EFFICIENT TECHNIQUES FOR PROBLEMS IN 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
Sandeep Deshpande
B.E., Jamia Millia Islamia, New Delhi, India, 1996
August 1998

Acknowledgments

I would like to express my gratitude to Dr. J. Ramanujam for his guidance throughout this work. His advise on matters other than the thesis work has also been very useful. I enjoyed working with him. I would like to thank Dr. J. L. Trahan and Dr. Subhash C. Kak for serving on my committee. I would like to express my gratitude to my parents and my sister for their constant encouragement. Last, but not the least, I would like to thank all my friends, especially in the EE department, who made my stay at LSU a pleasant one.

Contents

List of Tables

List of Figures

Abstract

Efficient techniques for design space exploration in high-level synthesis are critical for the automatic synthesis of optimal or near-optimal VLSI designs. High-level synthesis consists of several phases. An important phase that has a tremendous impact on the final design is the *scheduling* phase that is concerned with the distribution of various operations in time. Although the scheduling problem can be modeled as an integer linear programming problem in order to obtain optimal solutions, such a modeling under a variety of constraints can be cumbersome and the time to solve them is prohibitively high. It is therefore essential that heuristic solutions for the scheduling problems gain importance. This thesis concentrates on the improvement of the existing and development of new lower bounds as well as heuristics for scheduling problems under both resource and time (performance) constraints. In addition to scheduling, good *clock selection* impacts performance. In most synthesis systems, a convenient value of the clock is chosen or exact methods have been used for clock selection. This thesis discusses a novel heuristic approach for near-optimal clock selection for synthesis systems. Finally, this thesis presents a new approach to the *design space exploration* problem. Unlike other approaches that use exact solutions (and therefore are extremely time-consuming) or lower bounds (and therefore are inaccurate), the solution presented here uses a clever combination of lower and upper bounds while resorting to exact solutions only when necessary. This is shown to be very effective in reducing the time required in design space exploration.

Chapter 1

Introduction

Technological advances in VLSI design over the past decades have made it possible to fabricate a wide variety of large and complex integrated circuits. Computer Aided Design (CAD) tools have been extremely useful in circuit simulation, placement, routing and floorplanning; these primarily aid in design at a lower level of abstraction. There is a critical need for CAD tools at higher levels of design abstraction since the time-to-market for optimized large designs has become an important factor in the microelectronics industry.

Automating the design of integrated circuits offers several advantages including (a) shorter design cycles; (b) better exploration of different designs due to the ability to generate and evaluate the various possibilities in a short time; and (c) better quality of designs.

There are two major approaches to the problem of efficiently designing VLSI circuits: [GDWL92]:

Capture and simulation methodology, which advocates that good design knowledge and experience is essential and the design process cannot be replaced by tools that automate it. The focus is on bottom-up design, starting from the basic components such as transistors and gates and then building the circuit. The CAD tools used with this approach must be well equipped to simulate the design based on various parameters that affect the circuit performance.

Describe and synthesize methodology, which is based on the notion that synthesis tools can generate better designs in a shorter time compared to a human designer, and certainly help a designer better explore design options at the early stages. It is a top-down design process that focuses on the behavioral description of the operations to be performed by the design and the CAD tools are responsible for the netlist generation and then the migration to silicon. This is an interactive synthesis process wherein an experienced designer can aid the development of efficient algorithms for reducing the design space very fast.

Probably the best approach is to use a combination of both these approaches; however synthesis tools of the latter kind that generate near-optimal solutions are gaining popularity since they are cost-effective and reduce the time to-market.

Application Specific Integrated Circuits (ASIC's) can be simulated and manufactured in a very short time with the increasing automation. *Synthesis* as used in this thesis is the process of translation from a behavioral to a structural description; *High Level Synthesis* (HLS) is the process of translating a behavioral description to a register transfer level (RTL) structural description [Lin97]. A behavioral description of the problem is a natural starting point for high level synthesis. Hardware Description Languages (HDL) can capture the behavioral description and generate netlists and also allow the designer to simulate the design with a set of constraints like concurrency, timing, area and interconnect, to mention a few. Much like the expression of algorithms in high-level programming languages, HDLs allow designers to specify designs at high level of abstraction without being too conerned about the details of an actual implementation. One of the main reasons for the growing use of EDA tools in the microelectronics industry is their use in design space exploration, a technique that allows the designer to study the various parameters affecting the area and performance of the design. This process can make it possible for a designer to select an "optimal" design that conforms to various specifications. Although synthesis tools have a great potential for automating the process of design space exploration, formulating algorithms for efficient synthesis systems that can generate optimal solutions under multiple constraints is a challenging task.

A control data flow graph (CDFG) is used as an input to most of the high level synthesis systems. The data flow graph (DFG) captures the flow of data between operations and the dependency constraints between them. The control flow arises from the conditional branches and looping constructs in the behavioral description. In this thesis we are concerned with DFGs only, although the work needs to be extended to incorporate conditionals also. The three main phases of high level synthesis are *scheduling*, *allocation* and *binding*. The process of scheduling is the assignment of operations to control steps under a variety of constraints. The scheduling problem can be solved as an optimization problem, its quality of solution being measured in terms an objective function that is being minimized or maximized. In our discussion we concentrate on two kinds of scheduling techniques:

- Resource Constrained Scheduling (RCS) is a technique that minimizes the latency, given a set of resources.

- Time Constrained Scheduling (TCS) is a technique that minimizes the area of a design, given a fixed latency.

The control step mentioned above is generally equal to the system clock. The value of the clock is also an important factor affecting the quality of schedules we will show in the chapter on near optimal selection. Allocation is the selection of functional units or resources of a particular resource class. The functional units can be drawn from a single or different *libraries*. The high level synthesis system should be free to choose from any library to obtain the best possible design. The binding phase involves mapping of operations to functional units, intermediate variables to registers and data transfers to interconnect or buses.

In this thesis we will be concentrating on the scheduling and clock selection phases of high level synthesis and we assume that allocation and binding has already been done. In addition, design space exploration is discussed in detail. Design Space Exploration involves the computation of upper and lower bounds (heuristics) to get a fast characterization of the design space. Although the ideal approach is to formulate the problem as an *integer linear programming* problem and optimize for a given objective function, if the fraction of the time for this process in the complete design cycle is quite large, then such synthesis systems would be of little use. Recent research focuses on improving the run time for ILP based scheduling [NR97]. Another approach is to concentrate on formulating better algorithms for *near* optimal solutions that can be obtained in a relatively much shorter time. In this thesis, we concentrate on heuristics that reduce the time to explore the design space and also do not compromise on the quality of the solutions obtained.

1.1 Organization of this thesis

This thesis is organized as follows. Initially, in Chapter 2, we take a brief look at the background and certain assumptions for the scope of this work. In Chapter 3 and Chapter 4, we discuss improvements in obtaining lower bounds for the RCS problem and upper bounds for the TCS problem, respectively. Chapter 5 presents a new methodology for near optimal clock selection and we conclude with a novel approach for fast area-latency curve prediction in Chapter 6. We summarize this thesis and conclude in Chapter 7.

Chapter 2

Background and Notation

In this chapter we present the mathematical notation that we would use in the rest of this thesis, and provide a background on the scheduling in high-level synthesis. Many applications and algorithms in areas such as signal processing and solution of differential equations are prime candidates for implementation in hardware; it is no surprise that these algorithms are used as standard benchmarks to evaluate the effectivess of high-level synthesis. These applications are usually modeled using graphs, as either *data flow graphs,* or *sequencing graphs.*

In view of this we define some basic terminology related to graph theory. Then we discuss how the information in the graph can be used to formulate the scheduling problem and look at some basic scheduling algorithms.

2.1 Graphs

A graph G is a pair (V, E) where V is the set of vertices and E is the set of edges in the graph. Since the sequencing of operations to be performed is modeled as a graph, each operation to be performed corresponds to a vertex (or node) in the graph. We use the terms operations, vertices and nodes interchangeably. The set of edges in a directed graph is an ordered pair of vertices. We denote an edge $e_{i,j}$ as the ordered pair $\langle v_i, v_j \rangle$, such that the origin of the edge is the vertex v_i and the destination is the vertex v_j. Such an edge $e_{i,j}$ denotes the dependency of operation j on operation i, i.e., operation j can not begin executing till operation i completes. A path exists between vertices v_0 and v_n if there is a sequence of vertices $\langle v_0, v_1, \cdots, v_n\rangle$ and edges $e_{i,i+1}$ (for $0 \leq i \leq n-1$) between two successive vertices v_i and v_{i+1} in the path. For a cyclic graph, there may be at least one vertex v_k that is an origin for some edge $e_{k,m}$ and a destination for another edge $e_{n,k}$, where the nodes v_n, v_k, v_m all belong to the path. If this condition does not occur in a graph, then the graph is said to be acyclic. In this thesis, we consider only the scheduling of problems modeled using Directed Acyclic Graphs (DAGs).

For a given graph $G = (V, E)$, the predecessor subgraph with respect to any vertex v_i denoted by $G_{pred}(v_i)$ is the subgraph induced by the set of all nodes $u \in V$ such that there is a path from u to v_i. The successor subgraph of v_i written as $G_{succ}(v_i)$ is defined (similarly) as the set of all nodes $u \in V$ such that there is path from v_i to u. Basic graph algorithms such as depth-first search and breadth-first search [CLR90] can be used to find the predecessor and successor subgraphs of a node in a dag. In addition, these algorithms can be used find a numbering of the nodes which (depending on the kind of numbering used) can be extremely useful in improving the performance of the techniques used in high-level synthesis. A very useful numbering of the nodes of a dag is the *topological numbering.* The nodes of a dag are said to *topologically sorted* if to each node v_i we assign a (topological) number t_i such that for every edge $\langle v_i, v_j\rangle \in E$, $t_i < t_j$. A reverse graph G_r can be defined as $G(V, E')$ where E' is the set of vertices of the form $e_{j,i}$, for each $e_{i,j} \in E$. If there was an edge $\langle v_i, v_j\rangle$ in the original (forward) graph, then in the reverse graph contains the edge $\langle v_j, v_i\rangle$.

The scheduling problem can be cast as an optimization of an objective function which is generally the *area* or the *latency* of the design. Here we give an illustration of a general Linear Programming (*LP*) and an Integer Linear Programming (*ILP*) formulation and then in the following sections we formulate the problems individually in detail. The *LP* problem is as follows

$$\begin{aligned} &\text{Minimize} \quad \vec{c}\cdot\vec{x} \quad \text{subject to} \\ &\vec{P}\cdot\vec{x} \geq \vec{q} \\ &\vec{x} \geq 0 \\ &\vec{x} = (x_1, x_2, \cdots, x_n), \qquad x_i \in R \end{aligned} \tag{2.1}$$

where P is some matrix of size $m \times n$. There are known polynomial time algorithms to solve the *LP* problem. If x_i is supposed to be integral, then, the *ILP* formulation is the same but for the last constraint, which is modified to

$$\vec{x} = (x_1, x_2, \cdots, x_n), \qquad x_i \in Z. \tag{2.2}$$

There are no known polynomial time algorithms to solve the *ILP* problem, and it is NP-complete. For such a problem, a lower bound on the value of the optimal solution can be found by relaxing some of its constraints.

2.2 Assumptions for scheduling

For a given data flow (sequencing) graph, there can be a number of schedules possible depending on the kind of constraints and the objective function. We have a valid schedule if and only if the precedence constraints between operations are satisfied. Let Φ denote the set of all valid schedules, then any *feasible* schedule that satisfies some optimization criterion in addition to the precedence constraints is denoted by ϕ and $\phi \in \Phi$. In scheduling, we make certain assumptions regarding the various parameters affecting the schedule. We assume that each control step for scheduling is equal to the clock period of the system. We will denote the clock period by *clk*. Each operation that is performed on some *functional unit/resource* of a certain *class/type*. The resource utilization (or availability) in a particular schedule ϕ can be expressed as a resource vector $\vec{r}$ defined as follows

$$\vec{r} = \{r_1, r_2, ..., r_m\} \tag{2.3}$$

where, r_i is the number of resources of the type i. Each operation in the sequencing graph is mapped onto a functional unit. The functional unit can be an adder, multiplier, ALU, subtracter, etc., which can be obtained from various libraries. The major attributes such as area and delay of the functional unit are dependent on the library from which it is chosen. The delay of an operation mapped to vertex v_i in the graph is denoted by $delay(v_i)$. Thus if there is a dependence between two operations v_i and v_j such that there is an edge from v_i to v_j, then the latter operation can start only $delay(v_i)$ time later than the start time of the former operation. This constraint can be written in the form of an inequality as $\mathrm{T}(v_j) \geq \mathrm{T}(v_i) + delay(v_i)$, where T denotes the start time of a particular operation. The delay of a functional unit can also be mentioned in terms of the number of clock cycles it takes for the operation to complete execution. If the delay of an operation is greater than the clock cycle, then the functional unit is active over multiple steps, the number of steps being $\left\lceil \frac{delay(v_i)}{clk} \right\rceil$. This is called *multicycling* and the operation is said to be a *multicycle* operation . We consider multicycle operations without pipelining, thus we can consider the existence of latches between the multiple stages. In the non-pipelined case, while the functional unit is active, it cannot be shared by any other operation. Another feature is *chaining* which allows several operations to be executed in a single cycle if the sum of the delays of the operations is less than the clock cycle time.

2.3 Unconstrained schedules: ASAP and ALAP

There are two basic scheduling algorithms to which we shall be referring in the course of our discussion. The As Soon As Possible (ASAP) schedule is denoted by ϕ_{ASAP}. This is an unconstrained schedule and finds the least possible time for the schedule. Only precedence

Procedure $\mathsf{ASAPSchedule}$
For a Graph $G = (V, E)$, topologically sort the vertices $v_i \in V$
for each $v_i \in V$ **do**
    $e(v_i) \leftarrow 0$
enddo
$\lambda(\phi_{\mathsf{ASAP}}) \leftarrow 0$
for each $v_i \in$ *Topologically Sorted Order of G* **do**
    for each u_i adjacent to v_i **do**
        $e(u_i) \leftarrow \max\{e(u_i), e(v_i) + delay(v_i)\}$
    enddo
    $\lambda(\phi_{\mathsf{ASAP}}) \leftarrow \max\{\lambda(\phi_{\mathsf{ASAP}}), e(v_i) + delay(v_i)\}$
enddo

Figure 2.1: ASAP Algorithm

constraints need to be satisfied in this case. On the other hand, As Late As Possible algorithm is a latency constrained algorithm that tries to schedule an operation as late as possible without the schedule length violating the given latency constraint. The ASAP schedule time for an operation v_i is denoted by $\mathsf{ASAP}(v_i)$ and the ALAP schedule time of an operation is denoted by $\mathsf{ALAP}(v_i)$. The absolute difference in the two values for an operation is the mobility or a measure of *freedom* of the operation. The two algorithms are given in figures. The earliest time an operation, v_i, can be scheduled, denoted by $e(v_i)$, is the earliest time after all its predecessors have been scheduled. If an operation does not have any predecessors, then, it can be scheduled in the first time step. Similarly, $l(v_i)$ is the latest time an operation can be scheduled without violating the given time constraint. We specify the latency of a schedule to be λ.

2.4 RCS Problem

In a sequencing graph, an operation i is modeled by a node v_i. We consider a *sequencing graph* $G = (V, E)$, which is a behavioral representation of the operations to be performed. Here V is the set of vertices such that $v_i \in V$ and E is the set of edges such that there is an edge $e_{i,j} \in E$ if there is a precedence constraint from operation i to operation j. Since each

Procedure ALAPSchedule
ASAPSchedule(G, λ)
for each $v_i \in V$ in *reverse topological order of* G **do**
  $l(v_i) \leftarrow \lambda - delay(v_i)$
  for each u_i adjacent to v_i **do**
    $l(v_i) \leftarrow \min\{l(v_i), l(u_i) - delay(u_i)\}$
  enddo
enddo

Figure 2.2: ALAP Algorithm

operation is mapped on a functional unit of a particular type, we introduce an additional subscript for an operation, to accommodate the type. Hence an operation $v_{i,k}$ would be some operation i mapped to a functional unit of type k. In this thesis, the scope of the resource constraints only include a constraint on the number of functional units of different types that can be used. This can, however, be extended to include resource constraints on the number of registers, buses, etc. Let there be k resource classes; let N_j be the number of operations in the sequencing graph belonging to the resource class j and r_j be the number of available resources (constraint) of the class j. If N is the total number of operations in the graph, then we have $\sum_{j=1}^{k} N_j = N$. An operation is active for a time $delay(v_i)$ from the time it starts. If $\mathcal{T}$ is the set of time steps in a valid schedule ϕ, then the number of resources active in a particular time step $l \in \mathcal{T}$ for a valid schedule ϕ never exceeds the maximum resources available. In addition to this, all operations should satisfy the condition $v_i \in \phi$, i.e., all operations have to be scheduled. A precedence constraint between two operations v_i and v_j such that there is an edge $e_{i,j}$ between the two operations in the graph should also be satisfied. The set of constraints for the RCS problem can be formally summarized as follows. A binary variable x_i is used to model the start time of each operation v_i such that start time of each operation in ϕ is unique and each operation starts executing in some only one time step in the schedule. Thus for an operation v_i,

$$\sum_{l=1}^{\lambda} x_{il} = 1, \qquad \forall l \in \mathcal{T}. \tag{2.4}$$

Minimize $\quad \left\{ \sum_{l=1}^{\lambda} l \cdot x_{Nl} \right\} \quad$ subject to

$$\sum_{l=1}^{\lambda} x_{il} = 1, \qquad \forall i \in [0, 1, ..., n] \tag{2.5}$$

$$\sum_{l=1}^{\lambda} l \cdot x_{il} + delay(i) \leq \sum_{l=1}^{\lambda} l \cdot x_{jl}, \qquad \forall i, j \in E \tag{2.6}$$

$$\sum_{i=1}^{N} x_{il} \leq r_k, \qquad \forall l, k \in \mathcal{R}. \tag{2.7}$$

The above problem (P_1, say) is an optimization problem for the minimum latency schedule. The solution of a *relaxation* problem (P_2, say) of P_1 is useful for obtaining the lower bounds for the RCS problem. The dependence constraint relation in Equation 2.6 is relaxed. Rim and Jain [RJ94] illustrate a lower bounding algorithm based on such a relaxation. Let ϕ_{ASAP} and ϕ_{ALAP} represent the ASAP and ALAP schedules respectively for the operations in the graph G. $\mathsf{ASAP}(v_i)$ refers to the time step at which an operation v_i is scheduled in ϕ_{ASAP} and the $\mathsf{ALAP}(v_i)$ refers to the time step at which an operation v_i is scheduled in ϕ_{ALAP}. The time required for ϕ_{ASAP} is the critical path time $\lambda_{critical}$ for the graph. If the latency for executing all the operations is λ, then the time span ($\mathsf{TimeFrame}(v_i)$) in which the operation v_i is guaranteed to be scheduled for a valid schedule is

$$\sum_{l \in T} l \cdot x_{i,l} = \mathsf{ASAP}(v_i) - \{\lambda - (\lambda_{critical} - \mathsf{ALAP}(v_i)\}. \tag{2.8}$$

2.4.1 Chaining and Multicycling

The above conditions can also be modified to support chaining and multicycling of operations as follows. Let there be a precedence constraint between two operations v_i and v_j (represented by the edge $e_{i,j}$ between them) that can be chained. Then,

$$\mathrm{T}(v_i) - \mathrm{T}(v_j) \leq 0 \qquad v_i \rightarrow v_j, \tag{2.9}$$

where $\mathsf{T}(v_i) = \sum_{l=1}^{\lambda} l \cdot x_{il}$ is the time at which an operation v_i has been scheduled.

