP(t_i) to ALAP(t_i) do
      SaveASAPValues;
      if (ResourceAvailable(j, Resource(t_i)))
         S(t_i) ← j;                            /* Schedule this node ... */
         UpdateASAP(i, j);
         Enumerate(i + 1);                      /* and then schedule the rest */
      endif
      RestoreASAPValues;
   enddo
endif

Procedure UpdateASAP(i, step)
for j ← i + 1 to N do
   if (t_j ∈ T_succ(t_i))
      ASAP(t_j) ← max(ASAP(t_j), step + Γ(t_i, t_j));
   endif
enddo
```

Figure 5.2: Complete Enumeration (Algorithm 2)

the schedule were to be made when we scheduled t_j, it would fail, and so there is no need to continue until all the remaining elements are scheduled. Another way of looking at this is that once operation t_i has been scheduled to time step $\mathcal{S}(t_i)$, operation t_j cannot begin executing until at least time step $\mathcal{S}(t_i) + \Gamma(t_i, t_j)$. So setting $\mathcal{S}(t_i)$ would imply that $\mathcal{S}(t_j) \geq \mathcal{S}(t_i) + \Gamma(t_i, t_j)$. This is in effect, altering the value of $\mathsf{ASAP}(t_j)$. If after each element is scheduled, the ASAP values are updated as above, then there will never be a dependency violation. So, the check for dependency violations can be eliminated.

Similarly, we can detect resource constraint violations early on by tracking the resource utilization of all the resources. The algorithm shown in Figure 5.2 is more efficient that the one in Figure 5.1, because the checks for the feasibility of the partial schedule are performed at each each recursive call, and any violations that can lead to infeasible schedules are detected early on.

While the algorithm shown in Figure 5.2 is an improvement over the one in Figure 5.1, it still is too inefficient to be of any practical use. Although this algorithm enumerates only the elements in ϕ, the number of such elements is (in the worst case) exponential in the number of nodes. To bring about an improvement in the performance of the algorithm, we must reduce the number of feasible schedules that are actually examined. By doing this, we are in effect, pruning the search space. In the next section, we show how this can be done using the upper and lower bounding routines.

5.2.1 Improving algorithm efficiency by more effective pruning

Consider a k-tuple $\langle X_1, X_2, \ldots, X_k \rangle$, $1 \leq k \leq |T|$. We call this k-tuple a partial schedule if all the following three conditions hold:

1. $X_i \in \mathsf{TimeFrame}(t_i)\ \forall i \in \{1, 2, \ldots, k\}$;
2. $\langle t_i, t_j \rangle \in E \ \ \Rightarrow \ \ X_i < X_j \qquad \forall i, j \in \{1, 2, \ldots, k\}$; (and)
3. $|\{t_i \in T|\, X_i = n \text{ and } \mathsf{Resource}(t_i) = \ell\}| \leq M_\ell$
$\forall n \in Z, \ell \in \{1, 2, \ldots, R\}, i \in \{1, 2, \ldots, k\}$.

We denote by $\langle\rangle$ the partial schedule corresponding to $k = 0$. For a partial schedule, decisions have been made about scheduling the first k nodes (in the ordering specified). Let Ψ be the set of all partial schedules. Clearly $\phi \subset \Psi$. Also notice that the partial schedule $\langle\rangle$ is an element of Ψ for which no scheduling decisions have been made. For any element ψ of Ψ, $|\ \psi\ | = k \Leftrightarrow \psi$ is a k-tuple.

Construct a tree as follows. The leaves are the elements of ϕ. The interior nodes are formed of the elements of $\Phi - \phi$. An element of $\psi_1 = \langle X_1, X_2, \ldots, X_{k_1} \rangle \in \Psi - \phi$ is the parent of an element of $\psi_2 = \langle Y_1, Y_2, \ldots, Y_{k_2} \rangle \in \Psi$ if

- $k_1 = |\ \psi_1\ | = |\ \psi_2\ | - 1 = k_2 - 1$ (and)
- $X_i = Y_i \quad \forall i \in \{1, 2, \ldots, k_1\}$.

So any ancestor of an element of the tree represents a partial schedule, for which fewer decisions have been made, and all the decisions that have been made for the ancestor are identical to the decisions that have been made for the descendant. Therefore, the root of the tree is $\langle\rangle$, and the leaves correspond to feasible schedules. What we require is the leaf of the tree corresponding to the least latency. It should be clear that the algorithm listed above is essentially a tree searching algorithm (or alternatively a depth-first traversal of the graph corresponding to this tree). Notice that this tree is very similar to the enumeration tree for branch and bound based ILP discussed in [NW88].

We will now show how the efficiency of this algorithm can be improved by pruning parts of the tree.

