Performance Driven Synthesis of Digital Systems

Ireneusz Karkowski

Performance Driven Synthesis of Digital Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof. ir. K.F. Wakker,
in het openbaar te verdedigen ten overstaan van een commissie,
door het College van Dekanen aangewezen,
op dinsdag 5 december 1995 te 10.30 uur

door

Ireneusz KARKOWSKI

magister inżynier elektroniki
geboren te Gdynia, Polen

Dit proefschrift is goedgekeurd door de promotor:
Prof. dr. ir. Ralph H.J.M. Otten, Technische Universiteit Delft

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter	
Prof. dr. ir. Ralph H.J.M. Otten, promotor	Technische Universiteit Delft
Prof. dr. ir. A.J.v.d. Goor	Technische Universiteit Delft
Prof. ir. G. Honderd	Technische Universiteit Delft
Prof. dr. dipl-ing. J.A.G. Jess	Technische Universiteit Eindhoven
Prof. dr. ir. T. Krol	Universiteit Twente
Dr. ir. P. Stravers	Philips Nat. Lab.

ISBN: 90-5326-022-6

Contents

List of Figures

List of Tables

PREFACE

Computer-aided design (**CAD**) of digital systems has been a hot topic for at least last 20 years. After an initial period with most designers looking with ironical smiles at **CAD** tools, they soon considered them indispensable. Now we can already without doubts say that only thanks to the existence of these tools it was possible to efficiently and successfully design large-scale and high quality circuits in a large variety of application areas, ranging from computers to telecommunication and manufacturing control. The interest in the **CAD** tools is even expected to grow in the coming years.

An important and interesting area in **CAD** has always been the *synthesis* and *optimization* of digital systems. Tools belonging to this category are able to generate a cost-effective circuit in a target technology from a given abstract model (for example from a specification in a high-level language). This way the designer can concentrate more on conceptual issues, while leaving the "dirty work" to a tool. In this thesis we only concentrate on this kind of **CAD** systems. When using terms *design automation* and *synthesis systems* we will mean these systems.

For a **CAD** tool to be accepted by the designers' community it must satisfy several strict requirements. Only then it will be used in the companies interested in putting their products on the highly competitive electronic markets. Such a tool must be able to handle the complexity of present days' **VLSI**[1] circuits (reasonable running times, memory and disk usage).

Optimizing the design - why and how?

Beside correct operation circuits have to satisfy strict design constraints (for example on power consumption or speed). Within the constraints usually many circuits are feasible. Among these, some circuits are preferable over others because of some characteristic, mostly competitiveness. This is often measured by cost criteria. On one side, a circuit with acceptable speed and low cost is preferred to unnecessarily fast and expensive one. If however without an increase in the production

[1]Very Large Scale of Integration

cost, a particular circuit implementation uses less power (much less than specified by the design constraints), it would be careless to ignore it.

Optimization can be seen as a process of searching for the best design, i.e. a circuit that optimizes some objectives, subject to constraints. Many different combinations of objectives and constraints can appear in practice. To name only two, we can minimize the chip area subject to timing constraints or maximize performance subject to a constraint on the total power dissipation. Having n comparison criteria we can create a n-dimensional evaluation space. All known (or all considered) circuit implementations define points in this space. An example evaluation space for two criteria ($n = 2$) is shown in figure 0.1. The black points repre-

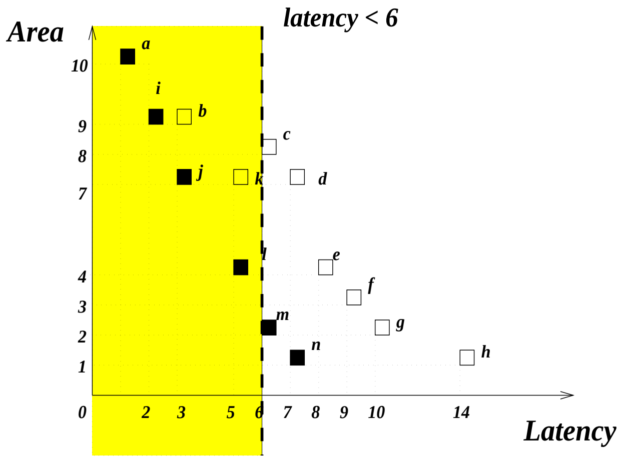


Figure 0.1. An example design space and design constraint $latency < 6$

sent so called *Pareto* points, i.e. points which correspond to the global optimum in a mono-dimensional evaluation space (there is at least one mono-dimensional evaluation-space where they are "better" than all other points). If we connect *Pareto* points by concave steps we obtain a so called *trade-off curve* (figure 0.2). The curve and the space above it (dashed area in figure 0.2 define the so called *design space* [Sin92] [SDM91]). All points above the curve represent suboptimal solutions. The space below the curve represents all not yet existing, or not yet known, solutions.