If there exists a precedence constraint between two operations v_i and v_k by virtue of chaining, then,

$$\mathrm{T}(v_i) - \mathrm{T}(v_k) \leq 1. \tag{2.10}$$

If the delay of an operation is greater than the clock cycle, then the operation occupies multiple steps. We consider only non-pipelined multicycling. The number of resources of type k active in a time-step l includes all resources that have been initiated $(l - delay(k) + 1)$ time-steps earlier. To incorporate this condition, we have,

$$\sum_{q=0}^{delay(k)-1} \sum_{i=1}^{N} x_{i,l-q} \leq r_k \qquad \forall l, \forall k \in \mathcal{R} \tag{2.11}$$

Also, for two operations v_i and v_j having a precedence constraint between them, the constraint that $\{\mathsf{T}(v_i) + delay(v_i) \leq \mathsf{T}(v_j)\}$ as given earlier still holds. In [RJ94], the constraint is expressed in the following manner: A multicycle operation executing over m steps can be thought of as a set of m chained unicycle operations, which leads to an additional constraint in the formulation of the RCS problem. Suppose v_i and v_j are chained unicycle operations, then the constraint for the consecutive scheduling of the operations is as follows

$$\sum_{l \in \mathcal{T}} l \cdot x_{i,l} + 1 = \sum_{l \in \mathcal{T}} l \cdot x_{j,l}. \tag{2.12}$$

2.5 TCS problem

The use of a functional unit or a resource has some cost associated with it. The objective of the TCS problem is to reduce the cost of utilization of resources in the design. The cost is minimized if the resources are fully utilized in a system. We know that operations in a sequencing graph are mapped to a particular type of resource. The operation of each type should be uniformly distributed throughout the control steps in a schedule. This can be done by minimizing the maximum number of operations of each type in a given control step. According to our assumptions, the number of resources of type k is denoted by r_k and $x_{i,l}$ represents the binary variable associated with start time of an operation v_i in the time step $l \in \mathcal{T}$. Let the total number of resource types be k and the total number of operations in the sequencing graph be N. If λ is the latency of the schedule and c_k is the cost of each resource of the type r_k , then using the idea in [HHL91], we formulate the optimization problem as follows.

Minimize $\sum_{k=1}^{m} \{c_k * r_k\}$ subject to

$$\sum x_{i,l} \leq r_k, \qquad l \in [1, \lambda] \tag{2.13}$$

$$\sum_{l=\mathsf{ASAP}(v_i)}^{\mathsf{ALAP}(v_i)} x_{i,l} = 1, \qquad i \in [1, n] \tag{2.14}$$

$$\sum_{l=\mathsf{ASAP}(v_i)}^{\mathsf{ALAP}(v_i)} \{l \cdot x_{i,l}\} - \sum_{l=\mathsf{ASAP}(v_j)}^{\mathsf{ALAP}(v_j)} \{l \cdot x_{j,l}\} \leq -1, \qquad v_i \to v_j. \tag{2.15}$$

Chapter 3

RCS Lower Bounding Techniques

An important phase of the design space exploration problem is the computation of good lower bounds on either the *area* or the *latency* of the design. In this chapter we present an improvement over the existing techniques for finding lower bounds on the latency or the time to complete a particular schedule based on resource constraints. Since the objective function of an optimization problem can never have a value better than a lower bound, the optimal solution will be greater than or at best be equal to the lower bound. We say that the lower bounding technique gets better or *tighter* as its value approaches the optimal solution. Ideally, we would want the lower bound to be equal to the optimal solution so that there is no need to solve the problem at hand. In this section a formulation of the RCS problem is discussed first, then an adaptation is made for the RCS lower bounding technique.

3.1 Previous Work

The ASAP and ALAP values of an operation reflect the particular interval to which an operation is guaranteed to be scheduled in some valid schedule ϕ. The greedy algorithm in [RJ94] uses this idea as a relaxation of the precedence constraints. The greedy algorithm for finding RCS lower bounds in [RJ94] is paraphrased in the Figure 3.1. [LC96] considers the notion of finding a recursive lower bound by considering subgraphs of the original graph. It uses the greedy algorithm in [RJ94] to recursively find better ASAP values under constraints

Procedure $\mathsf{RimJainLowerBound}(G(V,E))$
Find ASAP_i and ALAP_i $\forall i \in V$
Form a list L of all operations in the descending order of ALAP_i
for each operation $i \in L$ **do**
  if $(\vec{r} \leq \vec{r_{constraint}})$ and $(\mathsf{ASAP}_i \geq curcstep)$ **then**
    Schedule operation i in $curcstep$
  endif
enddo
$\mathsf{RJLowerBound} = \max_i \{cstep_i - ALAP_i\} + delay_{CP}$

Figure 3.1: Rim-Jain Lower Bound Algorithm

for subgraphs rooted at a given node v_i, but uses the same ALAP values as before. The new ASAP values are then used to calculate a tighter lower bound than [RJ94]. Another method of finding a lower bound is illustrated in [Nar98] which is an extension of the [LC96] technique. It exploits the information in the reverse graph and hence reverse subgraphs to calculate ALAP values under constraints. Thus, it considers constrained ASAP and ALAP values to compute a provably tighter bound that [LC96]. In short, quality of the bounds obtained by these methods can be expressed as

$$[RJ94] \quad \leq \quad [LC96] \quad \leq \quad [Nar98].$$

3.2 A General Approach

3.2.1 Motivation

In design space exploration using RCS, it is known from various algorithms and their results that the RCS upper bounding techniques find solutions that are almost optimal. The existing lower bounding techniques however, compute loose bounds in many cases. Since good lower bounds can improve the quality of other design space exploration techniques like

TCS upper bounds and also finding optimal solutions using Integer Linear Programming, the importance of improving the existing RCS lower bounding methods is essential.

The algorithms discussed in [RJ94], [LC96] and [Nar98] respectively, are successive improvements over the previous methods, there are some particular resource vectors for which there is no improvement in the bounds. As an example, for the *ARFILTER* benchmark, the lower bound on the latency for the resource constraint given by $\vec{r}$=[2 *adders*, 3 *multipliers*], considering a unicycle adder and a multicycle (2 cycles) multiplier, is found to be equal to 13 time steps, whereas the optimal solution is equal to 15 time steps, there being a difference of 2 time steps. The [RJ94] method considers precedence constraints only in the form of Timeframe of the operations. The [LC96] method computes tighter ASAP values by considering subgraphs and accounting for the effect on a particular operation due to the scheduling of its predecessors. The [Nar98] method in addition considers tighter ALAP values. All these methods of lower bound estimation do not take into account the resource demand arising out of operations that are active concurrently in the same time step. The demand on the resources in a particular time step does not arise due to the successors or predecessors of the operations in that time step. To improve the quality of bounds, subgraphs of the given graphs are considered with a view to consider demand due to concurrently active operations and the lower bounding techniques are applied to this portion of the graph. This method of calculating the lower bounds attempts to account for resource demand localized in a particular portion of the graph, the effect of which may not be seen when considering the graph as a whole. This idea is the basis of generating tighter lower bounds on the latency for the RCS problem.

3.2.2 Mathematical Notation

The operations of a certain type that can be concurrently active in a particular time step will compete for a resource of the same resource type. A reasonable estimate of such operations whose predecessors have been scheduled can be obtained from the unconstrained ASAP schedule. Thus, we consider subgraphs of the original graph based on the initial time frames of operations. The subgraphs are based on all possible time intervals in the ASAP schedule. To explain this concept further we introduce some notations that we will be using for the subgraphs used in our lower bounding algorithm. Consider a time interval $[s, t]$ defined by two time steps s and t in the unconstrained ASAP schedule. A subgraph, $G_{s,t} = (V', E')$ is defined such that

$$G_{s,t} \subset G,$$

where,

$$s, t \in T$$

$$s \leq \mathsf{ASAP}(u') \qquad \text{and} \qquad t \geq \mathsf{ALAP}(u') + \mathsf{delay}(u') \qquad \forall u' \in V'$$

Let S be the set of all operations in the graph G, S_{NG} be the set of operations that cannot be guaranteed to be scheduled in the interval $[s, t]$ and S_G be the set of operations that are guaranteed to be scheduled in that interval. These operations are contained in the subgraph $G_{s,t}$. Then an operation $v_i \in S_G$ in the interval satisfies the following condition:

$$\mathsf{TimeFrame}(v_i) \subseteq [s, t].$$

It follows that the set S_{NG} is defined as $S_{NG} = S - S_G$.

3.2.3 The Lower Bounding Technique

Proposition 3.1 *The lower bound obtained by considering portions of the graph defined by the values of 's' and 't' is greater than or at the most equal to the lower bound obtained by considering the whole graph:*

$$\mathsf{LB}(G) = \max\{\forall_{s,t \in T}\{\mathsf{LowerBound}(G_{s,t})\}\} \geq \mathsf{LowerBound}(G) \tag{3.1}$$

Proof. For the above it is assumed that $s < t$. Note that by definition, $G_{s,t}$ is empty if $s > t$. The $\mathsf{LB}(G)$ is the lower bound obtained by repeatedly finding lower bounds on the subgraphs and then taking the maximum. We present here a proof of the above proposition.

In general, according to the values of 's' and 't' the original graph G is split into three subgraphs denoted by $G_{<s}$, $G_{s,t}$ and $G_{>t}$.

$$G_{<s} \cup G_{s,t} \cup G_{>t} = G(V, E) \tag{3.2}$$

The graph $G_{<s}$ is a subgraph containing the operations whose ASAP values are less than the value s, thus those operations are guaranteed not to start in the interval $[s, t]$. Similarly, the operations in the graph $G_{>t}$ are guaranteed not to complete execution in the interval $[s, t]$. As mentioned earlier, the notion of splitting the graphs for finding a tighter lower bound is based on the following idea.

Suppose two nodes $a \in G_{<s}$ and $b \in G_{s,t}$, $a, b \in G$ are of the same type i and have an edge $e_{a,b}$ between them, then they are never going to compete for the same resource in the actual schedule of operations in the graph G. Non-availability of sufficient resources translates into resource demand and since scheduling of operations has to be delayed, it further translates to an increased delay for that graph over its critical path. This concept is used to compute a lower bound on the latency. In general, a resource constrained lower bounding technique LowerBound tries to capture the demand in the unconstrained schedule of the graph G. Extrapolating this, applying the same technique to a portion of the graph guaranteed to be scheduled in a given interval given by $\mathsf{LowerBound}(G_{s,t})$, say, will try to capture the demand due to operations which are concurrently active in that particular interval. The demand due to operations in that interval would tend to increase the unconstrained schedule length of that interval and may delay the original graph G, thus yielding a tighter lower bound on the latency. This particular effect may not be seen if the graph is considered as a whole.

The lower bound to be calculated can now be written as

$$\mathsf{LB}_{s,t}(G) = \{(s-1) + \mathsf{LowerBound}(G_{s,t}) + (\lambda - t)\}. \tag{3.3}$$

where, $\mathsf{LB}_{s,t}(G)$, is the lower bound on the latency for the graph $G = (V, E)$ when a particular subgraph $G_{s,t}$ is considered. It is evident that in the above equation that $\mathsf{LB}(G)$ will reduce to $\mathsf{LowerBound}(G)$ if the value of $s = 1$ and $t = \lambda$, i.e, when the graph is considered as a whole without any subgraphs. In that case it is the conventional RCS lower bound. Suppose for some value of s and t, due to a demand on the resources in that interval, the latency is increased by a value $\triangle\lambda_{s,t}^{demand}$, then according to the above equation, the lower bound on the latency is going to get tighter by that amount. The incremental demand in the *region* $[s, t]$ can then be expressed as

$$\triangle\lambda_{s,t}^{demand} = \mathsf{LB}_{s,t}(G) - \mathsf{LowerBound}(G). \tag{3.4}$$

If the incremental demand is calculated for all s and t such that $s < t$, then the increment in the lower bound on the latency can be expressed as the maximum over increments. Hence, the LB is a function of the resource demand that translates into an increase in the lower bound on the latency. We can say that

$$\lambda_{demand}^{G} = \forall_{s,t} \max_{\forall_{k \in \mathcal{R}}}(\triangle\lambda_{s,t}^{demand}(k)). \tag{3.5}$$

$\triangle\lambda_{s,t}^{demand}(k)$ is the increase in the latency due to demand on the resources of the type k. λ_{demand}^{G} will be the increase in the latency over the critical path delay for graph $G = (V, E)$ that could not have been accounted for when the bounding technique was applied to the whole graph $G = (V, E)$. It follows that

$$\mathsf{LB}(G) = \mathsf{LowerBound}(G) + \lambda_{demand}^{G}. \tag{3.6}$$

It is possible that $\lambda^G_{demand} = 0$. So in general,

$$\mathsf{LB}(G) \geq \mathsf{LowerBound}(G). \tag{3.7}$$

It is important to note that $\mathsf{LowerBound}(G)$ refers to any lower bounding technique which is responsible for the generic nature of the algorithm. Any of the lower bounding techniques discussed in [RJ94], [LC96], [Nar98] can be applied. □

Proposition 3.2 *The value given by* $\mathsf{LB}(G(V,E))$ *is a lower bound on the latency of the graph* $G = (V,E)$.

PROOF. We call the subgraph $G_{s,t} = (V_{s,t}, E_{s,t})$ the constrained portion of the graph $G(V,E)$, since this is the portion for which the RCS lower bounds are found. The lower bounding on the constrained portion generates a schedule $\phi_{\mathsf{LowerBound}}$ having a latency $\lambda(\phi_{\mathsf{LowerBound}})$. For the other subgraphs $G_{<s} = (V_{<s}, E_{<s})$ and $G_{>t} = (V_{>t}, E_{>t})$ the unconstrained schedule is found. The final lower bounding schedule can be expressed as

$$\phi_{\mathsf{LowerBound}}(G) = \phi_{\mathsf{LowerBound}}(G_{s,t}) + \phi_{\mathsf{ASAP}}(G_{<s}) + \phi_{\mathsf{ASAP}}(G_{>t}). \tag{3.8}$$

If Φ is the set of all possible schedules and $\phi_{feasible}$ be any feasible schedule for any given resource constraints, we can express the following relationship for schedules

$$\lambda(\phi_{\mathsf{ASAP}}) \leq \lambda(\phi_{\mathsf{LowerBound}}) \leq \lambda(\phi_{feasible}) \qquad \phi_{\mathsf{ASAP}}, \phi_{\mathsf{LowerBound}} \in \Phi. \tag{3.9}$$

It follows from the above that $\phi_{\mathsf{LB}}(G(V,E))$ is indeed a lower bound for the given resource constraint. □

3.3 An Illustration

Consider all operations of type k in the graph $G = (V, E)$ and let the number of such operations be denoted by n_k. Also assume a resource constraint of r_k resources of the type k. Then a trivial lower bound, SimpleBound, on the number of time steps can be found and hence the LowerBound can be expressed as

$$\mathsf{LowerBound}(G) = \mathsf{SimpleBound}(G) = \max_k \left\{ \left\lceil \frac{n_k}{r_k} \right\rceil \cdot delay(k) \right\}. \tag{3.10}$$

Now consider an interval $[s, t]$ such that $(s - t + 1) < \lambda_{critical}$ where $\lambda_{critical}$ is the number of time steps required for the ϕ_{ASAP} schedule. Any operation $i \in S_{NG}$ guaranteed not to be scheduled in this interval is not considered. Let there be n'_k operations of the type k in the graph $G_{s,t}$ and let the resource constraint still be r_k. Thus, a lower bound on the time, $\lambda_{s,t}$, for the graph $G_{s,t}$ is given by

$$\mathsf{SimpleBound}(G_{s,t}) = \max_k \left\{ \left\lceil \frac{n'_k}{r_k} \right\rceil \cdot \ delay(k) \right\}. \tag{3.11}$$

For the remaining portions of the graph, $G_{<s}$ and $G_{>t}$, the unconstrained latency, which is the length of the critical path (minimum possible latency), is found. Thus, a lower bound on the latency of the graph G can be expressed as

$$\mathsf{LB}_{s,t}(G) = \mathsf{SimpleBound}(G_{s,t}) + \lambda_{critical}(G_{<s}) + \lambda_{critical}(G_{>t}). \tag{3.12}$$

If for some value(s) of s and t such that $s - t + 1 < \lambda_{critical}$, and some resource class k, the number of operations belonging to that class happen to be equal such that $n_k = n'_k$, then

$$\mathsf{SimpleBound}(G_{s,t}) = \mathsf{SimpleBound}(G). \tag{3.13}$$

Hence, it follows that for some s, t

$$\mathsf{LB}_{s,t}(G) \geq \mathsf{LowerBound}(G). \tag{3.14}$$

Here, $\mathsf{LB}_{s,t}(G)$ is the lower bound on the latency for the original graph G while considering the subgraph $G_{s,t}$.

It is evident from the illustration in this section that the trivial lower bound on the latency, which does not take into account any precedence constraints, is greater than or equal to the lower bound obtained by considering no subgraphs. It then follows that if any of the lower bounding techniques as described in [RJ94], [LC96] or [Nar98] are used for the subgraph $G_{s,t}$, then the lower bounds obtained would be tighter than the trivial lower bounding illustrated above.

The algorithm for finding subgraphs is illustrated in Figure 3.2 and the RCS lower bounding algorithm using the subgraphs is shown in Figure 3.3.

3.4 Complexity of the method

Finding the subgraphs is each case depends on the ASAP and ALAP schedules, the complexity for which is $\mathrm{O}(V+E)$. If the latency of the ϕ_{ASAP} schedule is λ_{ASAP}, then the technique has to be performed at the most λ^2 times. The subgraphs have to be constructed for each of the λ^2 iterations.

3.5 Problem with decomposition of multicycle operations

[RJ94] considers breaking of long latency operations into multicycle operations. In the formulation of the lower bounding technique, the constraint for consecutive scheduling of decomposed multi-cycle operations is relaxed in addition to the relaxation of the precedence constraints. The problem associated with the former relaxation is illustrated in this section. Suppose there are n_k multicycle operations of type k and there is a resource constraint of

```
Procedure FormSubgraphs(G(V, E), s, t)
for i ← 1  to numofnodes do
    if s ≤ ASAP(v_i) < ALAP(v_i) ≤ t
        then S ← v_i
    else
        S_NG ← v_i
    endif
enddo

for i ← 1  to numofnodes do
    if i ∈ S
        then G_{s,t} ← v_i
    endif
    if i ∈ S_NG
        then if ASAP(v_i) < s
            then G_{<s} ← v_i
        endif
    endif
    if i ∈ S_NG
        then if ALAP(v_i) > t
            then G_{>t} ← i
        endif
    endif
enddo
```

Figure 3.2: Algorithm for finding subgraphs

```
Procedure RCSBound(λ_critical(G, s, t))
λ_max ← 0
for i ← 1  to λ_critical − 1 do
    for j ← i + 1  to λ_critical do
        LB_{i,j}(G) ← LowerBound(G_{i,j}) + λ_critical(G_{<i}) + λ_critical(G_{>j})
        if LB_{i,j}(G) > λ_max
            then λ_max ← LowerBound(G)
        endif
    enddo
enddo
LB(G) ← λ_max
```

Figure 3.3: Algorithm for finding lower bound

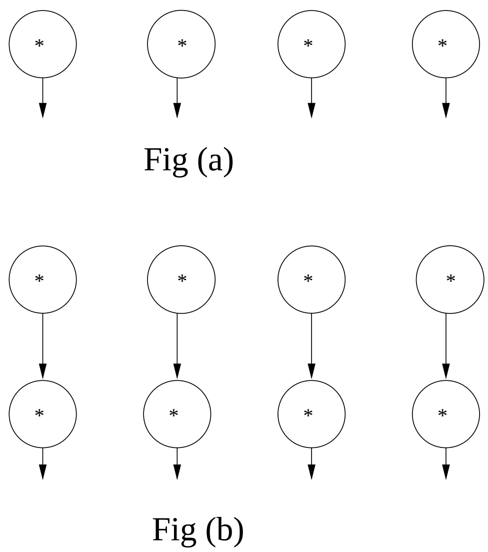

Figure 3.4: Problem with multicycle operations

r_k resource for the *type* k. Then, as shown in Figure 3.4, the following holds true.

$$\left\lceil \frac{n_k}{r_k} \right\rceil \cdot delay(k) \neq \left\lceil \frac{n_k \cdot delay(k)}{n_{r_k}} \right\rceil \tag{3.15}$$

To illustrate the above point, let $n_k = 4$, $r_k = 3$, let clock clk and the the delay $delay_k$ be such that $\left\lceil \frac{delay_k}{clk} \right\rceil = 2$, then the lower bound on the latency in term of *csteps* is $\left\lceil \frac{4}{3} \right\rceil * 2 = 4$. On the other hand, if each of the operations of the given type (multiplication in this case) is broken down such that each operation takes a single step to execute, then $n_k = 8$, $r_k = 3$, and $\left\lceil \frac{4*2}{3} \right\rceil * 1 = 3$.