Suppose we had a lower (upper) bound estimating function, which when given a graph G, a set of resource constraints, and a partial schedule ψ, will return a lower (upper) bound on the latency of any feasible schedule (leaf of the tree) which is a descendant of ψ in the tree G_T corresponding to G. Let $\mathsf{Lower}(\psi)$ and $\mathsf{Upper}(\psi)$ represent the lower and upper bounding functions. We will also require that the upper bounding function return an upper bound that is achievable, i.e., there should actually be a schedule that is a descendant of ψ with latency $\mathsf{Upper}(\psi)$. Let $\mathcal{U}(\psi)$ be one such schedule.

Suppose, for some partial schedule ψ, $\mathsf{Lower}(\psi) = \mathsf{Upper}(\psi)$. Then, $\mathcal{U}(\psi)$ is a schedule with the least latency in the subtree rooted at ψ. Therefore, we can obtain the optimal schedule in this sub-tree without having to exhaustively explore it. Thus, computing lower and upper bounds for every interior node of the tree results in a need to explore a sub-tree if and only if the lower and upper bounds differ. We use this concept to derive the refined algorithm listed in Figure 5.3. We will now illustrate this algorithm with an example. Consider the DFG shown in Figure 5.4. We will assume that the operations shown (1-11) are to be executed on two classes of resources. Operations 1, 2, 3, 4, 6 and 8 are to be executed on resource class A, and the remaining operation are to be executed on resource class B. We will assume that there are 2 instances of resource class A and 1 instance of resource class B. Further, for simplicity, we will assume that delay of all operations is 1 time step.

A portion of the enumeration tree is shown in Figure 5.5. For the purpose of our example, we will use very simplistic bounds estimators. The values in parenthesis below each node (partial schedule) represent the values returned by our bounds estimators for the partial schedule. So, the optimal schedule has latency at least 4 and at most 5, and all schedules in which the first operation is scheduled to time step 2 (represented by the subtree rooted at the node (partial schedule) $\langle 2 \rangle$) must have latency at least 5, and there is at least one such schedule with latency ≤ 6. The encircled numbers besides the node indicate the order of exploration of the tree. Therefore, node $\langle \rangle$ is the first node to be explored, followed by node $\langle 1 \rangle$, $\langle 1, 1 \rangle$ and $\langle 1, 1, 2 \rangle$.

Initially the algorithm upper and lower bounds all schedules. Since the values of the upper (5) and the lower (4) bound do not match, we must further explore by subdividing the search space into sub spaces (or sub trees) and searching each of those spaces(trees).

When the algorithm reaches node (partial schedule) $\langle 1, 1, 2 \rangle$, the values of the upper and lower bound are equal and so it does not have to further enumerate the sub tree rooted at this node.

After this when it attempts to bound the latencies of the optimal schedule in the subtrees rooted at nodes $\langle 1, 1, 3 \rangle$ and $\langle 1, 1, 4 \rangle$, since the lower bounds of both these trees is not less than the latency of the best schedule found so far (5), these sub trees cannot possibly contain a schedule with latency less than the optimal schedule found so far. Therefore, these subtrees can be pruned away, and do not have to be explored.

The nodes marked in dotted lines represent those nodes whose lower and upper bound information (coupled with the latency of the best schedule so far) allow us to determine that the subtree rooted at this node cannot possibly contain a solution better than one we have already found. Notice that the node $\langle 1, 1, 2 \rangle$ comes under this category because it cannot possibly contain a schedule with latency less than 5, and a schedule with latency equal to 5 (returned by the function $\mathcal{U}(\langle 1, 1, 2 \rangle)$) exists. The nodes marked in solid lines, however represent the nodes whose bounds do not provide enough information to determine if this subtree can possibly contain an optimal schedule, and therefore must be explored further.

5.3 Lower Bound estimation

The values of ASAP and ALAP that we use in our algorithm are obtained by running the algorithm in Chapter 4. We also use this algorithm as the lower-bound estimating routine we require.