3.6 Summary and Results

In this chapter we have shown that we can potentially obtain tighter RCS lower bounds by considering subgraphs of the original graph. The fact that the demand on functional units is dependent on the number of operations active concurrently is the motivating factor for considering subgraphs based on the ASAP and ALAP values. Any of the RCS lower bounding techniques can be used at the core of this method which are essentially used to compute the

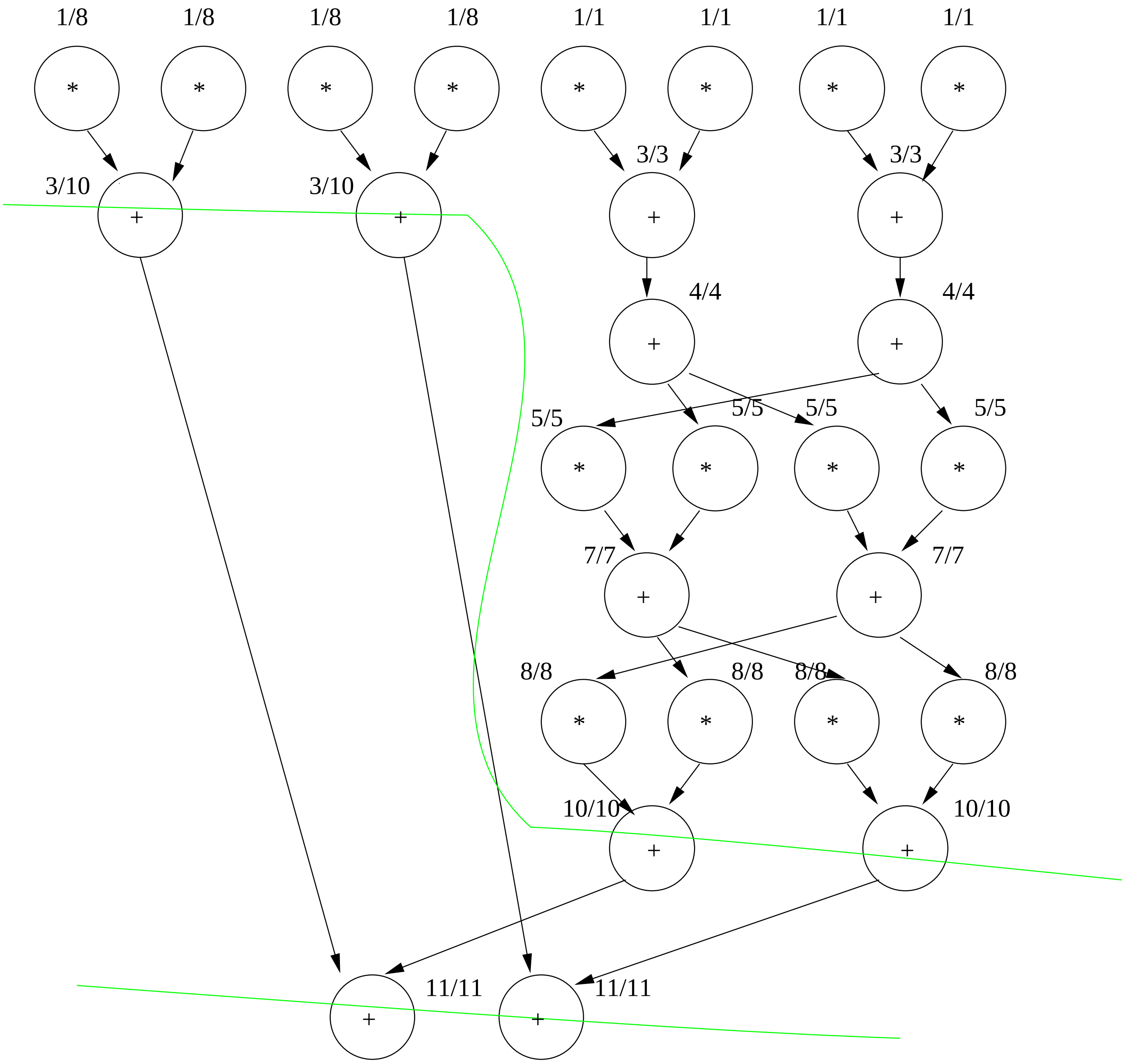

Figure 3.5: Splitting the graph

lower bounds in the constrained portion of the graph. The important contribution of the RCS lower bounding approach is that it is not only a new approach for computing lower bounds but also can be used in conjunction with other lower bounding methods to obtain tighter bounds on the latency. Hence, our approach is a general approach for RCS lower bounding. As an illustration of the formation of subgraphs in the lower bounding method, the Figure 3.5 shows the formation of subgraphs for s and t values corresponding to 1 and 9, respectively. Each pair of values labeled along the vertices depicts the ASAP/ALAP values. For this particular case as shown, the lower bound for $ARFILTER$ benchmark for a resource constraint of $\langle 2+, 3* \rangle$, according to the illustration in Section 3.3 was found to be 14 whereas using other lower bounding methods, the lower bound of 13 is obtained. Similarly, in the case of HAL benchmark for a resource constraint of $\langle 1+, 3* \rangle$, the lower bound is 7, there being an improvement of one time step. The above is for a case in which we use a multicycle multiplier (2 cycles) and a unicycle adder/ALU.

Chapter 4

TCS Upper Bounding Techniques

The Time Constrained Scheduling (TCS) problem is concerned with minimization of area (in particular, the number of functional units) of a design under a latency or time constraint. For the sake of simplicity, it is assumed that clock selection and module selection have already been done. The TCS problem can be solved as an Integer Linear Programming problem to obtain optimal solutions as discussed in the following section. To get an estimate of the resource utilization, the TCS problem can also be solved to obtain lower bounds on the number of resources and is called TCS Lower Bounding. On the other hand, the TCS upper bounding problem is solved to obtain upper bounds on the number of resources required. While existing techniques find good TCS lower bounds, there is a scope for improving the the TCS upper bounding techniques. The time constraint for scheduling a sequencing graph can be translated to a time constraint on the schedule interval for each operation in the graph. The ϕ_{ASAP} and the ϕ_{ALAP} are the unconstrained schedules for a sequencing graph. The schedule times of an operation v_i in the two schedules are referred to as $\mathsf{ASAP}(v_i)$ and $\mathsf{ALAP}(v_i)$, respectively, and the difference between the two values called *mobility*. The schedule interval of an operation is constrained by its *mobility*. If the given constraint $\lambda_{constraint}$ on the time to completion is less than the ASAP value for a particular graph, then the schedule is infeasible for that constraint. If $\lambda_{constraint} \geq \mathsf{ASAP}$, then there is a feasible schedule and the mobility of each operation will be increased by a value$\{\lambda_{constraint} - \mathsf{ASAP}\}$. The operations with the least or zero *mobility* are called *critical* operations in the graph. In this chapter, we start with the description of an ILP formulation for the TCS problem. Then some improvements over heuristic algorithms like *List Rescheduling* and *Force Directed Scheduling* for obtaining upper bounds comparable to the optimal resource requirement are discussed.

4.1 Previous work

4.1.1 List Scheduling Heuristic

A family of list scheduling algorithms are used to solve RCS or TCS problems quickly. These algorithms do not guarantee optimal solutions but nevertheless are very useful for a fast but reasonably precise design space exploration. Since the heuristic algorithms can be modeled and solved easily in contrast to the optimal solution methodology, these gain a lot of importance. All list scheduling algorithms are based on priority assignment to the various operations. To begin with, an example of a list scheduler for RCS problem is illustrated in Figure 4.1, followed by the TCS list scheduler in Figure 4.2 for which improvements are discussed in Section 4.2.

Consider a priority list of operations ranked according to their ALAP values. Then, as shown in the figure, at each step, called *cstep*, candidate operations $v^c_{i,j}$, the operations whose predecessors have been scheduled and can be initiated in the control step *cstep*, are found and among those operations the most *urgent* operation is scheduled to the current time step. At each time step, the resource constraint vector $r_{\vec{constraint}}$, should not be violated. The measure of *urgency* of an operation is determined from a *priority* list of operations. In this particular case, the candidate operation having the least ALAP value will have the highest priority. It is worth mentioning that the RCS upper bounds as computed by such a list scheduler are quite close to the optimal solutions. Also the quality of upper bounds is dependent upon a proper choice of priority function. For the same heuristic, if ALAP values under resource constraints are considered, the quality of the bounds obtained can be even better as shown in [Nar98]. A (forward) graph $G = (V, E)$ can also be reversed

Procedure RCSListScheduler
List $L \leftarrow \forall i = [1, N], j \in \mathcal{R}, v_{i,j}$ in increasing order of ALAP
while all $v_{i,j}$ not scheduled **do**
 $cstep \leftarrow 1$
 for each resource type $j \in \mathcal{R}$ **do**
  Find candidate (ready) operations $v_{i,j}^c$ in cstep
  Schedule most urgent $v_{i,j}^c$ according List L such that
  $r_{\vec{cstep}} \leq r_{\vec{constraint}}$
 enddo
 $cstep \leftarrow cstep + 1;$
enddo

Figure 4.1: Heuristic Algorithm for minimum latency under resource constraints

to obtain $G_R = (V, E)$ and scheduling performed on the reversed graph, under the same resource constraints to possibly obtain better solutions. The List Scheduling heuristic can also be implemented for TCS problems to minimize the resource utilization in a design and thus reduce the total area of the circuit. The heuristic illustrated in [De 94] considers a seed (initial) resource vector $r_{\vec{seed}}$ that is initialized to 1. The priority function used here is the *slack* or the time to the $\mathsf{ALAP}(v_i)$ of an operation v_i from the current step, *cstep*, under consideration in the scheduling process. Thus, the operation associated with a least value of slack in a priority list L of operations is the most *urgent*. For a given latency constraint $\lambda_{constraint}$, the ALAP values for each operation are obtained so that at each step in the scheduling process, the slack for each operation can be known. As the scheduling proceeds, the resource vector, $r_{\vec{seed}}$, is updated when a zero slack operation needs to be scheduled since this is reflective of the absolute deadline for an operation to be scheduled. Such a list scheduler generates upper bounds that increments the resource requirements aggressively and may generate a higher resource requirement than necessary. This is because it does not take into account the effect of increased resource availability on the previously scheduled operations. A few alternatives to this algorithm are discussed in the next section that try to incorporate the effect on the schedule due to the change in resource requirement at each

Procedure TCSListScheduler
$\vec{r} \leftarrow \vec{r_{seed}}$
List $L \leftarrow \forall_{i \in [1,N], j \in \mathcal{R}} \mathsf{ALAP}(v_{i,j})$
if $(\lambda_{constraint} < \lambda(\phi_{\mathsf{ASAP}}))$ **then**
    Schedule Infeasible
    Exit
endif
$cstep \leftarrow 1$
while $\forall_{v_{i,j}}$ not scheduled **do**
    for each resource type $j \in \mathcal{R}$ **do**
        Find candidate (ready) operations $v^c_{i,j}$
        $\forall_{i \in [1,N], j \in \mathcal{R}} slack(v_{i,j}) \leftarrow \mathsf{ALAP}(v_{i,j}) - cstep$
        Schedule all $v^c_{i,j}$ for which $slack(v_{i,j}) = 0$
        Update $\vec{r}$
        Schedule all $v_{i,j}$ that require no additional resources
    enddo
    $cstep \leftarrow cstep + 1$
enddo
return ($\vec{r}$, $cstep$)

Figure 4.2: Heuristic Algorithm for minimum area under latency constraints

update of the resource vector $\vec{r}$. The list scheduling heuristics as discussed above have a complexity of O(n) where n is the total number of operations in the data flow graph and each operation in the adjacency list has to be accessed once in order to schedule it to a particular time step. Since the computational complexity of implementing the algorithm is low, for large graphs, near exact solutions could be found easily.

4.1.2 Force Directed Scheduling

This is another heuristic for obtaining upper bounds for the TCS problem. The heuristic originally presented in [PK87] and [PK89] is a constructive algorithm that attempts to solve the TCS problem by uniformly distributing operations of different types in the schedule. The operations are, however, constrained by their time frames imposed by the latency constraint. The time frames of operations in the *HAL* graph for a latency constraint of 6 time steps is illustrated in Figure 4.3. This algorithm is also based on the idea that uniform

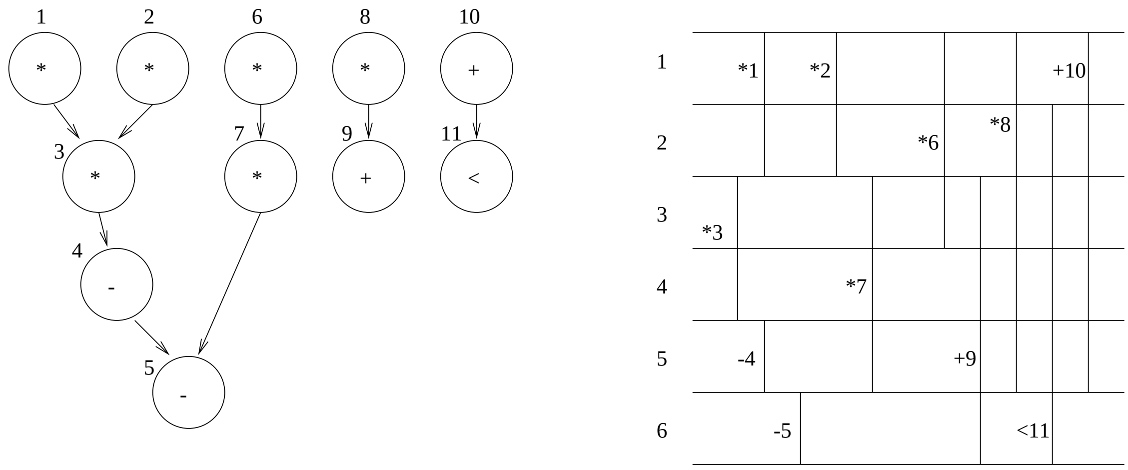

Figure 4.3: Time frames of operations for ASAP schedule

distribution of operations tries to minimize the maximum utilization of functional units for the design. In this section, existing forms of the algorithm are discussed and improvements are suggested in the later sections.

4.1.2.1 Original Algorithm

The Force Directed Scheduling algorithm, as shown in Figure 4.4, computes the probability distribution of operations of various types over the control steps in a schedule and then tries to balance the distribution over the control steps. For this, an initial probability distribution of operations is found according to the latency constraint specified. $\mathsf{TimeFrame}(v_i)$, the difference in the values $\mathsf{ALAP}(v_i)$ and $\mathsf{ASAP}(v_i)$ for the operation v_i is the interval in which it can be scheduled. Thus, the probability, $\sigma_{i,l}$, of an operation to be scheduled in a time step l, $\mathsf{ASAP}(v_i) \leq l \leq \mathsf{ALAP}(v_i)$, is given by,

$$\sigma_{i,l} = \frac{1}{\mathsf{TimeFrame}(v_i)} = \frac{1}{\mathsf{ALAP}(v_i) - \mathsf{ASAP}(v_i) + 1}. \tag{4.1}$$

The probability of all operations for all time steps are calculated in a similar fashion. Histograms for probability distribution for each resource type k for each control step l can be constructed. The histogram for each l shows the expected utilization of the resource type k in the that step. The height of the histogram for a particular l can be written as

$$|\mathsf{Histogram}_{k,l}| = \sum_{i \in k} \sigma_{i,l} \tag{4.2}$$

The maximum number of resources of type k required over all steps can be expressed as $\left\lceil \max \forall_{l=[1,\lambda]} |\mathsf{Histogram}_{k,l}| \right\rceil$. The objective of this heuristic is to reduce this value as far as possible in order to minimize the resource requirement.

Force Directed scheduling attempts to construct a schedule such that for each unscheduled operation v_i, the change in the height of the histograms is evaluated for each step $l \in \mathsf{TimeFrame}_i$, and the (l, v_i) combination that results in the smallest increase (or the greatest decrease) is chosen as the schedule step for that operation v_i. This results in the histograms being updated and scheduling continues until all operations are scheduled. The change in the height of the histograms for a particular step l is based on the notion of *self force* and *indirect force* of scheduling an operation to that step. The *self force* of an operation v_i, if scheduled in a step l such that $l \in \mathsf{TimeFrame}(v_i)$ is

$$\mathsf{SelfForce}_{v_i,k,l} = |\mathsf{Histogram}_{k,l}| - \sum_{s=\mathsf{ASAP}(v_i)}^{\mathsf{ALAP}(v_i)} \frac{|\mathsf{Histogram}_{k,s}|}{\mathsf{ALAP}(v_i) - \mathsf{ASAP}(v_i) + 1}. \tag{4.3}$$

The scheduling of any operation v_i to a step l affects the time frames of the *predecessors* and the *successors* and an indirect force that occurs on v_i due to these operations is also taken into account.

4.1.2.2 Improvement to Force Directed Scheduling

The original version of the algorithm eagerly schedules operations to a time step. This makes the algorithm much too greedy in terms of assigning operations to control steps. The improvement discussed in [VAKL91] tries to schedule operations through a gradual reduction in its time frame. Definite scheduling of operations to control steps is postponed to make the algorithm less greedy, expecting the algorithm to generate improved results.

```
Procedure ForceDirectedScheduler
For given λ Find φ_ASAP and φ_ALAP
Determine TimeFrame(v_{i,j}) ∀_i
Generate Histograms for each resource type k ∈ R
for each operation v_{i,j} do
    for each step l ∈ TimeFrame(v_{i,j}) do
        Compute increase △|Histogram_{j,l}| if v_{i,j} is scheduled in step l
        if △|Histogram_{j,l}| < △_min then
            △_min ← △|Histogram_{j,l}|
            v_{i,j} scheduled in step l
        endif
    enddo
    Modify Histograms
enddo
```

Figure 4.4: Original Force Directed Scheduling

4.1.3 Lower Bounds on the number of functional units

None of the heuristics discussed before base their upper bound estimation on the lower bounds on the functional units. The variations to the above algorithms that are considered later incorporate lower bounds as a seed or the starting point, so a basic method to obtain the lower bounds on the number of functional units is discussed. Lower bounds are essential means to prune the design space so that exploration need only be done for the useful regions. TCS lower bounds have been found to be quite accurate experimentally so this further motivates their use in improved upper bound estimation. The idea of TCS lower bounding, first discussed by Fernandez and Bussell [FB73] and subsequently by Sharma and Jain [SJ93], [OKD94], is discussed here in brief.

Given a Directed Acyclic graph $G = (V, E)$ and a latency λ_{given}, the lower bounds on each resource type can be calculated independently of the other resource types. Assume that there are a total of N operations in the graph and the number of operations of a particular type k be denoted by n_k. If there are R resource classes ($|\mathcal{R}|=R$) then the following holds true,

$$N = \sum_{k=1}^{R} n_k \tag{4.4}$$

Also assume that the delay of each operation of type k is $delay_k$. Given a time constraint, the possible time frames for the schedule time of the operations can be determined. For any interval $[s, t]$ such that $s < t$, consider an operation v_k of the type k which must be scheduled and completed in that interval. In follows that $\mathsf{ASAP}(v_i) \geq s$ and $\mathsf{ALAP}(v_i) + delay_k - 1 \leq t$. Let $n_k^{s,t}$ be the number of operations of type k that are guaranteed to be scheduled and completed in the interval $[s, t]$. Thus the minimum (or lower bound) on the number of resources of type k for the interval $[s, t]$ can be expressed as

$$\mathsf{TCSLB}_{s,t} = \left\lceil \frac{n_k^{s,t} * delay_k}{t - s + 1} \right\rceil. \tag{4.5}$$

The lower bound on the number of resources of type k for the graph $G(V, E)$ can be written as

$$\mathsf{TCSLB}_G = \max_{[s,t] \in [1, \lambda_{given}]} \left\{ \left\lceil \frac{n_k^{s,t} * delay_k}{t - s + 1} \right\rceil \right\}. \tag{4.6}$$

We can say that the above technique finds the maximum of the minimum value of the resource requirement over all intervals $[1, \lambda_{given}]$. The complexity of such an algorithm is thus $\mathrm{O}(\lambda^2)$ where λ is any specified latency for the design. Since the lower bound on each resource type can be obtained separately (independently), it is an absolute lower bound resource vector.

4.2 List Rescheduling

In this section improvements are suggested to the TCSListScheduler. Three possible variations to the algorithm are based upon

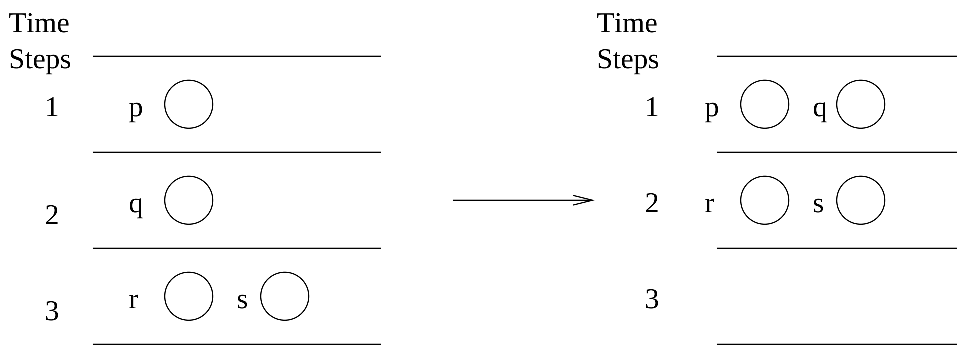

Figure 4.5: Effect of Rescheduling

- using the TCS lower bounds and
- *rescheduling* when $\vec{r}$ gets updated to improve the results on the TCS upper bounds.

4.2.1 Type 1 : Incorporating TCS Lower Bounds

The first variation to the TCSListScheduler is to change the $r_{\vec{seed}}$ value from 1 to $r_{\vec{tlb}}$ where *tlb* is referred to as trivial lower bounds for the TCS problem, which can be obtained by the method illustrated earlier. Thus, the only change in the algorithm will be with respect to the initialization:

$$r_{\vec{seed}} = r_{\vec{tlb}}. \tag{4.7}$$

This variation does not consider rescheduling but only reduces the search space for the scheduler.

4.2.2 Type 2 : Effect of TCS Lower Bounding and Rescheduling

4.2.2.1 Rescheduling

In this variation we consider *rescheduling*. One of the important drawbacks of the List scheduler in [De 94] is that it does not take into account the impact on the schedule due to the update in the resource vector, $\vec{r}$, at a particular instant of time, say *cstep*, during the scheduling process. To illustrate this point, consider Figure 4.5 and let $\vec{r} = 1$. Further, let

p, q and r be three operations of type k (such that there is no dependency between the three operations) in some dag $G = (V, E)$. Assume that these three operations are scheduled in the time steps *step* to $(step + 2)$ respectively. Now consider a fourth candidate operation s of the same type k, the slack for which is zero in time step $(step + 2)$. This operation would be forced to be scheduled in $(step + 2)$ and the resource vector would be updated such that the number of resources for the type k is now 2. Now the operations p and q that were forced to execute in consecutive time steps can now be scheduled in the same time step. Thus it is important to propagate the effect of update in $\vec{r}$ to the portion of the graph scheduled prior to this time step. The *rescheduling* can be implemented in two forms.

4.2.2.2 Resource vector initialization

1. Let the $r_{\vec{seed}} = 1$ and whenever $\vec{r}$ is updated, the earlier schedule is discarded and graph is rescheduled with the new $\vec{r}$ as the *seed* value.

2. Let the $r_{\vec{seed}} = r_{\vec{tlb}}$. Here again, each time $\vec{r}$ gets updated, rescheduling is performed.

In both the above variations, each time during the course of scheduling that a zero slack operation is found, the $\vec{r}$ is updated and scheduling is done afresh with the updated vector. This process proceeds till the sequencing graph is finally scheduled and can also be called an *Iterative Rescheduling* process. The above two variations differ only with respect to their initial *seed* value for the design space exploration. The second variation reduces the search space of the scheduler. The update of the resource vector in given time step, *cstep*, can be done in the following two ways.

- The slack of one or more operations of the same type k is zero, but only a unit increment in the availability of resources for type k is allowed.

- The slack of two or more operations of different types is zero, but only a unit increment in the availability of resources of *all* those types is allowed.

4.2.2.3 Updating the resource vector

1. Assume that for two successive iterations i and j of the scheduling process, the utilization is given by $\vec{r_i}$ and $\vec{r_j}$ respectively, and $j > i$, then the difference $\{\vec{r_j} - \vec{r_i}\} = \triangle\vec{r} = \hat{r}$, where $\hat{r}$ is the unit vector such that if $\vec{r_i} = \{r_1, r_2, \cdots, r_k\}$, then only one element r_i for some $i \in [1, k]$ in the vector is unity and the other elements are all 0.

2. For two successive iterations i and j, $j > i$, then the difference $\{\vec{r_j} - \vec{r_i}\} = \triangle\vec{r} = \vec{1}$ such that each element in the vector $\triangle\vec{r}$ is unity.

Each of two kinds of resource vector updating will affect the scheduling process in the way the design space is explored. The latter method of updating hastens the scheduling process if there are operations of different types having a zero slack in the same step, still maintaining the less aggressive nature by allowing for a unit increment in the resource availability of each resource type and rescheduling. Combining each of the two ways to initialize the resource vector and the two ways to update it, there can be four possible ways in which the design space can be explored for obtaining the TCS upper bounds. This is tabulated in the Table 4.1. The results for the list rescheduling algorithm for the various benchmarks have been tabulated in Table 4.2 through Table 4.5

4.3 Complexity of List Rescheduling

The list scheduling algorithms are based on a sorted list of operations called a priority list. The initial overhead for finding such a sorted list of n operations to be scheduled is $O(n \cdot logn)$. The complexity of list scheduling algorithm is simply $O(n)$. In our approach, the worst case bound on the number of reschedulings will be equal to the number of operations, n, if all operations are to be executed in only a single time step. In such a worst case scenario, the complexity of our approach will be $O(n^2)$. Except for such an extreme case, in a majority of the cases, the number of reschedulings will be $O(k \cdot n)$, where k is some constant factor.

Table 4.1: List Rescheduling Variations

$\delta(\vec{r})$	$\vec{r_{seed}}$
$\hat{r}$	$\vec{1}$
$\hat{r}$	$\vec{r_{tlb}}$
$\vec{1}$	$\vec{1}$
$\vec{1}$	$\vec{r_{tlb}}$

Table 4.2: List Rescheduling results for the HAL benchmark (Multicycle Multipliers)

	LowerBound		List		Optimal	
steps	*	ALU	*	ALU	*	ALU
13	1	1	1	1	1	1
12	1	1	2	1	2	1
11	2	1	2	1	2	1
10	2	1	2	1	2	1
9	2	1	2	1	2	1
8	2	1	2	1	2	1
7	2	1	2	2	2	2
6	3	1	3	2	3	2

4.4 Improvements to Force Directed Scheduling

4.4.1 Choosing the step with maximum area pressure

Since the main feature of the algorithm is uniform distribution of operations over the schedule, the time steps that are *crowded* should be selected so that the pressure on resource utilization can be reduced. Realizing the importance of incorporating lower bounds in heuristic solutions for the TCS problems, the resource vector $\vec{r}$ is initialized to $\vec{r_{tlb}}$. The *histograms* for each resource type k are constructed and the value computed by

$$\max\{\forall_{k\in\mathcal{R},l\in[1,\lambda]}\left\{(|\mathsf{Histogram}_{k,l}| - \mathsf{LB}_k) * Area_k\right\}\} \tag{4.8}$$

is the increment $\triangle Area$ over the lower bound on area. The time step l for which the increase in area is maximum (or the maximum positive deviation from the lower bound

Table 4.3: List Rescheduling results for the EWF benchmark (Multicycle Multipliers)

	LowerBound		List		Optimal	
steps	*	ALU	*	ALU	*	ALU
28	1	1	1	1	1	1
27	1	1	1	2	1	2
26	1	2	1	2	1	2
25	1	2	1	2	1	2
24	1	2	2	2	1	2
23	1	2	2	2	1	2
22	1	2	2	2	1	2
21	1	2	2	2	1	2
20	1	2	2	2	2	2
19	1	2	2	2	2	2
18	2	2	3	3	2	2
17	3	3	3	3	3	3

area) is chosen as the critical time step l_{crit} to move out operations of type k such that the $\triangle Area \to \epsilon$, such that $\epsilon \to 0$. The next step is to select a *candidate* operation of the type k_{crit} which would reduce the value $\triangle Area$ by the maximum possible amount.

4.4.2 Ordering of operations

The original force directed scheduling algorithm does not define a proper order of selecting candidate operations to schedule, in a given scheduling iteration. The order in which the various operations are chosen will have an impact on the final schedule. In this improvement, we consider scheduling based upon dynamically selecting operations to schedule at a given instant of time in the scheduling process. Earlier we selected the particular resource class from which to choose operations from. Even after selecting a critical resource class k_{crit} in the time step l_{crit} according to the algorithm illustrated in Figure 4.6, there could be candidate operations $v_{i,k}$ such that $l_{crit} \in \mathsf{TimeFrame}(v_{i,k})$. A list L_k of all such operations of the selected type k_{crit} is constructed. Amongst the operations in L_k based on ALAP[1]

[1] Minimum ALAP is chosen in general but in two cases for EWF, maximum ALAP produced better results.

Table 4.4: List Rescheduling results for the *ARF* benchmark (Multicycle Multipliers)

	LowerBound		List		Optimal	
steps	*	ALU	*	ALU	*	ALU
34	1	1	1	1	1	1
33	2	1	2	1	2	1
32	2	1	2	1	2	1
31	2	1	2	1	2	1
30	2	1	2	1	2	1
29	2	1	2	1	2	1
28	2	1	2	1	2	1
27	2	1	2	1	2	1
26	2	1	2	1	2	1
25	2	1	2	1	2	1
24	2	1	2	1	2	1
23	2	1	2	1	2	1
22	2	1	2	1	2	1
21	2	1	2	1	2	1
20	2	1	2	1	2	1
19	2	1	2	1	2	1
18	2	1	2	1	2	1
17	2	1	3	1	3	1
16	3	1	3	1	3	1
15	3	1	3	2	3	2
14	3	1	4	2	4	2
13	4	2	4	2	4	2
12	4	2	4	2	4	2
11	4	2	4	2	4	2

Table 4.5: List Rescheduling results for the *FDCT* benchmark (Multicycle Multipliers)

	LowerBound		List		Optimal	
steps	*	ALU	*	ALU	*	ALU
34	1	1	2	1	1	1
33	2	1	2	1	2	1
32	2	1	2	1	2	1
31	2	1	2	1	2	1
30	2	1	2	1	2	1
29	2	1	2	1	2	1
28	2	1	2	1	2	1
27	2	1	2	1	2	1
26	2	1	3	1	2	1
25	2	2	2	2	2	2
24	2	2	2	2	2	2
23	2	2	2	2	2	2
22	2	2	2	2	2	2
21	2	2	3	2	2	2
20	2	2	3	2	2	2
19	2	2	3	2	2	2
18	2	2	3	2	2	2
17	3	2	4	2	3	2
16	3	2	4	2	3	2
15	3	2	4	2	3	2
14	3	2	5	2	3	2
13	3	2	4	3	-	-
12	4	3	4	3	4	3
11	4	3	5	3	4	3
10	5	3	7	3	5	3
9	6	3	10	4	8	3
8	8	4	8	4	8	4

Procedure $\mathsf{FindCriticalStepAndType}$
Find Histograms based on λ
$maxarea \leftarrow 0$
$\triangle Area_k \leftarrow 0$
for each resource type $k \in \mathcal{R}$ **do**
  for each $l \in [1, \lambda]$ **do**
    $\triangle Area_k \leftarrow (|\mathsf{Histogram}_{k,l}| - \mathsf{LB}_k) * area_k$
    if $\triangle Area_k > maxarea$ **then**
      $maxarea \leftarrow \triangle Area_k$
      $k_{crit} \leftarrow k$
      $l_{crit} \leftarrow l$
    endif
  enddo
enddo

Figure 4.6: Algorithm for selecting step with maximum pressure and the resource type causing the pressure

as the priority function. If for an operation $v_{i,k}$, the $\mathsf{TimeFrame}(v_{i,k}) = l_{crit}$, such that its probability distribution $\sigma(v_{i,k}) = 1$, the operation $v_{i,k}$ is said to have been scheduled and cannot be included in the list L_k. The algorithm for selection of a single candidate operation from the list L_k is shown in in Figure 4.7.

4.4.3 Splitting Time frames: Gradual Time frame reduction

As shown in [VAKL91], delaying the actual assignment of an operation to a schedule step tends to make the scheduler less greedy and improves the upper bound estimates. This is achieved by gradually shrinking the time frames of the operations till the time frame reduces to a single time step and can be shrunk no more. In this approach, however, the time frame reduction occurs by either increasing the ASAP or reducing the ALAP of an operation by a unit time step. We generalize this concept further by allowing the time frame of an operation to be a union of disjoint time frames. This is achieved as follows: For a candidate operation $v^c_{i,k}$ selected previously, the time step $l \in \mathsf{Timeframe}(v^c_{i,k})$ for which

Procedure $\mathsf{FindCandidateOperation}$
$k \leftarrow k_{crit}$
$l \leftarrow l_{crit}$
for each $v_{i,k}$ such that $l \in \mathsf{TimeFrame}(v_{i,k})$ **do**
  if $\sigma(v_{i,k})$ in time step $l \neq 1$ **then**
    $L_k \leftarrow v_{i,k}$
  endif
enddo
Sort L_k according to ALAP
return $L_k[0]$

Figure 4.7: Algorithm for finding Candidate Operation

Procedure $\mathsf{SplitTimeFrames}$
$v^c_{i,k} \leftarrow \mathsf{FindCandidateOperation}$
$maxforce \leftarrow 0$
for each $l \in \mathsf{TimeFrame}(v^c_{i,k})$
$\mathsf{TotalForce}(l) \leftarrow \mathsf{SelfForce}(l) + \mathsf{IndirectForce}(l)$
if $maxforce > \mathsf{TotalForce}(l)$ **then**
  $maxforce \leftarrow \mathsf{TotalForce}(l)$
  $l^c \leftarrow l$
endif
enddo
$\sigma(v_{i,k})$ in step $l^c \leftarrow 0$

Figure 4.8: Algorithm for gradual time frame reduction

the *total force* is maximum, its probability of occurrence $\sigma(v_{i,k})$ in l is 0. The algorithm for time frame reduction is shown in Figure 4.8.

4.5 Summary

In this chapter we incorporate the TCS lower bounds and use the concept of Iterative Rescheduling to obtain an improvement in the solutions for TCS upper bounds. Using this approach, optimal solutions were obtained for all time constraints in the case of *HAL* and *ARFILTER* benchmarks as shown in Table 4.2 and Table 4.4 respectively. The solutions for *EWF* benchmark with respect to the optimal solutions were off by one resource unit each

Table 4.6: FDS results for the *HAL* benchmark (Multicycle Multipliers)

	OurFDS		Optimal	
steps	*	ALU	*	ALU
13	1	1	1	1
12	2	1	2	1
11	2	1	2	1
10	2	1	2	1
9	2	1	2	1
8	2	1	2	1
7	2	2	2	2
6	3	2	3	2

of ALU and the multiplier for a time constraint of 18 steps, as can be seen from the results tabulated in Table 4.3. Also for the same benchmark, for time constraint of 19 through 24, an additional multiplier was required for scheduling using list rescheduling.

For the improvements in the Force Directed Scheduling algorithm, a dynamic selection of the operations was performed to reduce the time frames of the operations gradually. Also the time frames of the operations were allowed to be a collection of disjoint sets of time steps rather than a continuous set of time steps. The improved force directed scheduling algorithm was found to be computationally quite expensive. It generated optimal solutions for all time constraints for small sized graphs like *HAL* and *EWF* as shown in Table 4.6 and Table 4.7 respectively. The results for the *EWF* benchmark were compared[2] with those cited in [PK89] and [VAKL91]. For the *ARFILTER* benchmark, optimal solutions were obtained for time constraint of 11 through 15 steps and from 21 through 34 steps. For the remaining time constraints, the results were off with respect to the optimal solutions, as can be seen in Table 4.8.

[2] The blank spaces in Table 4.7 indicate that no results were cited for those particular cases in the related references.

Table 4.7: FDS results for the EWF benchmark (Multicycle Multipliers)

	OurFDS		Optimal		Original FDS		Eindhoven	
steps	*	ALU	*	ALU	*	ALU	*	ALU
17	3	3	3	3	3	3	3	3
18	2	2	2	2	2	3	2	2
19	2	2	2	2	2	2	2	2
20	2	2	2	2	-	-	-	-
21	1	2	1	2	1	2	1	2
22	1	2	1	2	-	-	-	-
23	1	2	1	2	-	-	-	-
24	1	2	1	2	-	-	-	-
25	1	2	1	2	-	-	-	-
26	1	2	1	2	-	-	-	-
27	1	2	1	2	-	-	-	-
28	1	1	1	1	-	-	-	-

Table 4.8: FDS results for the *ARF* benchmark (Multicycle Multipliers)

	OurFDS		Optimal	
steps	*	ALU	*	ALU
34	2	1	2	1
33	2	1	2	1
32	3	1	2	1
31	3	1	2	1
30	2	1	2	1
29	2	1	2	1
28	2	1	2	1
27	2	1	2	1
26	2	1	2	1
25	2	1	2	1
24	2	1	2	1
23	2	1	2	1
22	2	1	2	1
21	2	1	2	1
20	3	1	2	1
19	2	2	2	1
18	3	1	2	1
17	4	2	3	1
16	3	2	3	1
15	3	2	3	2
14	4	2	4	2
13	4	2	4	2
12	4	2	4	2
11	4	2	4	2

Table 4.9: FDS results for the *FDCT* benchmark (Multicycle Multipliers)

	OurFDS		Optimal	
steps	*	ALU	*	ALU
34	2	2	1	1
33	2	1	2	1
32	2	1	2	1
31	2	1	2	1
30	2	1	2	1
29	2	1	2	1
28	2	1	2	1
27	2	2	2	1
26	2	2	2	1
25	2	2	2	2
24	2	2	2	2
23	2	2	2	2
22	2	2	2	2
21	2	2	2	2
20	2	2	2	2
19	3	3	2	2
18	3	3	2	2
17	3	3	3	2
16	3	3	3	2
15	3	3	3	2
14	4	3	3	2
13	4	3	-	-
12	4	4	4	3
11	5	4	4	3
10	5	4	5	3
9	8	4	8	3
8	8	4	8	3

Chapter 5

Clock Selection

There are three major constraints in the scheduling problem in high level synthesis of VLSI circuits, namely, resource, latency and clock. Scheduling in the presence of resource constraints is called Resource Constrained Scheduling (RCS) which basically tries to minimize the latency of the schedule. On the other hand, Time Constrained Scheduling (TCS) tries to minimize the area of the design or in particular tries to minimize the number of functional units used in the design. In both these approaches to scheduling, the clock selection is assumed to have been done *a priori*. The use of TCS and/or RCS in scheduling is a 2-dimensional approach for exploration. As stated in [BW96], this method would be useful for exploring only a small portion of a complete design space. For a more useful and exhaustive search, the clock value is also considered to be a constraint for the scheduling problem and this adds the third dimension to the search space. Although a value of the clock is required prior to scheduling, the best value of the clock can be found only after evaluating different schedules. This interdependence makes it difficult to select a good value of clock prior to scheduling. A variety of schedules can be found, each for a different clock value and this provides the designer with greater flexibility to choose his design. Since the clock value has to be known before the scheduling process itself, a good estimation of the clock value is essential. An optimal value can be found by finding all possible schedules with different candidate clock lengths exhaustively and then the best (or optimal) value can be chosen, but this would take a long time and hence it is highly undesirable. It is

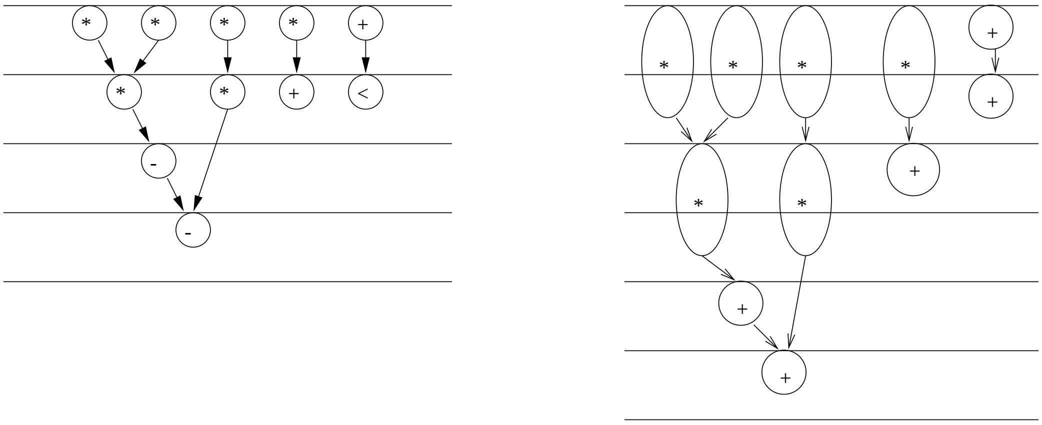

Figure 5.1: Effect of clock length on scheduling the *HAL* graph

therefore necessary to come up with algorithms that are fast and generate a near optimal value of the clock length if not the optimal value. In this chapter we discuss briefly some of the previous work on the clock selection problem as discussed in [BW96] and [CBW97]. We then describe our method of finding the clock length as improvement over [BW96], [Kri97] and illustrate the differences with respect to that method. Our approach finds the clock length independent of scheduling and hence it is not computationally expensive, in that respect it is similar to [BW96]. Also as is evident later, the technique generates almost as good results as compared to an exhaustive, computationally more expensive method in [CBW97].