In the previous section, we required a topological ordering of the operation of the graph $G = (T, E)$. Calculation of the ASAP (ALAP) values as mentioned above ensures that

```
l ← Lower(⟨⟩);
u ← Upper(⟨⟩);
if (l = u)
    return U(⟨⟩);
endif
best_latency ← u ;
best_schedule ← U(⟨⟩);
ENUMERATE(⟨⟩);
return best_schedule;

Procedure Enumerate(⟨X_1, X_2, ..., X_i⟩)
if (i = N + 1)                                   /* Scheduled all the elements ? */
    if (best_latency > Latency(⟨X_1, X_2, ..., X_N⟩))
        best_schedule ← ⟨X_1, X_2, ..., X_N⟩);
        best_latency ← Latency(⟨X_1, X_2, ..., X_N⟩);
    endif
else
    for j ← ASAP(t_i) to ALAP(t_i) do
        SaveASAPValues;
        if (ResourceAvailable(j, Resource(t_i)))
            l ← Lower(⟨X_1, X_2, ..., X_i⟩);
            u ← Upper(⟨X_1, X_2, ..., X_i⟩);
            if (u < best_latency)                    /* Found a better schedule */
                best_latency ← u;
                best_schedule ← U(⟨X_1, X_2, ..., X_i⟩);
            endif
            if (l < best_latency) /* Can this sub-tree contain a better schedule ? */
                S(t_i) ← j;                          /* Schedule this node ... */
                UpdateASAP(i, j);
                Enumerate(i + 1);                    /* and then schedule the rest */
            endif
        endif
        RestoreASAPValues;
    enddo
endif
```

Figure 5.3: Refined Complete Enumeration

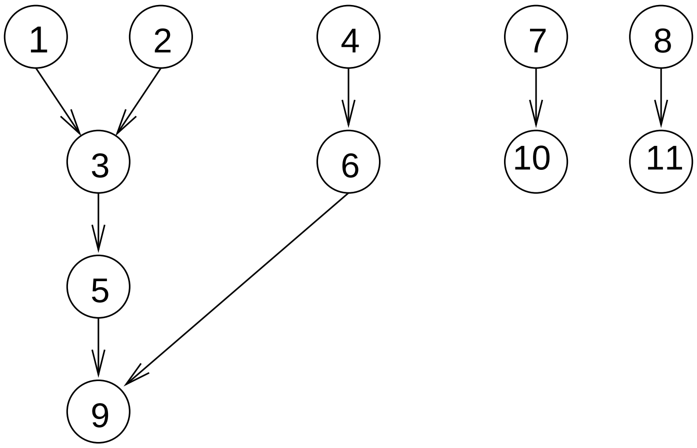

Figure 5.4: Example data flow graph

$\mathsf{ASAP}(t_i) < \mathsf{ASAP}(t_j) \forall t_i \in T_{pred}(t_j)$. Therefore, sorting the operations based on their ASAP (ALAP) values ensures that the operations are topologically ordered. The topological ordering that we use is the one obtained by sorting the operations based on their ALAP value.

Besides just lower bounding the latency of all schedules of the graph G, we also need to lower bound the latency of just those schedules for which the first m operations (in order of increasing ALAP values) are assigned to specific steps; in other words, we require a lower bound on the latency of the set of schedules which are descendants of a partial schedule, i.e., a lower bound on

$$\min_{\substack{\mathcal{S}\in\phi \\ \mathcal{S}(x_i)=s_i, 1\leq i\leq m}} latency(\mathcal{S}).$$

This can be done by a slight modification of the algorithm shown in Figure 4.3 The modified algorithm is shown in Figure 5.6.

5.4 Upper bound estimation

The latency of any valid schedule is a valid upper bound on the latency of the optimal schedule. Therefore, to obtain tight estimates on the upper bound of the latency, we just employ a good heuristic to compute an valid schedule, and return the the latency of that schedule as the upper-bound required. Notice that this fits in with our requirement that the upper-bound returned must be feasible. There are a number of good heuristics available. In our implementation, we used a list scheduler, which uses the ALAP values of the operations as a priority function. The algorithm is shown in Figure 5.7.

We choose a list scheduler over other algorithms such as force directed Scheduling [PK89] because of its speed. This procedure will be called several times and so speed is important.

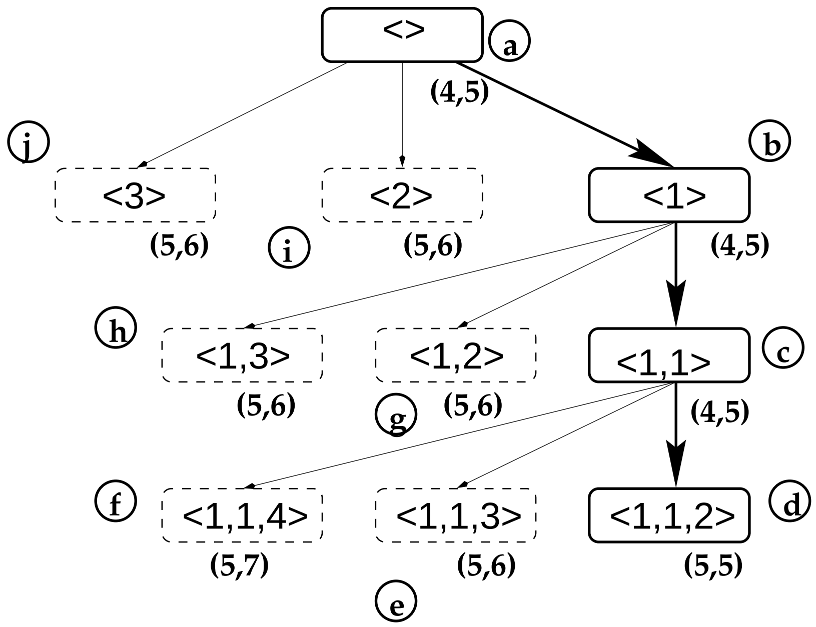

Figure 5.5: Search tree for the example data flow graph

5.5 Discussion of the Algorithm

As mentioned before, this algorithm is, in a sense, very similar to a branch-and-bound algorithm that uses linear relaxation (or BABLR). In fact, linear relaxation of the ILP model could replace the lower bounding routine from [LC96]. Using linear programming as a bounding routine has the advantage that if all the variables of the optimal solution are integral then the optimal solution represents a valid schedule, and so there is no need to upper bound the search space. Gebotys and Elmasry [GE93] and Chaudhuri and Walker [CW94] have shown that a tight ILP formulation will cause the linear relaxation to yield a integral solution in a number of cases. However, we have noticed that in many cases this does not occur, and a linear relaxation based branch-and-bound solver may take inordinately long to solve non-trivial problems.