5.1 Effect of clock on the schedule length and idle time of operations

The choice of a clock pulse affects the scheduling of a sequencing graph. We assume that all arithmetic operations can be performed on the ALU and the multiplication operations on the multiplier, then for the HAL sequencing graph, we can have two different schedule lengths depending on the value of the clock used and the delay of the different functional units. If the delay of the ALU is $25ns$ and the delay of the multiplier is $50ns$ and a clock of $50ns$ is used to schedule the graph (assuming no resource constraints), then we can complete the schedule in $200ns$. This is illustrated in the figure on the left in Figure 5.1. In this particular case, the ALU is idle for $25ns$ per clock. On the other hand, if we use a clock of $25ns$, then the schedule time is $150ns$, and in this case, none of the resources is idle as shown in the figure on the right in Figure 5.1. It is thus important to choose a proper value of the clock to reduce the schedule length and the idle time on the various resources.

5.2 Background

We assume that module selection has already been done and hence delays for the functional units have already been provided from a particular library. We denote a functional unit by FU_i and the delay by $delay_i$. Since a clock length has to be chosen before scheduling, some of the trivial choices a designer may make are (i) the delay of one of the functional units, (ii) the delay of the slowest/fastest functional unit, and (iii) the GCD of the delays of all functional units. All of these plausible options have one point in common, i.e, they try to zero out the idle time on at least one of the functional units, but it does not necessarily guarantee a good clock. The selection of the clock length not only affects the schedule, but it also affects the hardware size and the complexity of the controller. Amongst the three methods cited above, the last one might generate a clock length which is so small that it may not be practically feasible to implement the control circuit. We define the *slack* of an operation mapped to a functional unit FU_i as the time during which the functional unit is idle. To illustrate this point assume that the clock length is clk, then the slack for a functional unit FU_i is defined as the following

$$slack_i = clk \cdot \left\lceil \frac{delay_i}{clk} \right\rceil - delay_i. \tag{5.1}$$

The slack is the time for which the FU_i is not going to do any useful work. The utilization of the functional units can be made more efficient by decreasing the value of the slack and also possibly reducing the schedule length. We need to evaluate a set of candidate clock lengths and then choose the one for which the slack is minimum. Ideally, we would like to have zero slack for all functional units but since this is not practically possible, we try to look at possible clock values for which the slack would be minimum. The slack on a functional unit FU_i would be zero if the value of the clock perfectly divides its delay. This particular value of the clock need not necessarily reduce the slack on some other functional unit FU_k. It is thus important to find a value of the clock that would reduce the slack on all the functional units. Although such a value might be difficult to find, a measure that we would like to consider is an average value of slack over all functional units as illustrated in [CGN96] and then obtain a value for the clock length based on the minimum average slack. The average slack function is defined as

$$slack_{avg}(clk) = \frac{\sum_j \{num(v_{i,j}) \cdot slack_j(clk)\}}{N} \tag{5.2}$$

where $num(v_{i,j})$ is the number of operations of type j, $slack_j(clk)$ is the slack on that operation for a clock clk and N is the total number of operations in the DFG. The objective is to minimize the above slack function with respect to the value of the clock. All the points of local minima for the above average slack function are the possible values of the clock cycle in the search space. The average slack is evaluated for all these points and the clock length value corresponding to the minimum average slack can then be chosen as the clock value. This approach does not consider any constraints either on the availability of resources or latency. It also does not take the structure of the graph into account.

Another approach to finding the optimal value of the clock is described in [BW96] and [CBW97]. The optimal clock selection technique as described in [BW96] considers all possible integral clock lengths which are ceilings of the integer divisors of the delays of the different functional units used in the design. This can be expressed as

$$\mathsf{C} = \left\{ clk | clk = \forall i, k \left\lceil \frac{delay_i}{k} \right\rceil \right\} \tag{5.3}$$

subject to the condition $clkmin \leq clk \leq \max(\forall_i(delay_i))$, where $clkmin$ is a technological limitation, a clock cycle lesser than this value is not practically feasible. The set C for the candidate clock lengths can be pruned further the size of the set C can be reduced further by considering the slack on the various functional units for a particular value of the clock. A slack vector $sl\vec{a}ck(clk)$ that defines the slack for m functional units types for a particular value of clock clk can be written as

$$sl\vec{a}ck(clk) = \{slack_1, slack_2, ..., slack_m\}\,. \tag{5.4}$$

If $sl\vec{a}ck(clk) \leq sl\vec{a}ck(clk')$, then the clk' can be replaced by clk. Hence, the reduction in the size of the initial set C to C'(say). For each $clk \in \mathsf{C}'$, the resultant set of clock values, the exact solutions for RCS (TCS) problem is found for various resource vectors and then the clk corresponding to the minimum schedule length (minimum resource requirement) is chosen as the optimal value of the clock.

5.3 The Clock Selection Technique

5.3.1 The critical paths

The Slack Minimization Algorithm in [CGN96] considers a weighted average of slack for all operations in the graph. It does not take into account the structure of the graph. The critical path in the sequencing graph carries the essential dependency information and is the longest path in the graph. We would like to reduce the slack of the operation that has a higher frequency of occurrence in the critical path since reducing the delay in the critical path would improve the completion time of the schedule. The frequency of occurrences of operations belonging to different resource classes is considered and the slack of that particular functional unit is weighted with the frequency of occurrence of that particular resource class. All possible paths formed by the critical nodes in the data flow graph are evaluated. The slack function is written as

$$Slack_{weighted}(clk) = \sum_{j} \left\{ freq_j^{critical} \cdot slack_j \right\} \tag{5.5}$$

where the term $freq_j^{critical}$ is the average of the fraction of the occurrence of operations belonging a particular resource class j over all critical paths in the data flow graph.

$$freq_j^{critical} = \left\{ \frac{\sum_{allP_c} \left\{ \frac{\sum_j (num(v_{i,j}))}{N_{c,p}} \right\}}{P_c} \right\} \tag{5.6}$$

where P_c is the total number of critical paths in the data flow graph and $N_{c,p}$ is the total number of nodes on the pth critical path.

5.3.2 Generating candidate clock lengths

If the expression for slack is substituted in the above equation, then we have the following expression

$$Slack_{weighted}(clk) = \sum_{j} \left\{ (freq_j^{critical}) \cdot ((clk \cdot \left\lceil \frac{delay_j}{clk} \right\rceil) - delay_j) \right\}. \tag{5.7}$$

In the rest of this chapter, we would refer to the above critical path weighted average slack as CPWslack. Our aim is to minimize the weighted slack function above and find the corresponding value of clock clk for which the function is minimum. If we consider a single term of the equation above as a function of the clock clk, then

$$f_j(clk) = K * \left\{ (clk \cdot \left\lceil \frac{delay_j}{clk} \right\rceil) - delay_j \right\}. \tag{5.8}$$

We know that the greatest integer function is discontinuous at the integer points. Hence the points at which the above function is discontinuous is governed by the term $\left\lceil \frac{delay_j}{clk} \right\rceil$. The discontinuous points are $\frac{delay_j}{k}$, where k is an integer. These are the points for which

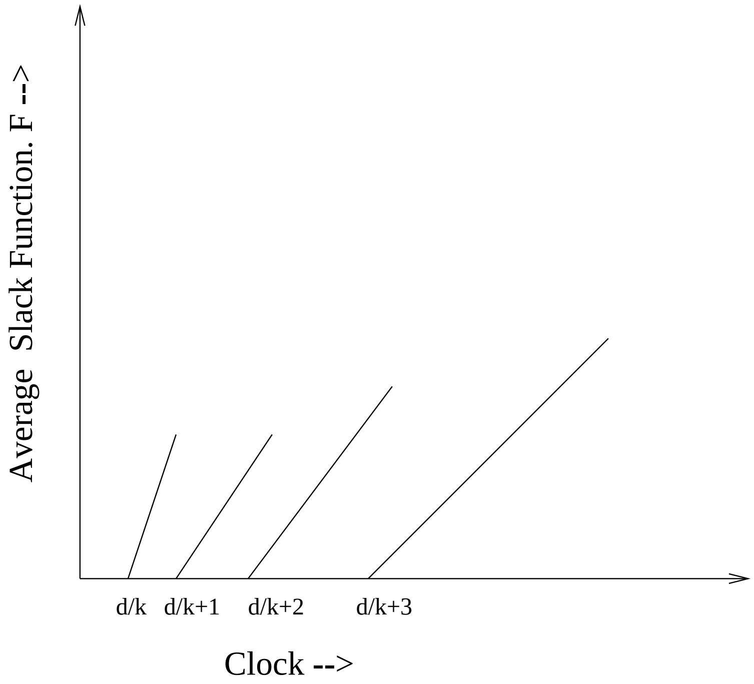

Figure 5.2: Discontinuous points

the slack function would reach a minimum value for a particular clock. All these points belong to the set of candidate clock lengths C. The function will be continuous between the points $clk = (\frac{delay_j}{k}, \frac{delay_j}{k-1})$. Consider some arbitrary value for $clk = \frac{delay_j}{k_r}$, where $k_r \in Z$ where Z is the set of reals and $k - 1 < k_r < k$. Then slack function is a straight line between the points $\frac{delay_j}{k}$ and $\frac{delay_j}{k-1}$ and the slope of the line is $\frac{\lceil k_r \rceil}{k_r}$. A plot of the function is as shown in the Figure 5.2. The algorithm for generating the candidate clock lengths is shown in Figure 5.4.

5.3.3 The Activity Factor

For all possible $clk \in \mathsf{C}$, the value of the critical path weighted slack function is evaluated. This value of slack reflects the performance of the clock in absolute terms only. The absolute value of the slack does not convey the fraction of the useful clock. The major objective of using slack for clock selection is not only to select a clock which causes the least slack but also the slack which is relatively very less in comparison to the value of the clock. For this purpose, we define a new figure of merit called the *activity factor* to chose the best possible clock. Such a value of clock would reflect the maximum amount of useful work that can be done.

We define the activity factor in general as the following

$$\alpha(clk) = \left\{1 - \frac{slack}{clk}\right\}. \tag{5.9}$$

In our definition of the slack, we will be using critical path weighted average slack, denoted by $\mathsf{CPWslack}(clk)$. The various candidate clocks can be pruned and the best clock selected using the activity factor, α. As an illustration, consider two clock values clk_1 and clk_2, such that $clk_1 > clk_2$. Let the critical path weighted average slacks for the two values of clocks be $\mathsf{CPWslack}(clk_1)$ and $\mathsf{CPWslack}(\mathsf{clk}_1)$, such that $\mathsf{CPWslack}(clk_1) = \mathsf{CPWslack}(clk_2)$. It is evident that the fraction, $\alpha(clk_1) > \alpha(clk_2)$, and hence clk_1 is a better value of clock that clk_2 because it reflects a better utilization of the available clock. Another important aspect is that for all values of the clock which result in an equal value of CPWslack, the highest value of the clock is always preferred. A higher value of the clock reduces hardware complexity and associated power dissipation in the circuit. The algorithm for selection of the near-optimal clock selection based on activity factor is illustrated in Figure 5.5. The clock length causing the highest value of α is selected as the optimal clock.

The above is a scalar comparison of the activity factor values for different clocks. This is in contrast to the vector comparison over slacks for different types of resources for various clocks as discussed in [BW96].

5.4 Complexity of the algorithm

The complexity for the ASAP and ALAP schedules is $\mathrm{O}(V + E)$. Assuming that n is the number of candidate clock lengths, complexity of evaluating the activity factor is $\mathrm{O}(n)$. The

Procedure $\mathsf{FindCriticalNodes}(G(V, E))$
for $i \leftarrow 1$ **to** *num of nodes* **do**
  find $\mathsf{ASAP}(v_{i,j})$ and $\mathsf{ALAP}(v_{i,j})$
  if $(\mathsf{ASAP}(v_{i,j}) = \mathsf{ALAP}(v_{i,j}))$ **then**
    $S_{critical} \leftarrow v_{i,j}$
    $Num_j^{critical} \leftarrow Num_j^{critical} + 1$
  endif
enddo

Figure 5.3: Algorithm for finding Nodes on the Critical Path

Procedure $\mathsf{FindCandidateClocks}$
$\mathsf{TYPE} \leftarrow$ *num of resource classes*
for i $\leftarrow 1$ **to** TYPE **do**
  $j \leftarrow 1$
  flag $\leftarrow true$
  while (flag) **do**
    $clk \leftarrow \frac{delay_i}{j}$
    if $(clkmin < clk \leq delay_i)$ **then**
      $\mathsf{C} \leftarrow clk$
      $j \leftarrow j + 1$
    else
      flag $\leftarrow false$
    endif
  enddo
enddo
return C

Figure 5.4: Algorithm for finding candidate clock lengths

Procedure FindNearOptClock
$\alpha_{max} \leftarrow 0.0$
$optclk \leftarrow 1.0$
for all $clk \in \mathsf{C}$ **do**
  $\alpha(clk) \leftarrow \left\{1 - \frac{\mathsf{CPWslack}(clk)}{clk}\right\}$
  if $(\alpha(clk) > \alpha_{max})$ **then**
    $\alpha_{max} \leftarrow \alpha(clk)$
    $optclk \leftarrow clk$
  endif
enddo

Figure 5.5: Algorithm for finding a near optimal clock

number of points (clock values) searched is equal to

$$n = \sum_i \left\{ \left\lfloor \frac{delay_i}{minclock} \right\rfloor \right\} + 1 \tag{5.10}$$

where, *minclock* is the minimum value of the clock length feasible.

The critical paths in the unconstrained scheduled graph or the sequencing graph are the longest paths from the source to the sink. The problem of finding all the critical paths thus reduces to a problem of finding single source all possible longest paths in the original graph. However, in our approach, we consider only the edge disjoint paths formed by the critical nodes in the graph. The maximum number of such edge disjoint paths possible will be given by $|E|$ where E is the set of edges in the subgraph formed by the critical nodes. Thus, the complexity of obtaining the distribution of operations of different types in the critical paths is $O(E)$.

5.5 Chaining

The value of the clock can also be affected by scheduling operations (having a precedence constraint between them) in the same step. This is more commonly called *chaining* which tries to minimize the slack by scheduling an operation during the the idle time in a clock cycle. We consider two basic types of chaining as shown in the Figure 5.6 and follow a similar approach of determining candidate clock values as in the non-chained case. Also for simplicity, we consider only two kinds of chains, viz., a multiplier can be chained to an adder or an adder can be chained to an adder. The chained pair of operations is considered to be a *new type* of operation. For every type of chaining that we consider, the candidate clock lengths are chosen such that the slack on the chained nodes is zero.

- For the Type 1 chaining, the chained operations must execute in a single step. Then the set of integral divisors of the delay for the longest chain is considered and the integers for which the type 1 chaining is not possible for the type of chains we consider are dropped from the set. The average slack over all operations in the critical path is computed.

- For Type 2 chaining, the slower functional unit can be multicycled as shown in Figure 5.6. Here we candidate clock lengths are the integral divisors of the longest possible chain under the constraint that the smallest possible clock value has to be greater than the delay of the slowest functional unit. Also type 2 chaining cannot be possible for the longest chain delay since it reduces to a type 1 chain. Average slack function is computed over all operations in the critical path.

The results for chaining, i.e, the activity factor for the various candidate clock lengths for the two types of chains mentioned above have been tabulated in Table 5.1 through Table 5.4 for the *ARFILTER* and *EWF* benchmarks. It can be noted that smaller clock lengths correspond to a higher value of the activity factor.

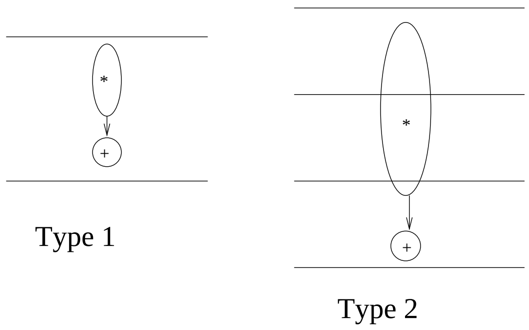

Figure 5.6: Types of chaining

Table 5.1: Results for Type 1 chaining for *ARFILTER* benchmark

clk	*	+	-	{*,+}	{+,+}	CPWSlack	α
211	48	163	155	0	N/A	65.2	0.69
96	29	48	40	N/A	0	20.55	**0.79**

Table 5.2: Results for Type 2 chaining for *ARFILTER* benchmark

clk	*	+	-	{*,+}	CPWSlack	α
106	49	58	50	1	23.2	0.78
71	50	23	15	2	9.2	0.87
53	49	5	50	1	2.0	**0.96**

Table 5.3: Results for Type 1 chaining for *EWF* benchmark

clk	*	+	-	{*,+}	{+,+}	CPWSlack	α
211	48	163	155	0	N/A	118.81	0.44
96	29	48	40	N/A	0	15.0	**0.84**

Table 5.4: Results for Type 2 chaining for EWF benchmark

clk	*	+	-	{*,+}	CPWSlack	α
106	49	58	50	1	42.45	0.60
71	50	23	15	2	17.27	0.76
53	49	5	50	1	3.9	**0.93**

5.6 Discussion

For each value of the candidate clock lengths, the heuristic computes the weighted average slack function (abbreviated as CPWslack) and the activity factor, (α_{CPW}). The performance of the heuristic is evaluated for various benchmarks and the results are tabulated in Table 5.9 through Table 5.18. For a comparison with the approach in [CGN96], the values of the average slack over all operations abbreviated as CGslack and the corresponding α_{CG} have also been tabulated. In this thesis, the slack on the operations is weighted by the frequency of operations in the critical path. Since the schedule length of a sequencing graph is dependent on the length of the critical path, the algorithm tries to minimize the slack on the operations in the critical path by zeroing out the slack on the most frequent operation. It assumes the availability of unlimited number of resources. This is in contrast to the approach in [BW96] wherein the clock selection is based on exact scheduling under constraints. Our approach tries to extract some essential information from the structure of the DFG by emphasizing on the operations on the critical paths. A sample critical path in $ARFILTER$ is shown in figure 5.7. Also the optimal solutions have been evaluated for some typical resource vectors for two benchmarks $ARFILTER$ and EWF as shown in Table 5.5 and Table 5.6. It can be noted that an optimal value of the clock for the least schedule length depends on the resource constraint and if sufficient resources are available, then the optimal value of the clock would approach the value computed in our approach as the best value of the clock computed by the heuristic assumes unlimited resources. To illustrate this point, consider

Table 5.5: *RCS* results without chaining for *EWF*

	1*,2+	2*,2+	3*,3+
clk	*ns* (steps)	*ns* (steps)	*ns* (steps)
163	2608 (16)	2608 (16)	2282 (14)
82	1804 (22)	1476 (18)	1476 (18)
55	1650 (30)	1210 (22)	1210 (22)
48	1824 (38)	1248 (26)	1248 (26)
41	1804 (44)	1476 (36)	1476 (36)
33	1716 (52)	1320 (40)	1320 (40)
28	1680 (60)	1232 (44)	1232 (44)
24	**1632 (68)**	**1152 (48)**	**1152 (48)**
21	1722 (82)	1302 (62)	1302 (62)

the exact RCS solutions in Table 5.5 and the heuristic solutions for integral clocks for the critical path weighted case in Table 5.14. The best value for *clk* found by the heuristic is $24ns$ (for maximum value of α_{CPW}). It can be seen from exact RCS solutions, in most cases, the same value of $24ns$ is the optimal result. The next best (or comparable) value of the clock in terms of the total schedule length (in ns) according to the heuristic is $55ns$ which is also the next best value as evaluated by exact RCS solutions. In contrast to this, if we consider the values for α_{CG} (based on the approach in [CGN96]) in Table 5.14, the value of $55ns$ is a better clock than $24ns$ since $\alpha_{\mathsf{CG}}(55) > \alpha_{\mathsf{CG}}(24)$. This is in contrast with the exact RCS solutions in Table 5.6 since the clock length of $24ns$ results in the fastest schedule.

Thus, the definition of average slack along critical path as well as relating the slack to the fraction of useful clock is a better measure for the selection of near-optimal clock length. Such an approach showed that the results obtained are no worse than the approach in [CGN96].

Table 5.6: *RCS* results without chaining for *ARFILTER*

	1*,2+	2*,4+	4*,2+	6*,3+	6*,4+
clk	ns (steps)	ns (steps)	ns (steps)	ns (steps)	ns (steps)
163	2119 (13)	1630 (10)	1304 (8)	1304 (8)	1304 (8)
82	1640 (20)	1558 (19)	902 (11)	902 (11)	902 (11)
55	1595 (29)	1485 (27)	825 (15)	770 (14)	770 (14)
48	1776 (37)	1680 (35)	912 (19)	816 (17)	816 (17)
41	1640 (40)	1558 (38)	902 (22)	902 (22)	902 (22)
33	1650 (50)	1518 (46)	828 (26)	825 (25)	825 (25)
28	1624 (58)	1512 (54)	840 (30)	784 (28)	784 (28)
24	**1584 (66)**	**1488 (62)**	**816 (34)**	**744 (31)**	**744 (31)**
21	1659 (79)	1533 (73)	861 (41)	819 (39)	819 (39)

5.6.1 Engineering Issues

- If the number of clock steps is large although the actual time to schedule is lesser, a larger and more complex controller might be required.

- Although we have based our approach of selection of a near-optimal clock on the activity factor, α, and the experimental results obtained compare very well with the exact RCS solutions as illustrated earlier, such an approach may not always yield the best practical value of the clock length.

  Consider two clock lengths $clk_1 > clk_2$, the activity factors corresponding to which are such that $\alpha(clk_2) > \alpha(clk_1)$ and the latency $\lambda(clk_1) > \lambda(clk_2)$. Although our method would choose clk_2 to be the optimal value of the clock, if the fraction, $\dfrac{\lambda(clk_1) - \lambda(clk_2)}{\lambda(clk_2)} \to \epsilon$ (ϵ being a very small value), and the number of control steps for scheduling using the clock value clk_2 is greater than the clock clk_1, then designing a controller for clk_1 will be better because of practical design issues like size and complexity of the controller.

- We have looked at finding a near-optimal value of the clock length based on minimizing the critical path weighted average slack in a given time step and then relating this particular value of slack to the activity factor.. Another important issue that affects clock selection related to *power dissipation* in a VLSI circuit is discussed in [RJD98]. As the number of clock cycles increases, the clock distribution network in the circuit is repeatedly charged and discharged which leads to increased power consumption. Larger clock cycles may result in fewer state transitions leading to a smaller controller and hence a reduction in controller power. On the other hand, larger clock allows for chaining of functional units and since their outputs can be glitchy, it results in increased glitching power consumption.

5.7 Summary and Results

The trend of the results obtained from this method of clock selection compared quite well with the exact solution technique discussed in [BW96]. The clock selection methodology discussed in this chapter also allows for non-integral clock lengths as opposed to the approach in [BW96] which formulates the problem for integral clock lengths only. For evaluating the performance of our approach, the delays of the various functional units used are shown in Table 5.7. The slack on different functional unit types for various candidate clock lengths are shown in Table 5.8. The highlight of our technique is the weighting of the slack function by the frequency of the occurrence of operations in the critical path in a data flow graph thus taking into account the structure of the graph in some form. We also define a term called activity factor, α, which can be thought of as the useful work done in a clock cycle. The method has also been extended to include two basic types of chaining. Using α as the figure of merit for optimal clock selection, we see from table 5.5 and table 5.6 that the clock value for which the total schedule time is the least is the same as the clock value for which the α is maximum. An important point can be highlighted by analyzing the results for $ARFILTER$ from table 5.14. The activity factor for clock value 24 ns (α=0.92) is marginally greater than that for 55 ns (α=0.91). From Table 5.6, it can be noted that for a resource constraint of $\langle 1*, 2+\rangle$, the completion time (in ns) for the clock value of 24 ns

Table 5.7: Delays for functional units

FU_i	delay (ns)
multiplier	163
adder	48
subtractor	56

is the least but the scheduling time (in ns) for a clock of 55 ns is only 11 ns greater, more importantly, the number of clock steps corresponding to a clock value of 55 ns is only 29 steps as compared to 66 steps for clock value of 24 ns. This makes 55 ns as a better value of clock from controller design point of view. In this respect, we can say that for two clock values clk_1 and clk_2, if $clk_1 > clk_2$, $\alpha(clk_1) \simeq \alpha(clk_2)$ and $\lambda(clk_1) - \lambda(clk_2) \rightarrow \epsilon$ (where ϵ is a small value as compared to the schedule lengths given by $\lambda(clk_1)$ and $\lambda(clk_2)$), then a higher value of clock length, clk_1, would lead to a lesser number of time steps. This is better in terms of minimizing the number of states for the finite state machine controller.

Table 5.8: Slack for different clocks

clk (ns)	mult (ns)	add (ns)	sub (ns)
163	0	115	107
82	1	34	26
81.5	0	33	25.5
56	5	8	0
55	2	7	54
54.33	0	6.33	52.66
48	29	0	40
41	1	7	26
40.75	0	33.5	25.5
33	2	18	10
32.59	0	17.19	9.19
28	5	8	0
27.16	0	6.33	25.5
24	5	0	16
23.28	0	21.85	13.85
21	5	15	7
20.37	0	13.12	5.12

Table 5.9: Results for *ARFILTER* for non-integral clocks

clk (ns)	CGslack (ns)	α_{CG}	CPWslack (ns)	α_{CPW}
163	49.45	0.70	71.87	0.56
81.5	14.40	0.82	20.9	0.74
54.3	**2.72**	**0.95**	**3.9**	**0.93**
48	16.53	0.66	10.8	0.77
40.75	14.40	0.65	20.9	0.48
32.59	7.40	0.77	10.7	0.67
27.16	2.72	0.85	3.9	0.86
24	**2.85**	**0.88**	**1.8**	**0.92**
23.28	9.40	0.60	13.6	0.41
20.37	5.64	0.72	8.2	0.60

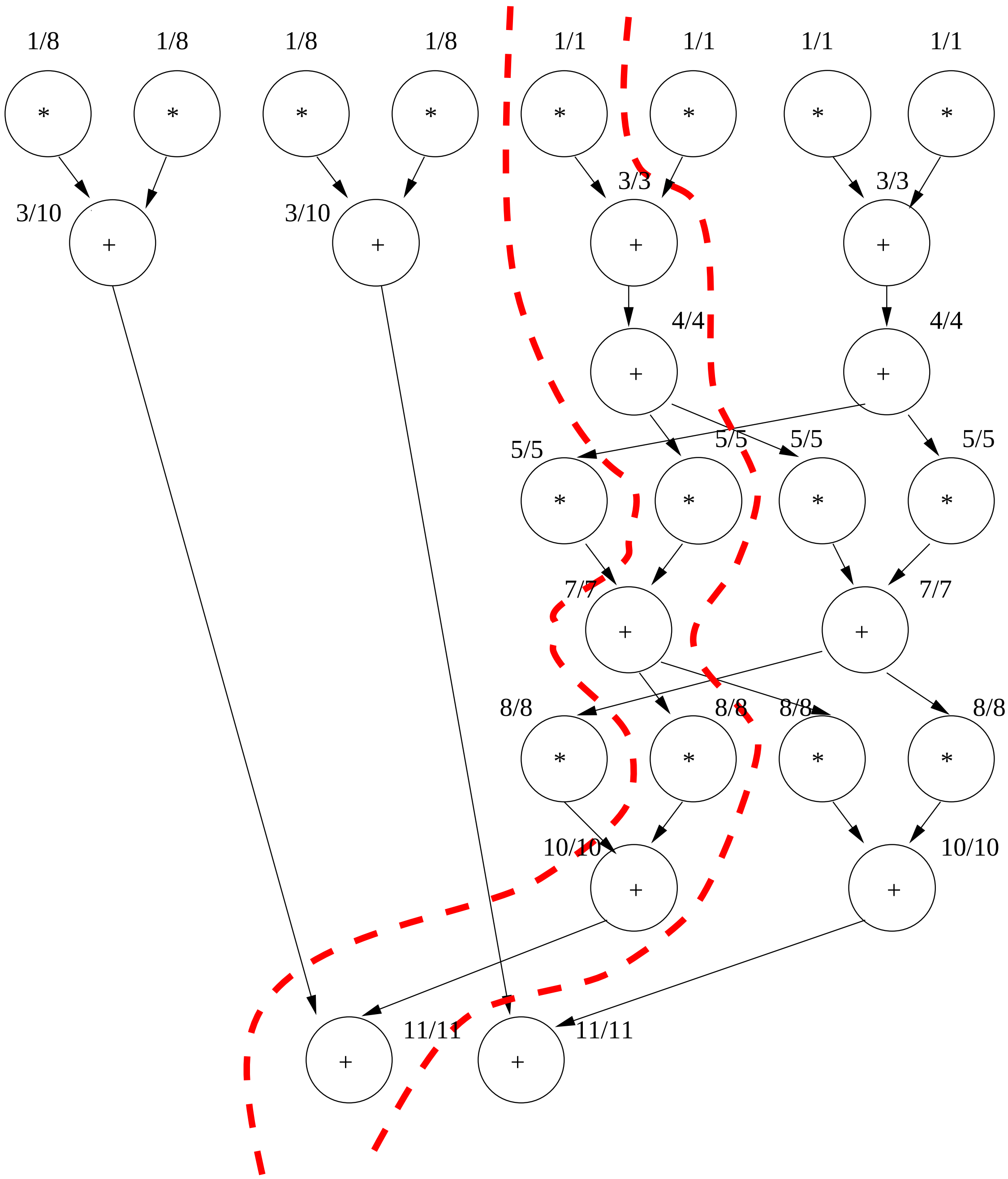

Figure 5.7: Critical path in *ARFILTER* graph

Table 5.10: Results for *DCT* for non-integral clocks

clk (ns)	CGslack (ns)	α_{CG}	CPWslack (ns)	α_{CPW}
163	55.15	0.66	94.57	0.42
81.5	15.21	0.81	26.60	0.67
56	**5.27**	**0.90**	**6.17**	**0.89**
54.3	10.05	0.81	13.01	0.76
48	20.79	0.57	11.52	0.76
40.75	15.21	0.63	26.60	0.35
32.59	7.23	0.78	13.00	0.60
28	5.27	0.81	6.17	0.78
27.16	5.98	0.78	8.48	0.69
24	4.95	0.79	3.50	**0.85**
23.28	9.51	0.59	16.89	0.27
20.37	5.23	0.74	9.61	0.53

Table 5.11: Results for *F2* for non-integral clocks

clk (ns)	CGslack (ns)	α_{CG}	CPWslack (ns)	α_{CPW}
163	83.95	0.48	76.65	0.53
81.5	24.45	0.70	22.33	0.73
56	7.19	0.87	6.99	0.87
54.3	**4.62**	**0.91**	**4.22**	**0.92**
48	7.83	0.84	9.66	0.80
40.75	24.45	0.40	22.33	0.45
32.59	12.56	0.61	11.46	0.65
28	7.19	0.74	6.99	0.75
27.16	4.62	0.83	4.22	0.84
24	**1.35**	**0.94**	**1.66**	**0.93**
23.28	15.95	0.31	14.56	0.37
20.37	9.58	0.53	8.74	0.57

Table 5.12: Results for *FDCT* for non-integral clocks

clk (ns)	CGslack (ns)	α_{CG}	CPWslack (ns)	α_{CPW}
163	68.82	0.58	77.19	0.53
81.5	18.29	0.78	19.96	0.75
56	**4.38**	**0.92**	**3.70**	**0.93**
54.33	18.29	0.66	25	0.54
48	23.42	0.51	27.23	0.43
40.75	18.29	0.55	19.96	0.51
32.59	8.18	0.75	8.52	0.74
28	**4.38**	**0.84**	**3.70**	**0.87**
27.16	9.87	0.64	12.95	0.52
24	6.86	0.71	8.73	0.64
23.28	11.07	0.52	11.79	0.50
20.37	5.66	0.72	5.66	0.72

Table 5.13: Results for *EWF* for non-integral clocks

clk (ns)	CGslack (ns)	α_{CG}	CPWslack (ns)	α_{CPW}
163	87.16	0.46	90.27	0.45
81.5	25.22	0.69	26.29	0.68
56	6.99	0.87	7.35	0.87
54.33	**6.20**	**0.89**	**4.97**	**0.91**
48	7.87	0.84	6.23	0.87
40.75	25.22	0.38	26.29	0.35
32.59	12.83	0.61	13.50	0.58
28	6.99	0.75	7.35	0.74
27.16	5.39	0.80	4.97	0.82
24	**1.63**	**0.93**	**1.07**	**0.95**
23.28	16.37	0.30	17.15	0.26
20.37	9.73	0.52	10.30	0.49

Table 5.14: Results for *ARFILTER* for integral clocks

clk (ns)	CGslack (ns)	α_{CG}	CPWslack (ns)	α_{CPW}
163	49.45	0.70	71.87	0.56
82	15.19	0.81	21.62	0.74
55	**4.15**	**0.92**	**5.12**	**0.91**
48	16.53	0.65	10.87	0.77
41	15.19	0.63	21.62	0.47
33	8.88	0.73	12	0.64
28	6.29	0.77	6.87	0.75
24	**2.85**	**0.88**	**1.87**	**0.92**
21	9.3	0.56	11.25	0.46

Table 5.15: Results for *DCT* for integral clocks

clk (ns)	CGslack (ns)	α_{CG}	CPWslack (ns)	α_{CPW}
163	55.15	0.66	94.57	0.42
82	15.97	0.80	27.18	0.67
56	**5.27**	**0.91**	**6.17**	**0.89**
55	11.50	0.79	14.02	0.74
48	20.79	0.57	11.52	0.76
41	15.97	0.61	27.18	0.34
33	8.64	0.74	14	0.57
28	5.27	**0.81**	6.17	**0.78**
24	4.95	**0.79**	3.5	**0.85**
21	8.70	0.59	12	0.43

Table 5.16: Results for *F2* for integral clocks

clk	CGslack	α_{CG}	CPWslack	α_{CPW}
163	83.95	0.48	76.65	0.53
82	25.09	0.69	23.00	0.72
56	7.19	0.87	7.00	0.87
55	**5.65**	**0.90**	**5.33**	**0.90**
48	7.83	0.84	9.66	0.80
41	25.09	0.39	23.00	0.44
33	13.68	0.58	12.66	0.62
28	7.19	0.74	7.00	0.75
24	**1.35**	**0.94**	**1.66**	**0.93**
21	12.30	0.41	11.66	0.44

Table 5.17: Results for *FDCT* for integral clocks

clk (ns)	CGslack (ns)	α_{CG}	CPWslack (ns)	α_{CPW}
163	68.82	0.58	77.19	0.53
82	18.98	0.77	20.64	0.75
56	**4.38**	**0.92**	**3.7**	**0.93**
55	19.67	0.64	26.42	0.52
48	23.42	0.51	27.23	0.43
41	18.98	0.54	20.64	0.50
33	9.44	0.71	9.74	0.70
28	**4.38**	**0.84**	**3.7**	**0.87**
24	6.86	0.71	8.73	0.64
21	8.72	0.58	8.61	0.59

Table 5.18: Results for *EWF* for integral clocks

clk (ns)	CGslack (ns)	α_{CG}	CPWslack (ns)	α_{CPW}
163	87.16	0.46	90.27	0.45
82	25.83	0.68	26.9	0.67
56	6.99	0.87	7.35	0.87
55	**7.19**	**0.87**	**5.92**	**0.89**
48	7.87	0.84	6.23	0.87
41	25.83	0.37	26.9	0.34
33	13.90	0.58	14.56	0.56
28	6.99	0.87	7.35	0.74
24	**1.63**	**0.93**	**1.07**	**0.95**
21	12.31	0.41	12.85	0.39

Chapter 6

Area Delay Prediction

The area-delay curve for a design is a set of trade-off points (curve) governed by the two most important characteristics of the design, namely *area* and the *latency*. The area of a design depends not only on the number of functional units used in the design; it is a complex function of the number of registers, interconnect and other control units required for the functioning of the design. However, in most high-level synthesis systems, the area of the functional units is assumed to dominate the total area of the design. Even then the problem is not really simplified, because depending on the VLSI technology, the same functional units could have different properties. Thus, the same functional unit in different libraries can have a different area and delay. Also functional units performing the same function in a design can be obtained from different libraries. For example a designer may choose to perform an operation on a faster (slower) module having greater (lesser) area. Module selection is therefore an important issue in synthesis systems. Another factor affecting the design is the clock used in VLSI circuits. *Design space exploration* thus depends on a variety of parameters; we discuss several techniques that are developed for design space exploration based on these parameters in the next sections.

An optimal trade-off curve can be found between the area and latency by modeling all constraints in the design properly and solving the problem as an optimization problem in 2-Dimensional space. Two kinds of optimization problems can be modeled. One of them, called the *Time Constrained Scheduling* problem, optimizes area for a given latency and, on the other hand, the *Resource Constrained Scheduling* problem optimizes latency for a specified area of the design. Though such problems, when modeled as ILP problems, guarantee optimal solutions, they are computationally expensive and are not practical in many cases because a designer is interested in a getting a characterization of the design space in a short time. Recent research focuses on reducing the need to use compute exact solutions for either the RCS or the TCS problem, thereby reducing the time and complexity.

6.1 Pareto points

A design for a specified latency can be synthesized such that the functional units requirement for different resource classes can be different but the total area of the circuit remains the same. A designer is interested in finding the least area for the design for the specified latency constraint. In a 2-dimensional space, we consider the latency for a design plotted on *x-axis* and the area on the *y-axis.* The performance of the design can be evaluated for various feasible latencies λ_i corresponding to the area a_i. There can be infinite number of ordered pairs, $\langle \lambda_i, a_i \rangle$ but only some of them will be feasible. Amongst the feasible pairs, it is important to find the optimal solutions (or points) with respect to both the *area* and *latency* axes. A plot of these points and their interpolation is called *characterization* in a 2-D design space. In this chapter we will deal with the 2-D space although characterizing the space is 3-D with the *clock* as the another constraint (and hence a third dimension in the design space) in addition to the latency and area constraints is more useful. We look at a subset of a larger problem first which can be generalized later.