The better performance of our branch-and-bound strategy can be attributed to two main factors:

1. better bounding schemes;

2. better exploration strategy.

As we have shown, it is possible for ILP based techniques to exploit (2). Both the lower and the upper bounding algorithms that we have used are problem specific. Heuristics in general, are designed to make "good decisions", and are better able to use problem specific information. Further, the performance of the algorithm specified above is dependent on the quality of the heuristic. If, for example, in 90% of the cases, the upper-bound and the lower-bounds returned are exact, then in 90% of the cases, no branching is required, and

procedure PSLBE$((\langle s_1, s_2, \ldots s_m \rangle)$
$latency\_estimate \leftarrow \min(s_1, s_2, \ldots s_m)$;
for $i \leftarrow 1$ **to** $|T|$ **do**
  $t \leftarrow i^{th}$ operation in T in order of increasing ALAP value.
  $k \leftarrow \mathsf{ASAP}(t)$;
  while ($\mathsf{AvailableResources}(k, \mathsf{Resource}(t)) = 0$) **do**
    $k \leftarrow k + 1$;
  enddo
  Decrement $\mathsf{AvailableResources}(k, \mathsf{Resource}(t))$;
  $latency\_estimate \leftarrow \max(latency\_estimate, k + \mathsf{CP}(T_{succ}(t)))$;
enddo

Figure 5.6: Partial Schedule Lower Bound Estimation

procedure PSUBE$((\langle s_1, s_2, \ldots s_m \rangle)$

for $i \leftarrow 1$ **to** $|T|$ **do**
  $x \leftarrow i^{th}$ operation in T in order of increasing ALAP value.
  $k \leftarrow \mathsf{ASAP}(t)$;
  while (there is a member of $T_{pred}(t)$ scheduled on of after step k) **do**
    $k \leftarrow k + 1$;
  enddo
  while ($\mathsf{AvailableResources}(k, \mathsf{Resource}(t)) = 0$) **do**
    $k \leftarrow k + 1$;
  enddo
  Decrement $\mathsf{AvailableResources}(k, \mathsf{Resource}(t))$;
  $latency\_estimate \leftarrow \max(latency\_estimate, k + \mathsf{CP}(T_{succ}(t)))$;
enddo

Figure 5.7: Partial Schedule Upper Bound Estimation

Table 5.1: Optimal Solutions for the TCS Problem for the *HAL* benchmark(Multicycle Multipliers)

#steps	$*_m$	ALU	Area	Refined ILP(F)	Refined ILP(R)	Vanilla ILP
13	1	1	7	0.08	0.22	0.43
12	2	1	12	0.08	0.06	317.91
11	2	1	12	0.05	0.04	76.01
10	2	1	12	0.07	0.03	28.26
09	2	1	12	0.07	0.06	3.76
08	2	1	12	0.01	0.01	0.09
07	2	2	14	0.01	0.01	0.03
06	3	2	19	0.01	0.01	0.02

the algorithm terminates extremely quickly. Therefore, improvements in heuristic quality will also translate directly to an improvement in the performance of this algorithm. Given the number of good heuristics available, this could prove to be very advantageous.

It is possible that custom BABLR algorithms will be able to yield better performance that the algorithm presented above. However, we have noticed that in many cases, the bounds produced by the lower bounding algorithm that we have used is superior those produced by the linear relaxation of the ILP problem. Further, this lower bounding algorithm is superior to the linear programming in terms of both complexity and in terms of actual execution time.