The points of interest to the designer are the *Pareto* points. As defined in [De 94] *a point is a Pareto point if there is no other point with at least an inferior objective, all others being inferior or equal.* A Pareto point represents a global optimum in the design space and hence is a useful source of information for the designer. The interpolation of Pareto points is called a *trade-off curve* or *surface* in a two or three dimensional space respectively. If for a given latency, there is a least area, then for that area, if the given latency is the least

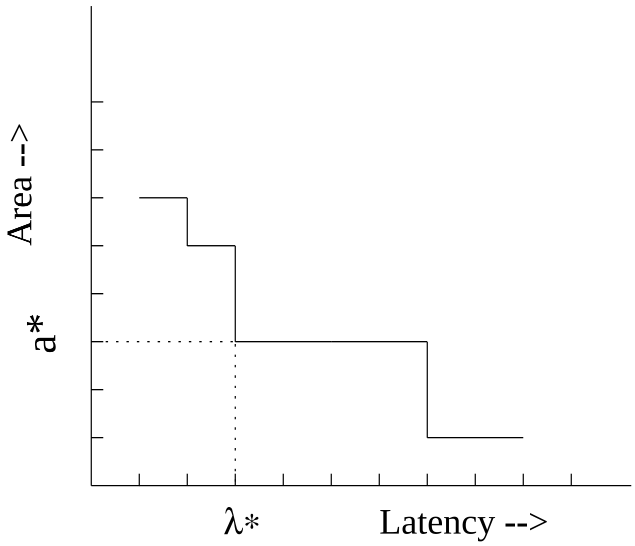

Figure 6.1: Pareto Points

possible, then the point is called a *Pareto* point. The ordered pair $\langle \lambda^*, a^* \rangle$ in the Figure 6.1 is an example of a Pareto point in a two-dimensional space. Consider the latency λ^*; a small decrease in the latency $\delta\lambda$ will increase the area of the design and any increase in the latency for the same small amount may reduce the area which is mathematically expressed as,

$$f(\lambda^* - \delta\lambda) > f(\lambda^*) \tag{6.1}$$

$$f(\lambda^* + \delta\lambda) \leq f(\lambda^*) \tag{6.2}$$

Expressed in another way, the ordered pair $\langle \lambda^*, f(\lambda^*) \rangle$ is a Pareto point if for any $\lambda < \lambda^*$, $f(\lambda) > f(\lambda^*)$ and there is no $\lambda > \lambda^*$ for which $f(\lambda) > f(\lambda^*)$. From a graphical viewpoint, the rectilinear rectangle bounded by the vertices $\langle 0, 0 \rangle$, the Pareto point $\langle \lambda^*, a^* \rangle$, and the points $\langle 0, a^* \rangle$ and $\langle \lambda^*, 0 \rangle$ contains no other point (that lies on the area-delay curve) characterizing the design space.

6.2 Terminology

We first define some mathematical terms that we will use in the following explanation of the various techniques for design space exploration. $\mathsf{TCSLB}(\lambda_i)$ denotes the TCS lower bounding function that generates a lower bound on the number of resources of each resource type. Such a lower bound on the resources is encapsulated in the resource vector $\vec{r}_{\mathsf{LB}}$. We consider module and clock selection to have been done prior to this. The particular library for resources (in particular, the multipliers and the ALU) that we use for analyzing the technique is such that the $Area$ function is defined by the equation $5 * area(ALU) + 2 * area(multiplier)$. In general the $Area$ is given by the scalar product

$$Area = \vec{r} \cdot \vec{a} \tag{6.3}$$

where, $\vec{a} = \{a_1, a_2, ..., a_m\}$. Each element a_i is the area of the resource of type i.

The RCS upper bounding function is denoted by $\mathsf{RCSUB}(a_i)$ which generates an upper bound on the latency for a given area of the design a_i. The resource vector to the area mapping is not unique since for a given $Area$, the resource vector $\vec{r}$ is not unique, i.e, there may be more that one possible resource vectors that may have the same total area.

In a similar manner, RCSLB and TCSUB denote the lower bounding for the RCS problem and the upper bounding for the TCS problem respectively.

Also TCSILP and TCSOPT denote the exact solutions for the TCS problem and similarly RCSOPT is the exact solution for the RCS problem.

6.3 Previous work

The area delay prediction problem has been addressed in several ways. Instead of using only exact solution methodology, a combination of various bounding methods (upper as well as lower) have been used to either explore along the time axis or the area axis. Heuristic bounding methods are useful in this respect to achieve a fast characterization. We take a brief look at two approaches to the area-delay prediction problem.

6.3.1 NEAT group approach

The NEAT group approach [THJ93], [TJ93], [Tim93] is based on characterizing the design space based on TCS and RCS lower bounding techniques. The technique is an iterative process that is initialized by the least possible $\vec{r}$ for which a valid schedule can be found. The lower bound on the latency, λ_{LB}, is calculated for that resource vector. The latency is thereafter decremented by an amount $\delta\lambda$ and a lower bound a_{LB} on the resource utilization (area) is found. For a_{LB}, the lower bound on the latency is found. The latency is repeatedly decremented and the process continued till reaching a least latency for which a feasible schedule can be found. In this approach the TCS lower bounding problem is solved as a relaxed Mixed Integer Linear Programming (MILP) problem by relaxing the precedence constraints between operations in the graph. The formulation also allows for selecting resources from different libraries. The RCS lower bounding problem is formulated as a heuristic based on *Bipartite Graph Matching*. The procedure is illustrated by means of an algorithm in Figure 6.2.

6.3.2 The Voyager approach

Another approach for characterizing the exact design space can be found in the Voyager project's solution [BW96] and [CBW97]. This method also incorporates module selection as well as the clock as another parameter to get a more realistic feel of the design space. From among a large set of points in the design space, the approach attempts at fast classification of the points into two sets, namely, the set of *Pareto* points and the set of *non-Pareto* points. As mentioned earlier, a combination of a variety of techniques in high-level synthesis can be used; this particular method explores only along the time axis. It repeatedly solves the TCS lower bounding and the TCS exact solutions problems between a range $[\lambda_{\mathsf{min}}, \lambda_{\mathsf{max}}]$ to obtain the trade-off points. The latency, λ_{min}, is the minimum possible latency and λ_{max} is

Procedure $\mathsf{NEATExploration}$
$\lambda_{\mathsf{absmin}} \leftarrow$ Minimum possible latency
$\lambda_{\mathsf{init}} \leftarrow \mathsf{TCS}(\vec{r}_{\mathsf{absmin}})$
$\lambda \leftarrow \lambda_{\mathsf{init}}$
$flag \leftarrow \mathsf{TRUE}$
while $(flag = \mathsf{TRUE})$ **do**
  $\lambda \leftarrow \lambda - \delta\lambda$
  if $(\lambda = \lambda_{\mathsf{absmin}})$ **then**
    $flag \leftarrow \mathsf{FALSE}$
  else $area_{\mathsf{LB}} \leftarrow \mathsf{MILPTCSLB}(\lambda)$
    $\lambda_{\mathsf{LB}} \leftarrow \mathsf{RCSLB}(area_{\mathsf{LB}})$
  endif
enddo

Figure 6.2: Characterization by NEAT group

the maximum possible latency for valid schedules. At each time constraint $\lambda \in [\lambda_{\mathsf{min}}, \lambda_{\mathsf{max}}]$, the schedule ϕ_{ASAP} is calculated first to determine whether a feasible schedule exists. This if followed by computing a TCS lower bound, a_{LB}, for that λ. If the current a_{LB} is smaller than the previous lower bound on the area, then it is a *potential Pareto* point. Then an upper bound on the area, a_{UB}, is found using a TCS upper bounding method. If $a_{\mathsf{LB}} = a_{\mathsf{UB}}$, then the point is an optimal solution. If an optimal solution is found in the case of a potential Pareto point, then it is a guaranteed Pareto point. This approach is illustrated in Figure 6.3. If the lower and upper bounds are not equal, then a tighter lower bound on the area is found by TCSLP relaxation. If this also fails to be useful, then an exact solution is obtained based on an ILP formulation. The technique is also extended to include module and clock selection.

6.4 A New Approach

As illustrated above, both the methods for an area-delay curve prediction are based either only on one kind of bounding method or on one kind of scheduling method. The NEAT approach uses TCS and RCS lower bounding, whereas the approach in [BW96] uses only

```
Procedure VoyagerExploration
Find λ_min and λ_max
a_cur ← maxarea
Find all candidate λ_i ∈ [λ_min, λ_max]
for each λ_i ∈ [λ_min, λ_max] do
    if λ_i < λ(φ_ASAP) then
        Infeasible Schedule
        Exit
    endif
    a_LB ← TCSLB(λ_i)
    a_UB ← TCSUB(λ_i)
    if (a_LB ≥ a_cur then
        ⟨λ_i, f(λ_i)⟩ is not a Pareto point
    else
        if (a_LB=a_UB) then
            (λ_i, f(λ_i)) is a Pareto point
            a_cur ← a_LB
        else
            a_RLB ← LPTCSLB(λ_i)
            if (a_RLB=a_UB) then
                (λ_i, f(λ_i)) is a Pareto point
                a_cur ← a_RLB
            else
                a_OPT ← ILPTCS(λ_i)
                if (a_OPT < a_cur) then
                    (λ_i, a_OPT) is a Pareto point
                    a_cur ← a_OPT
                else
                    ⟨λ_i, f(λ_i)⟩ is not a Pareto point
                endif
            endif
        endif
    endif
enddo
```

Figure 6.3: Characterization in Voyager

TCS upper and lower bounding for design space exploration. Based on bounding methods, there can be six possibilities in which we can combine the various techniques at hand. This motivates us to look at other combinations of heuristic bounding methods for exploring the design space. It can be seen that the NEAT approach involves solving the time constrained MILP lower bounding problem for each latency in the range $[\lambda_{min}, \lambda_{max}]$. Even in the approach adopted by [CBW97], the it may be required to find LP relaxed lower bounds and in a worst case, it may also be required to run the ILP solution. This can at times turn out to be quite expensive to classify the design space into a set of *Pareto* and *non-Pareto* points. This motivates us to come up with techniques utilizing good bounding techniques so as to reduce the use of exact solutions. It has been shown (experimentally) [Nar98] that the results obtained from the TCS F-B type bounding method compares quite well with the LP based lower bounds. The RCS list scheduler also generates optimal solutions using tighter ALAP based priority of operations [Nar98]. This motivates us to use a combination of these to search the design space along the area and the latency axes. This alternating between the area and time axis and also between upper and lower bounding techniques is the important highlight of our approach.

6.4.1 Using TCS lower and RCS upper bounds

Exploration using lower and upper bounding methods along the same axis (either along the area or along the latency axes) yields optimal solution with respect to that axis if lower bound equals the upper bound. As an illustration, if exploring along the latency axis, if $\mathsf{TCSUB}(\mathsf{TCSLB}(\lambda_i))$, then we have an optimal area requirement for some time constraint λ_i. But such an optimal solution does not guarantee a Pareto point in the design space because, if $\mathsf{TCSOPT}(\lambda_i) = a_i$, then it may be possible that $\mathsf{RCSOPT}(a_i) \leq \lambda_i$. Only if the equality holds true, then we have a Pareto point. It follows that every Pareto point is an optimal solution, but all optimal solutions are not Pareto points. Essentially, by the definition of a Pareto point, it is an optimal solution with respect to both the *area* as well as the *latency* axis. This prompts us to explore the design space using upper and lower bounding methods along both the axes to obtain the Pareto points efficiently.

Design Space Exploration based on upper and lower bounding methods can be done in two ways.

- We can explore starting either along the area axis or the latency axis. If exploration is in the direction of increasing latency for each λ_i (say), then the lower bounds on area $(a_{\mathsf{LB}})_i$ (actually the functional units) for each latency can be calculated, and for each point $(a_{\mathsf{LB}})_i$ an upper bound on the latency can be calculated. The points at which the lower bound curve coincides with the upper bound curve are the Pareto points. The time constraints for which Pareto points cannot be ascertained using the bounding methods, further exploration is required, perhaps by using tighter bounding techniques or computing exact solutions for those time constraints.

- Another variation of the above scheme could be to start exploring in the increasing direction of the area axis in a similar manner but as mentioned earlier we use the former in our methodology since good TCS lower bounds and RCS upper bounds can be obtained.

We begin exploring the design space in the direction of increasing latency.

6.4.2 Finding Pareto Points

As a first step, the TCS lower bounds are found in the direction of increasing latency. As explained above, a lower bound on the area $(a_{\mathsf{LB}})_i$ for different integral latencies λ_i is computed. From the definition of *Pareto* points, we can say that if $\lambda_{i+1} > \lambda_i$, i.e, if there is a transition in the lower bound on the area such that $(a_{\mathsf{LB}})_{i+1} < (a_{\mathsf{LB}})_i$, then the ordered pair $\langle \lambda_{i+1}, f(\lambda_{i+1}) \rangle$ is a *potential Pareto point.*

$\mathsf{RCSUB}(\mathsf{TCSLB}(\lambda_i)) = \lambda_i$**:** the ordered pair $\langle \lambda_i, \mathsf{TCSLB}(\lambda_i) \rangle$ is an *optimal solution* and a *Pareto point*. In this case, the lower bound on the area, $a_{\mathsf{LB}} = \mathsf{TCSLB}(\lambda_i)$. A lower bound on the number of resources denoted by $\vec{r}_{\mathsf{LB}}(\lambda_i)$, for a given latency λ_i requires that for any other $\vec{r}$ for the same latency the difference $\vec{r}_{\mathsf{LB}} - \vec{r} < \vec{0}$. This holds true since the lower bound on the number of resources belonging to a resource class i, $(r_{\mathsf{LB}})_i \in \vec{r}_{\mathsf{LB}}$ is calculated independent of any other resource class and hence the $\vec{r}_{\mathsf{LB}}$ is an absolute lower bound on the number of resources used for each resource class. This translates to an absolute lower bound on the area for that resource vector. Moreover, for any two values of latency λ_i, λ_j such that $\lambda_j < \lambda_i$, then $\vec{r}_{\mathsf{LB}}(\lambda_j) \geq \vec{r}_{\mathsf{LB}}(\lambda_i)$ will always hold true. In particular, if $\vec{r}_{\mathsf{LB}}(\lambda_j) > \vec{r}_{\mathsf{LB}}(\lambda_i)$ (strictly greater than), then the ordered pair $\langle \lambda_i, \vec{r}_{\mathsf{LB}}(\lambda_i) \rangle$ is a *potential Pareto point*. For this unique resource vector $\vec{r}_{\mathsf{LB}}(\lambda_i)$, if the upper bounding function $\mathsf{RCSUB}(a_{\mathsf{LB}}) = \lambda_i$, then we have an optimal solution and a Pareto point. If the potential Pareto point is not confirmed to be an optimal solution, then further exploration is required in order to obtain the Pareto point.

$\mathsf{RCSUB}(\mathsf{TCSLB}(\lambda_i)) > \lambda_i$**:** here, it is possible that either the RCS upper bounding or the TCS lower bounding technique is not as tight in this particular case and if some ordered pair $\langle \lambda_i, a_{\mathsf{LB}} \rangle$ happens to be a potential Pareto point, then we cannot guarantee whether the point is actually a Pareto point and further exploration would be required. This is achieved by performing an exact $\mathsf{TCSILP}(\lambda_i)$ formulation and finding the optimal solution for area, a_{OPT}.

$\mathsf{RCSUB}(\mathsf{TCSLB}(\lambda_i)) < \lambda_i$**:** here, it is possible that more than one $\vec{r}$ may map to the same area for the design, so the upper bounds on the latency need to be calculated for each of these vectors. Let Res be the set of resource vectors possible for the same area. For the vector $\vec{r}_{\mathsf{LB}} \in Res$ found earlier, the upper bound can never be less than λ_i. It is possible for some resource vector $\vec{r}$ such that $Area(\vec{r}) = Area(\vec{r}_{\mathsf{LB}})$ and further it may also be possible that upper bound on the latency for $\vec{r}$, given by $\mathsf{RCSUB}(\vec{r}) = \lambda_j$, such that $\lambda_j < \lambda_i$. But, by the property of lower bound, the TCS lower bound, $\mathsf{TCSLB}(\lambda_j)$ will be at the most be equal to $Area(\vec{r})$. Therefore, the curve formed by the interpolation of points obtained by TCS lower bounding is guaranteed to lie '*below*' or at the most coincide with the curve formed by the interpolation if the points obtained by RCS upper bounding.

An important thing to be kept in mind with respect to the interpolation of the RCS upper bounding curve is as follows:

A given latency λ_i can be satisfied by more than one resource vectors and hence ordered pairs of the form $\langle \lambda_i, a_j \rangle$, where $j = [1, n]$, such that there are n possible values of a_j for λ_i. These are the points lying along a vertical line in the design space. The ordered pair $\langle \lambda_i, \min(\forall_{j=[1,n]} a_j) \rangle$ is the dominating point along that vertical line and it is selected to lie on the RCS upper bounding curve, all other points along that vertical line are discarded.

The algorithm for characterizing the design space using TCSLB and RCSUB is illustrated in Figure 6.4.

6.4.3 Further Exploration of Potential Pareto Points

It may be possible that a predicted *potential Pareto point* may not actually be an optimal solution. The following conditions may arise.

- The predicted potential Pareto point may not be a Pareto point at all. This occurs if the TCS lower bounding technique generates '*loose*' bounds.
- A point which was not predicted as a potential Pareto point may turn out to be a Pareto point.

Thus if a potential Pareto point is not an optimal solution and hence not a Pareto point in the design space, then a region that requires further exploration needs to be defined.

Procedure Exploration
Find λ_{min} and λ_{max}
$a_{cur} \leftarrow maxarea$
Find all candidate $\lambda_i \in [\lambda_{\mathsf{min}}, \lambda_{\mathsf{max}}]$
for each $\lambda_i \in [\lambda_{\mathsf{min}}, \lambda_{\mathsf{max}}]$ **do**
    if $\lambda_i < \lambda(\phi_{\mathsf{ASAP}})$ **then**
        Infeasible Schedule
        Exit
    endif
    $a_{\mathsf{LB}} \leftarrow \mathsf{TCSLB}(\lambda_i)$
    if $(a_{\mathsf{LB}} < a_{cur})$ **then**
        $\langle \lambda_i, a_{\mathsf{LB}} \rangle$ is a *potential Pareto point*
        $a_{cur} \leftarrow a_{\mathsf{LB}}$
    endif
    for all $\vec{r}$ such that $a_{\mathsf{LB}} = area(\vec{r})$ **do**
        $\lambda_{\mathsf{UB}} \leftarrow \mathsf{RCSUB}(\vec{r})$
        if $(\lambda_i = \lambda_{\mathsf{UB}})$ **then**
            $\langle \lambda_i,\ a_{\mathsf{LB}} \rangle$ is a *Pareto point*
        else
            $\langle \lambda_i,\ a_{\mathsf{LB}} \rangle$ is not a *Pareto point*
        endif
    enddo
enddo

Figure 6.4: Exploration using TCSLB and RCSUB

The region to be further exploration is given by, $\lambda_i \leq \lambda_k \leq \lambda_j$, where a potential Pareto point is predicted for the latency λ_i and the next Pareto point in the direction of increasing latency occurs for a latency λ_j. We have a *Pareto* point for $\langle \lambda_k, \mathsf{TCSOPT}(\lambda_\mathsf{k}) \rangle$ if and only if $\mathsf{RCSOPT}(\mathsf{TCSOPT}(\lambda_k)) = \lambda_k$.