While we are using upper and lower bounding techniques from [LC96] in our implementation, the algorithm is not specific to those techniques. We could just as easily use other lower bounding techniques such as those in [OKD94, FB73b, KP87, RK92, HGC93, HC94] or other upper bounding algorithms (heuristics) [De 94, GDWL92, PK89, Pau91, RCJ72, Hu61].

5.6 Experimental Results

In this section, the effect of the algorithms on various benchmarks is presented. Since some of these benchmarks are very large, it was not possible to obtain solutions to all instances of the TCS and RCS problems for all the benchmarks. Cases for which the solution was not obtained within 1 hour is indicated with a (-). For all our examples, we assumed that the expression for the area was $5 \cdot \#multipliers + 2 \cdot \#adders$. For the problems to which solutions could not be obtained within 1 hour of CPU time, the latency of the best schedule obtainable in 1 hour is listed in the column UB, and the lower bound is specified in the column LB. For the Refined ILP procedure, we list two set of results, one obtained by runnnin the procedure on the graph, and the other obtained by running the procedure on the reverse graph.

Table 5.2: Optimal Solutions for the TCS Problem for the *HAL* benchmark(Pipelined Multipliers)

#steps	$*_p$	ALU	Area	Refined ILP(F)	Refined ILP(R)	Vanilla ILP
08	1	1	7	0.02	0.03	0.11
07	2	1	12	0.02	0.02	1.33
06	2	1	12	0.01	0.01	0.33

Table 5.3: Optimal Solutions for the TCS Problem for the *EWF* benchmark(Multicycle Multipliers)

#steps	$*_m$	ALU	Area	Refined ILP(F)	Refined ILP(R)	Vanilla ILP
28	1	1	7	19.9	29.03	-
27	1	2	9	-	-	-
26	1	2	9	1.97	2.17	-
25	1	2	9	1.01	1.22	-
24	1	2	9	0.87	0.76	-
23	1	2	9	4.12	0.08	-
22	1	2	9	2.44	0.26	-
21	1	2	9	0.81	0.86	-
20	2	2	14	0.17	0.12	-
19	2	2	14	0.29	0.23	-
18	2	2	14	0.11	0.02	0.05
17	3	3	21	0.01	0.01	0.02

Table 5.4: Optimal Solutions for the TCS Problem for the *EWF* benchmark(Pipelined Multipliers)

#steps	$*_p$	ALU	Area	Refined ILP(F)	Refined ILP(R)	Vanilla ILP
28	1	1	7	-	1014.99	35.11
27	-	-	-	-	-	8.92
26	1	2	9	2.53	16.64	-
25	1	2	9	1.15	1.14	-
24	1	2	9	0.92	1.19	-
23	1	2	9	0.08	4.53	-
22	1	2	9	0.79	0.48	-
21	1	2	9	0.50	1.68	-
20	1	2	9	0.35	0.83	-
19	1	2	9	0.38	0.08	730.77
18	1	3	11	0.03	0.12	0.06
17	2	3	16	0.01	0.01	0.03

Table 5.5: Optimal Solutions for the TCS Problem for the *ARF* benchmark(Multicycle Multipliers)

#steps	$*_m$	ALU	Area	Refined ILP(F)	Refined ILP(R)	Vanilla ILP
34	1	1	7	87.36	-	-
33	2	1	12	27.49	8.08	-
32	2	1	12	22.96	11.91	-
31	2	1	12	9.53	8.32	-
30	2	1	12	19.19	17.48	-
29	2	1	12	80.99	5.49	-
28	2	1	12	-	5.21	-
27	2	1	12	126.45	4.29	-
26	2	1	12	47.91	4.51	-
25	2	1	12	67.13	13.1	-
24	2	1	12	41.11	11.71	-
23	2	1	12	65.66	14.37	-
22	2	1	12	13.37	5.26	-
21	2	1	12	24.25	8.45	-
20	2	1	12	481.31	14.93	-
19	2	1	12	53.54	114.09	3.53
18	2	1	12	4.68	3.07	2.46
17	2	3	16	-	101.51	-
16	2	3	16	130.49	844.43	-
15	3	2	19	72.28	92.40	-
14	4	2	24	0.32	0.94	-
13	4	2	24	0.66	3.52	-
12	4	2	24	0.23	0.17	0.14
11	4	2	24	0.03	0.03	0.06

Table 5.6: Optimal Solutions for the TCS Problem for the *ARF* benchmark(Pipelined)

#steps	$*_p$	ALU	Area	Refined ILP(F)	Refined ILP(R)	Vanilla ILP
19	1	1	7	1872.11	1.77	4.65
18	2	1	12	6.53	7.14	-
17	2	1	12	1094.49	4.00	-
16	2	1	12	73.00	68.25	-
15	2	1	12	104.07	377.87	-
14	2	1	12	23.24	58.83	-
13	2	2	14	74.12	0.33	-
12	3	2	19	175.75	323.60	0.14
11	4	2	24	0.03	0.02	0.05

Table 5.7: Optimal Solutions for the TCS Problem for the *FDCT* benchmark (Multicycle Multipliers)

#steps	$*_m$	ALU	Area	Refined ILP(F)	Refined ILP(R)	Vanilla ILP
34				-	-	-
33				-	-	-
32				-	-	-
31				-	-	-
30				-	-	-
29				-	-	-
28				-	-	-
27				-	-	-
26				-	-	-
25	2	2	14	19.59	-	-
24	2	2	14	263.44	16.1	-
23	2	2	14	18.08	11.59	-
22	2	2	14	22.10	9.95	-
21	2	2	14	489.23	7.8	-
20	2	2	14	95.53	-	-
19	2	2	14	187.35	107.26	-
18	2	2	14	-	35.57	-
17	3	2	19	63.04	60.52	-
16	3	2	19	27.34	50.03	-
15	3	2	19	19.4	-	-
14				-	-	-
13				-	-	-
12	4	3	26	5.14	5.22	-
11	4	3	26	2.14	18.72	-
10	5	3	31	185.93	272.72	-
09	8	3	46	1721.97	168.27	0.51
08	8	4	48	0.08	0.05	0.11

Table 5.8: Optimal Solutions for the TCS Problem for the *FDCT* benchmark(Pipelined Multipliers)

#steps	$*_p$	ALU	Area	Refined ILP(F)	Refined ILP(R)	Vanilla ILP
26				-	-	-
25	1	2	9	28.36	-	-
24	1	2	9	-	-	-
23	1	2	9	253.99	214.14	-
22	1	2	9	175.56	-	-
21	1	2	9	-	614.72	-
20	1	2	9	164.95	618.31	-
19				-	-	-
18	2	2	14	38.42	44.29	-
17	2	2	14	11.89	40.48	-
16	2	2	14	6.67	758.38	-
15	2	2	14	18.63	386.01	-
14	2	2	14	14.02	1444.42	-
13	2	2	14	-	63.75	-
12	2	3	16	5.31	5.01	-
11	2	4	18	-	0.89	-
10	3	3	21	10.58	2129.09	-
09	4	4*	28	14.34	9.79	0.62
08	10	8	48	0.07	0.16	0.15

Table 5.9: Scheduling times for the RCS Problem for the *EWF* benchmark

$*_p$	$*_m$	ALU	#steps	Vanilla ILP	Refined ILP	Custom BAB	UB	LB
0	1	1	28	-	3.72	< 0.01	28	28
0	1	2	21	3.65	1.92	< 0.01	22	21
0	2	1	28	-	3.40	< 0.01	28	28
0	2	2	18	0.05	0.33	< 0.01	18	18
0	2	3	18		0.02	< 0.01	18	18
0	3	2	18		0.36	< 0.01	18	18
0	3	3	17	0.05	0.01	< 0.01	17	17
1	0	1	28	-	3.00	< 0.01	28	28
1	0	2	19	0.42	2.33	0.01	19	19
1	0	3	18	0.08	0.02	< 0.01	18	18
2	0	1	28	-	3.70	< 0.01	28	28
2	0	2	18	3.02	2.51	0.02	18	18
2	0	3	18	< 0.01	0.01	< 0.01	17	17

Table 5.10: Scheduling times for the RCS Problem for the *ARF* benchmark

$*_p$	$*_m$	ALU	#steps	Vanilla ILP	Refined ILP	Custom BAB	UB	LB
0	1	1	34	-	8.52	< 0.01	34	34
0	1	2	34	-	10.26	< 0.01	34	34
0	2	1	18	-	0.32	< 0.01	18	18
0	2	2	18	-	0.45	< 0.01	18	18
0	2	3	15	908.15	293.05	44.63	15	15
0	3	1	16	726.26	0.01	< 0.01	16	16
0	4	1	16	3.06	0.01	< 0.01	16	16
0	4	2	11	0.02	0.01	< 0.01	11	11
1	0	1	19	1403.75	0.39	< 0.01	19	19
1	0	2	19	0.65	0.37	< 0.01	19	19
1	0	3	19			< 0.01	19	19
2	0	2	13	0.67	0.04	< 0.01	13	13
2	0	3	13	0.12	0.04	< 0.01	13	13
3	0	2	13	0.26	0.04	< 0.01	13	13
4	0	2	11	0.02	0.01	< 0.01	11	11

Table 5.11: Scheduling times for the RCS Problem for the *FDCT* benchmark

$*_p$	$*_m$	ALU	#steps	Vanilla ILP	Refined ILP	Custom BAB	UB	LB
0	1	1	34	-	< 0.01	< 0.01	34	34
0	1	2	34	-	< 0.01	< 0.01	34	34
0	2	1	26	-	< 0.01	< 0.01	26	26
0	2	2	18	-	< 0.01	< 0.01	18	18
0	2	3	18	-	< 0.01	< 0.01	18	18
0	3	2	14	-	< 0.01	< 0.01	14	14
0	4	2	13	2.36	< 0.01	< 0.01	13	13
0	4	3	11	12.80	-	2100	11	11
0	4	4	11	0.42	< 0.01	< 0.01	11	11
0	4	5	11	0.41	< 0.01	< 0.01	11	11
0	5	4	10	0.19	< 0.01	< 0.01	10	10
0	5	5	10	0.19	< 0.01	< 0.01	10	10
0	8	4	08	0.05	< 0.01	< 0.01	08	08
1	0	1	26	-	< 0.01	< 0.01	26	26
2	0	1	26	-	< 0.01	< 0.01	26	26
1	0	2	19	8.18	< 0.01	< 0.01	19	19
2	0	2	13	2.51	< 0.01	< 0.01	13	13
3	0	2	13	2.18	< 0.01	< 0.01	13	13
2	0	3	12	267.95	112.53	< 0.01	12	12
3	0	3	10	0.22	< 0.01	< 0.01	10	10
4	0	4	09	0.18	< 0.01	< 0.01	09	09
8	0	4	08	0.04	< 0.01	< 0.01	08	08

Chapter 6

Conclusions

The importance of optimizing the scheduling process is increasing because an increasing number of systems are resorting to exact schedules as a means of improving design performance. Therefore, there is a need for techniques for computing exact solutions to this problem quickly.