6.5 Summary

The area delay curve is useful to obtain the various optimal trade off points for a design. This chapter illustrates an approach based on heuristic bounding techniques for a fast pruning of the design space. The use of ILP techniques to obtain solutions is drastically reduced. Also the regions for which the optimal solutions need to be obtained can be found very

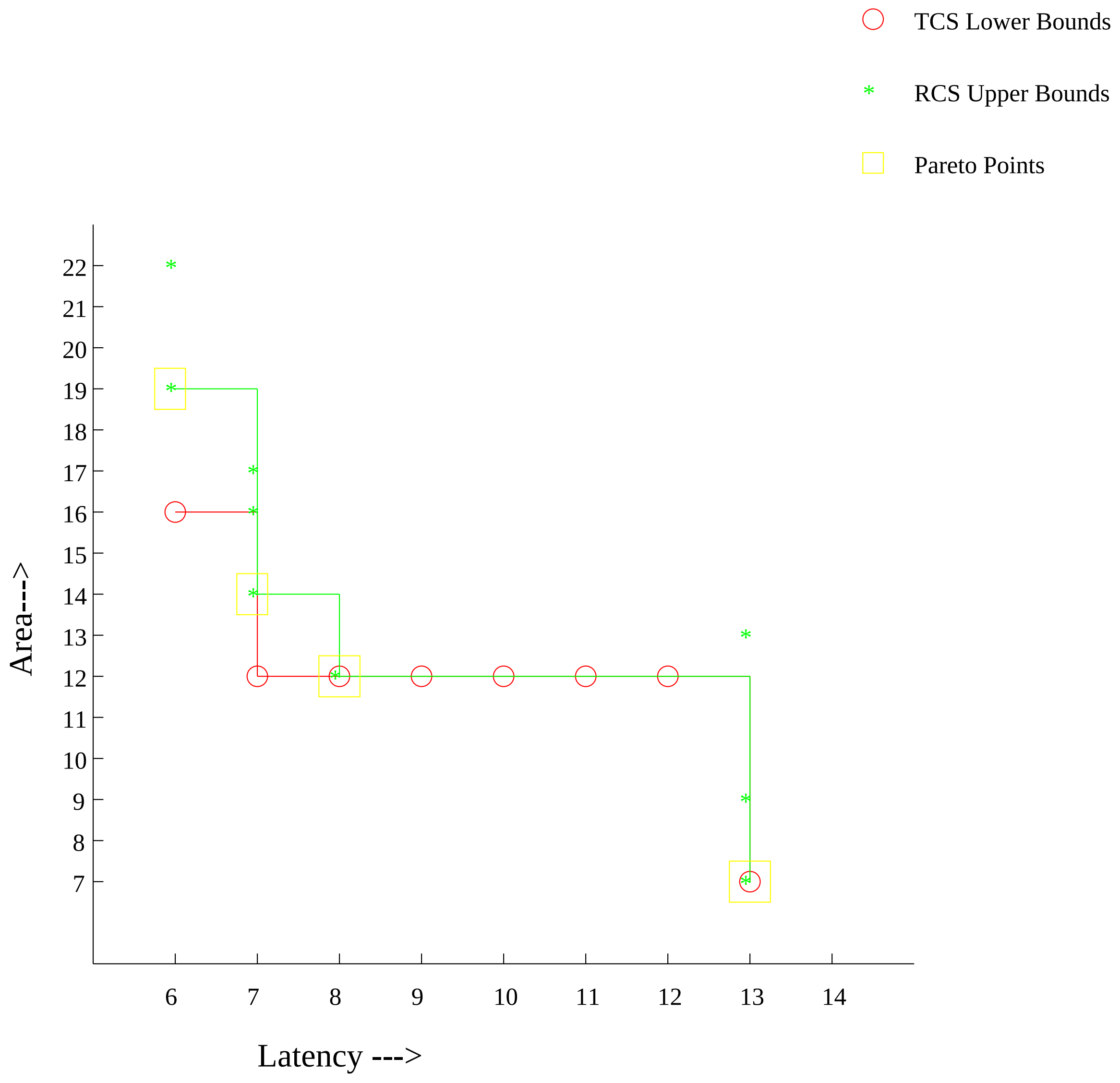

Figure 6.5: Area Latency Curve for *HAL* benchmark using multicycle multiplier

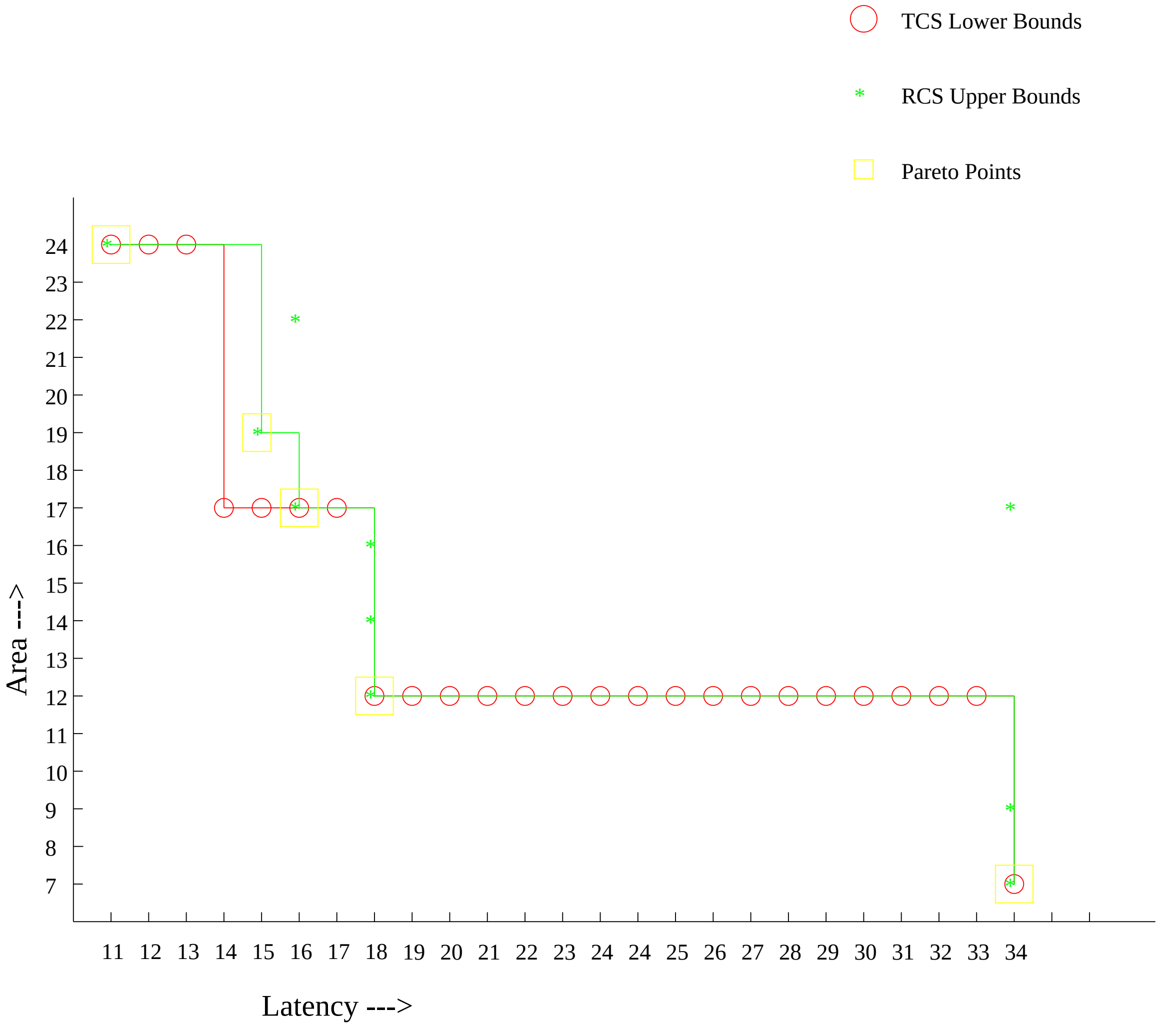

Figure 6.6: Area Latency Curve for ARF benchmark using multicycle multiplier

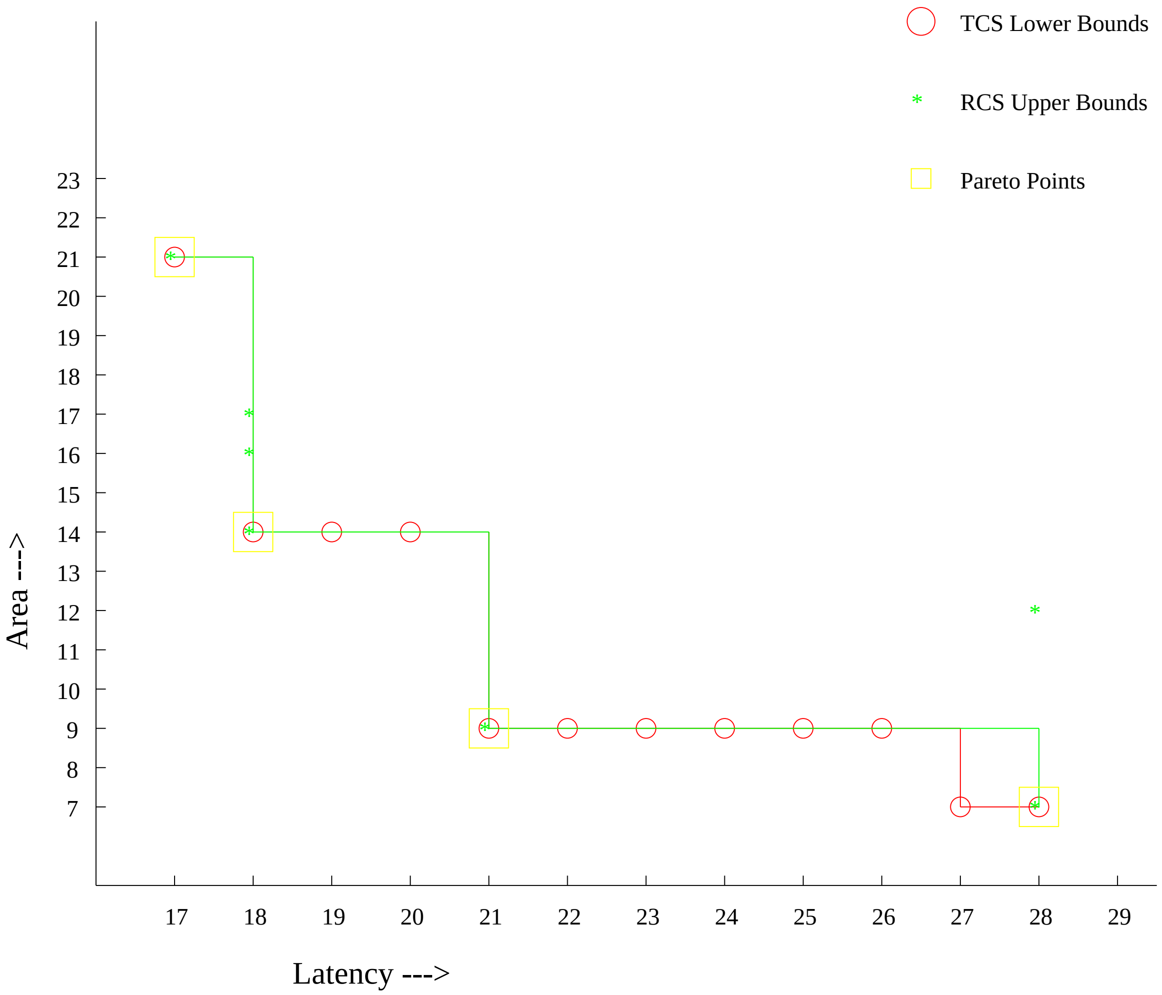

Figure 6.7: Area Latency Curve for *EWF* benchmark using multicycle multiplier

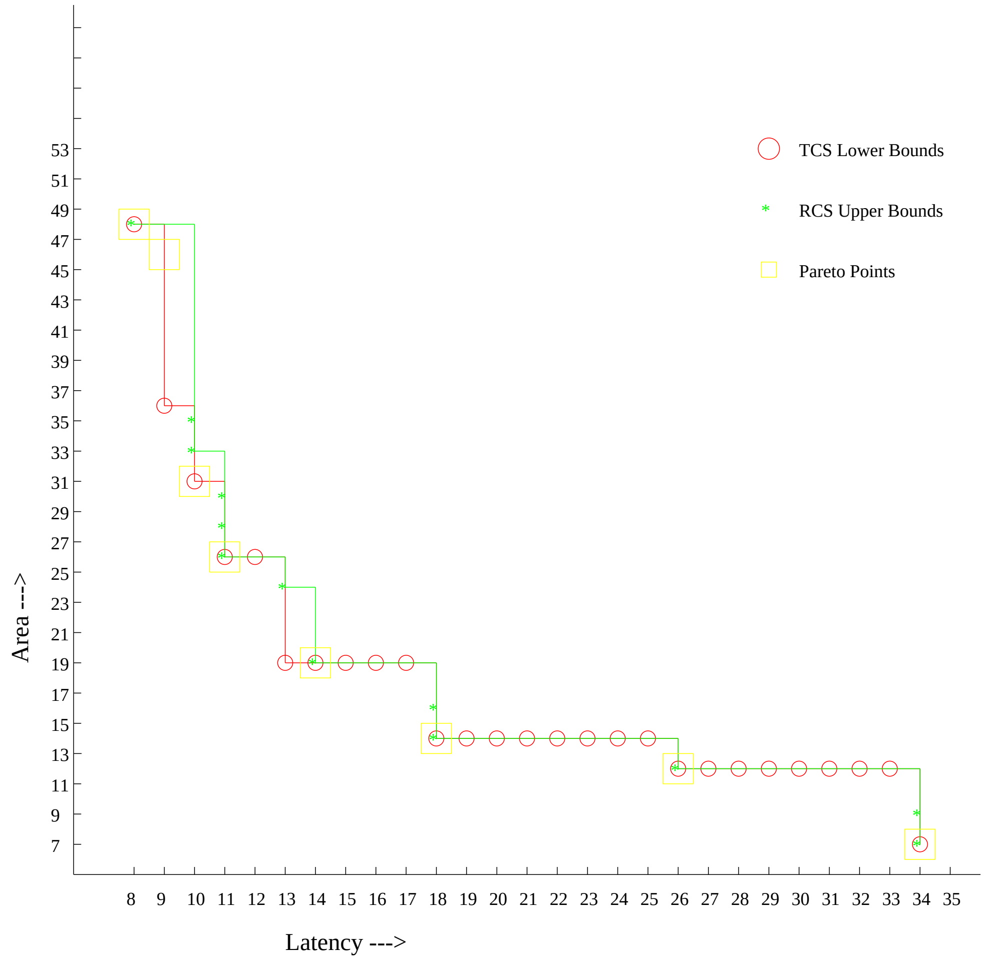

Figure 6.8: Area Latency Curve for $FDCT$ benchmark using multicycle multiplier

Table 6.1: Effectiveness of the Exploration technique

benchmark	ILP solutions	Time
HAL	2	0.02
ARFILTER	2	72.60
EWF	1	-
FDCT	3	440.99

quickly. In many cases, the predicted Pareto points or the potential Pareto points turn out to be the actual Pareto points or in the worst case are very near to them. In this manner, a combination of good bounding techniques and optimal solution techniques can help in obtaining the Pareto points quickly. Although in this work we have demonstrated the importance of the technique for a two dimensional design space exploration, there is a need to incorporate clock as well as module selection for a more holistic system level area delay curve prediction. The number of exact solutions needed for design space exploration for any benchmark determines the effectiveness of the technique. Using our approach, the number of exact solution required for the four benchmarks we used, are tabulated in Table 6.1. The approach in [BW96] considers TCS upper bounding for a preliminary exploration of Pareto points which may not be very useful since heuristics for TCS upper bounds are not tight. Hence in the loop body in Figure 6.3, the LP based lower bounding technique may have to be used for a large number of cases. None of that is necessary in our approach, since by a combination of bounding techniques along both the axes, we are able to find the exact regions for further exploration very fast. The area-delay curve characterization for various benchmarks is illustrated in Figure 6.5 to Figure 6.8. The number of instances in which exact solutions were required to be computed were 2 each for *HAL* , 1 for *EWF* and 3 for *FDCT*. The best times required for obtaining the exact solutions using Refined ILP and Vanilla ILP techniques have been cited from [Nar98].

Chapter 7

Conclusions

7.1 Contributions

This thesis focuses on the development of heuristic solutions for several problems that arise in efficient design space exploration for the synthesis of digital systems from high-level (or behavioral) specifications. It deals with time-constrained scheduling (TCS) as well as resource-constrained scheduling (RCS) problems. In particular, this work was concerned with developing better heuristic algorithms for obtaining lower bounds on the number of resources (and hence the area) utilized in the design. The importance of incorporating TCS lower bounds in obtaining solutions for the TCS upper bounds was stressed and better solutions were obtained by an improved iterative rescheduling technique. Also some improvements to the *force-directed scheduling* algorithm pertaining to the ordering of operations to schedule and gradual time frame reduction were incorporated and upper bounds were obtained for various benchmarks. Another highlight of the work was a selection of near-optimal clock value without the necessity of computing exact solutions and finally a fast method for computing Pareto points.

7.2 Future Work

Although the thesis stresses on heuristics, it was felt that a combination of tight heuristic techniques and exact solution techniques can be very effective in design space exploration. Although the two important factors affecting the scheduling process are functional unit area and the time to completion, the contribution to the area and the effect on the performance due to interconnect as well as registers should also be taken into account. The effect of a schedule on the power dissipation of a design is also an point of concern in low power CAD. Module selection, yet another important factor affecting the design, assumed to have been done *a priori* should also be incorporated in the scheduling process. The approach also should be extended to scheduling for loops and conditionals.

Bibliography

[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.

[CBW97] Samit Chaudhuri, Stephan A. Blythe, and Robert A. Walker. A solution methodology for exact design space exploration in a three dimensional design space. *IEEE Transactions on Very Large Scale Integration*, 5(1):0–0, Mar 1997.

[CGN96] En-Shou Chang, Daniel D. Gajski, and Sanjiv Narayan. An optimal clock period selection method based on clock slack minimization criteria. *ACM Trans. Design Automation of Electronic Systems*, 1(4):352–370, July 1996.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to algorithms.* MIT Press, 1990.

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

[FB73] 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.

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

[HHL91] C.T. Hwang, Y.C. Hsu, and Y.L. Lin. A formal approach to the scheduling problem in high level synthesis. *IEEE Trans. Computer-Aided Design*, 10(4):464–475, April 1991.

[Kri97] Vijay S. Krishnamurthy. Personal communication, May 1997.

[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.

[Nar98] M. Narasimhan. Exact scheduling techniques for high level synthesis, 1998. M.S. thesis, Louisiana State University.

[NR97] M. Narasimhan and J. Ramanujam. A fast approach to computing exact solutions to the resource-constrained scheduling problem, Jul. 1997. Submitted for publication, revised Apr. 98.

[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.

[PK87] P.G. Paulin and J.P. Knight. Force directed scheduling in automatic data path synthesis. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 195–202, Miami Beach, June 1987. ACM and IEEE Computer Society.

[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.

[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.

[RJD98] Anand Raghunathan, Niraj K. Jha, and Sujit Dey. *High-level Power Analysis and Optimization*. Kluwer Academic Publishers, 1998.

[SJ93] A. Sharma and R. Jain. Estimating architectural resources and performance for high-level synthesis applications. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 355–360, Dallas, June 1993. ACM and IEEE Computer Society.

[THJ93] A.H. Timmer, M.J.M. Heijligers, and J.A.G. Jess. Fast system-level area-delay curve prediction. In *first Asia Pacific Conference on Hardware Description Languages, Standards and Applications*, pages 198–207, Brisbane (Australia), December 1993.

[Tim93] A.H. Timmer. System-level estimation of functional area. In *Proceedings of the Workshop on Circuits, Systems and Signal Processing*, pages 107–112, Houthalen, Belgium, March 1993. IEEE Benelux & ProRisc.

[TJ93] A.H. Timmer and J.A.G. Jess. Execution interval analysis under resource constraints. In *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pages 454–459, Santa Clara, November 1993. IEEE Computer Society Press.

[VAKL91] W.F.J. Verhaegh, E.H.L. Aarts, J.H.M. Korst, and P.E.R. Lippens. Improved force-directed scheduling. In *Proceedings of the European Conference on Design Automation*, pages 430–435. IEEE Computer Society Press, February 1991.

Vita

Sandeep Deshpande is from New Delhi, India. After receiving a bachelor of engineering degree in Electronics and Communication Engineering from Faculty of Engineering and Technology, Jamia Millia Islamia, New Delhi, India, in 1996, he joined the graduate program in Electrical Engineering at the Louisiana State University in the Fall of 1996. He expects to receive his master of science degree in Electrical Engineering in August, 1998.

Deshpande, Sandeep, B.E., Jamia Millia Islamia, New Delhi, 1996
Master of Science in Electrical Engineering, Summer Commencement, 1998
Major: Electrical Engineering
Efficient Techniques for Problems in High Level Synthesis
Thesis directed by Associate Professor Jagannathan Ramanujam
Pages in thesis, 102. Words in abstract, 245.

Abstract

Efficient techniques for design space exploration in high-level synthesis are critical for the automatic synthesis of optimal or near-optimal VLSI designs. High-level synthesis consists of several phases. An important phase that has a tremendous impact on the final design is the *scheduling* phase that is concerned with the distribution of various operations in time. Although the scheduling problem can be modeled as an integer linear programming problem in order to obtain optimal solutions, such a modeling under a variety of constraints can be cumbersome and the time to solve them is prohibitively high. It is therefore essential that heuristic solutions for the scheduling problems gain importance. This thesis concentrates on the improvement of the existing and development of new lower bounds as well as heuristics for scheduling problems under both resource and time (performance) constraints. In addition to scheduling, good *clock selection* impacts performance. In most synthesis systems, a convenient value of the clock is chosen or exact methods have been used for clock selection. This thesis discusses a novel heuristic approach for near-optimal clock selection for synthesis systems. Finally, this thesis presents a new approach to the *design space exploration* problem. Unlike other approaches that use exact solutions (and therefore are extremely time-consuming) or lower bounds (and therefore are inaccurate), the solution presented here uses a clever combination of lower and upper bounds while resorting to exact solutions only when necessary. This is shown to be very effective in reducing the time required in design space exploration.