In this thesis, the scheduling problem for high level synthesis was formalized, analyzed, and efficient algorithms for computing optimal solutions to this problems were presented. The main contributions of this work are as follows

- Lower bounding algorithms were presented, which were then compared, both analytically and experimentally, with existing schemes.
- Techniques to refine estimates on the lifetimes of operations were shown.
- Ways of improving existing heuristics using the refined lifetimes of the operations were demonstrated.
- Procedures for improving the efficiency of ILP based techniques were presented.
- A new branch and bound algorithm was developed, which was experimentally shown to be faster than ILP for a majority of cases. This algorithm has the added advantage that it is numeraically more stable than ILP.

While the work done has reduced the time taken to compute exact schedules, there is still work to be done, especially in developing algorithms which will be able to exploit graph based information that can easily be derived from the graph. Besides this, the major limitation of the work done is that the scheduling algorithms presented could have only time or area as the optimizing criteria. There is a need for developing scheduling algorithms as well as bounding routines for optimizing parameters such as registers and interconnect. Alternate avenues for further research include devoloping additional valid inequalities and parallelizing the code.

Bibliography

[Bar95] Peter Barth. A davis-putnam based enumerative algorithm for linear pseudo-boolean optimization. Technical report, Max-Planck Institut Für Informatik, Jan 1995.

[BW95] Stephan A. Blythe and Robert A. Walker. A methodology for scheduling in 3D design space. In *Proceedings of the 8th International Symposium on System Synthesis*, pages 78–83, Cannes, France, Sep 1995. ACM-IEEE.

[BW96] Stephan A. Blythe and Robert A. Walker. Toward a practical methodology for completely charactrizing the optimal design space. In *Proceedings of the 9th International Symposium on System Synthesis*, pages 8–13, La Jolla, CA, Nov 1996. ACM-IEEE.

[CW91] R. Camposano and W. Wolf. *High-Level VLSI Synthesis*. Kluwer Academic, 1991.

[CW93] Samit Chaudhuri and Robert A. Walker. The structure of assignment, precedence and resource constraints in the ilp approach to the scheduling problem. In *Proceedings of the International conference on Computer Design*, pages 25–29, Cambridge, MA, Oct. 1993.

[CW94] Samit Chaudhuri and Robert A. Walker. Analyzing and exploiting the structure of the constraints in the ILP approach to the scheduling problem. *IEEE Transactions on Very Large Scale Integration*, 2(4):456–471, Dec. 1994.

[CW96] Samit Chaudhuri and Robert A. Walker. Computing bounds on functional units before scheduling. *IEEE Transactions on Very Large Scale Integration*, 4(2):273–279, June 1996.

[dac87] ACM and IEEE Computer Society. *Proceedings of the 24th ACM/IEEE Design Automation Conference*, Miami Beach, June 1987.

[De 94] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[FB73a] E. B. Fernández and B. Bussel. Bounds on the number of processors and time for multiprocessors. *IEEE Transactions on Computers*, C-22:745–751, Aug 1973.

[FB73b] E. B. Fernańdez and B. Bussell. Bounds on the number of processors and time for multiprocessor optimal schedule. *IEEE Transactions on Computers*, C-22:745–751, Aug 1973.

[GDP86] D. Gajski, N. Dut, and B. Pangrle. Silicon compilation (tutorial). In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 102–110, May 1986.

[GDWL92] D. Gajski, N. Dutt, A. Wu, and S. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic, 1992.

[GE90] C. H. Gebotys and M. I. Elmasry. A global optimization approach for architectural synthesis. In *ICCAD*, pages 258–261, Santa Clara, CA, Nov 1990.

[GE93] Catherine H. Gebotys and Mohamed I. Elmasry. Global optimization approach for architectural synthesis. *IEEE Trans. Computer-Aided Design*, 12(9):1266–1278, September 1993.

[HC94] Y. Hu and B.S. Carlson. Improved lower bounds for the scheduling optimization problem. In *Proceedings of the 1994 IEEE International Symposium on Circuits and Systems*, pages 295–298, London, England, June 1994.

[HGC93] Y. Hu, A. Ghouse, and B.S. Carlson. Lower bounds on the iteration time and the number of resources for functional pipelined data flow graphs. In *Proceedings of the IEEE International Conference on Computer Design*, pages 21–24, Cambridge, MA, Oct. 1993. IEEE Computer Press.

[HH93a] C. T. Hwang and Y. C. Hsu. PLS: A scheduler for pipeline synthesis. *IEEE Trans. Computer-Aided Design*, 12(9):1279–1286, Sept. 1993.

[HH93b] C. T. Hwang and Y. C. Hsu. Zone scheduling. *IEEE Trans. Computer-Aided Design*, 12(7):926–933, July 1993.

[HLH91] C. T. Hwang, T. H. Lee, and Y. C. Hsu. A formal approach to the scheduling problem in high level synthesis. *IEEE Trans. Computer-Aided Design*, 10(4):464–475, 1991.

[Hu61] T.C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848, November 1961.

[KP87] F.J. Kurdahi and A.C. Parker. REAL: A program for REgister ALlocation. In dac87 [dac87], pages 210–215.

[LC96] M. Langevin and E. Cerny. A recursive technique for computing lower-bound performance of schedules. *ACM Trans. Design Automation of Electronic Systems*, 1(4):443–456, Oct. 1996.

[Lin97] Youn-Long Lin. Recent developments in high-level synthesis. *ACM Trans. Design Automation of Electronic Systems*, 2(1):2–21, Jan. 1997.

[MPC90] M.C. McFarland, A.C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, February 1990.

[NR97] M. Narasimhan and J. Ramanujam. A fast approach to computing exact solutions to the resource-constrained scheduling problem, Jul. 1997. Submitted for publication.

[NW88] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* Wiley–Interscience, 1988.

[OKD94] S. Y. Ohm, F. H. Kurdahi, and N. Dutt. Comprehensive lower bound estimation from behavioral descriptions. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, San Jose, Calif., 1994. IEEE Computer Society.

[Pau91] P.G. Paulin. Global scheduling and allocation algorithms in the hal system. In Camposano and Wolf [CW91], pages 255–282.

[PK87] P.G. Paulin and J.P. Knight. Force directed scheduling in automatic data path synthesis. In dac87 [dac87], pages 195–202.

[PK89] P.G. Paulin and J.P. Knight. Force-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Trans. Computer-Aided Design*, 8(6):661–679, June 1989.

[PR88] R. Gary Parker and Ronald L. Rardin. *Discrete Optimization.* Academic Press, Boston, 1988.

[RCJ72] C.V. Ramamoorthy, K.M. Chandy, and M.J. Gonzalez Jr. Optimal scheduling strategies in a multiprocessor system. *IEEE Transactions on Computers*, C-21(2):137–146, February 1972.

[RJ94] Minjoong Rim and Rajiv Jain. Lower bound performance estimation for the high-level synthesis scheduling problem. *IEEE Trans. Computer-Aided Design*, 13(4):451–458, April 1994.

[RK92] C. Ramachandran and F.J. Kurdahi. Combined topological and functionality based delay estimation using a layout-driven approach for high level applications. In *Proceedings of the European Design Automation Conference*, pages 72–78, Hamburg, September 1992. IEEE Computer Society Press.

[SJ93] A. Sharma and Rajiv Jain. Estimating architectural resources and performance for high-level synthesis applications. *IEEE Transactions on Very Large Scale Integration*, 1(2):175–190, June 1993.

[SM89] Harvey M. Salkin and Kamlesh Mathur. *Foundations of Integer Programming.* North-Holland, New York, 1989.

[Ull75] J.D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, June 1975.

[Wal97] Robert A. Walker. Personal communication, May. 1997.

[Wil85] H. P. Williams. *Model Building in Mathematical Programming.* Wiley, New York, second edition, 1985.