The design constraints restrict the *design space* to only these implementations which satisfy design specifications. The constraints define a space of *allowable* designs. For example, if a design constraint was: *the latency should be smaller than 6 time units*, only the points to the left of the line $latency = 6$ would represent allowable designs (shaded space in figure 0.1).

To select the best allowable design, all *allowable* circuit implementations should be compared on the basis of some criteria. For example, on chip area, power dissipation, noise margins, performance or number of i/o pins. If we are interested in only one criterion, we will just select the configuration which minimizes/maximi-

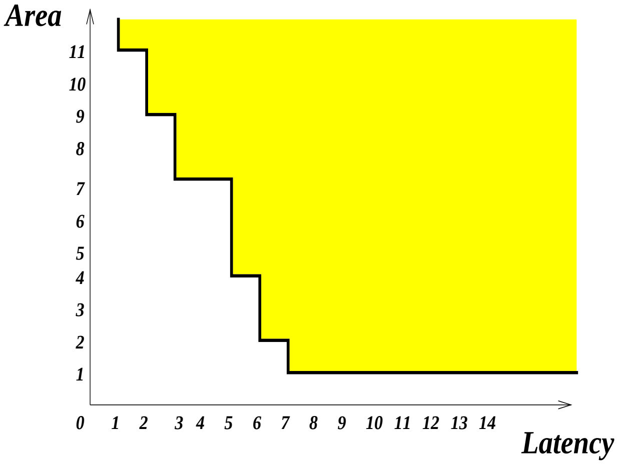


Figure 0.2. The trade-off curve

zes that criterion. Often however, several optimization criteria have to be considered simultaneously. Which solution will be selected depends on our definition of the points of optimality.

In the last years a shift of interest from the optimization for minimal area to the maximization of performance has been observed. Due to the advances in ic-technology still larger circuits can be implemented on a single carrier. In the same time the performance and power consumption requirements are more aggressive. In this thesis we will concentrate mostly on performance issues. The methods, we will propose, are often applicable to other optimization criteria.

How does a typical synthesis system look like?

Traditionally the automatic synthesis process of digital systems is divided into several stages, starting from the high-level specification and ending with a working circuit. The specification is processed by a "pipeline" of synthesis and optimizations programs (we call them *processes*) (see figure 0.3). This is analogous to the process of compilation of, for example, *C*++ source code to a binary code for a given processor[AU77]. As one of the differences, the circuit synthesis process is often potentially interactive (at least at some stages). Most of the processes are able to perform all the steps without user's intervention.

Where does the problem lie?

The big advantage of the above described organization is that a complex synthesis task is divided into smaller subtasks, which are much easier to handle. If also the

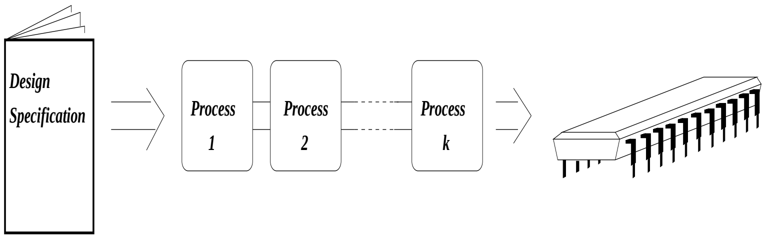


Figure 0.3. A pipeline of the synthesis tools

processes for the subtasks are independent from each other, we obtain the same result as if we were processing the whole task at once. Unfortunately this is usually not the case in circuit synthesis. Let us illustrate it with a small example.

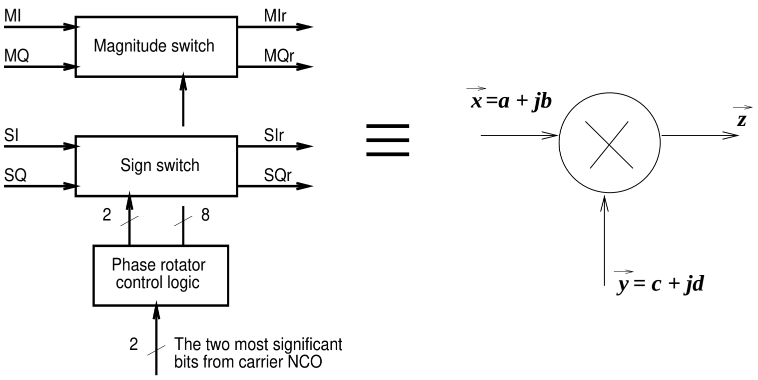


Figure 0.4. The phase rotator

Example 1 *We design a GPS correlation engine for the* GOLLUM *navigation system [EA92]. In figure 0.4 one part of it - a phase rotator is presented (we will refer to it later in this book). The rotator multiplies two complex vectors $\vec{x}$ (binary coded as signals MI,MQ,SI,SQ) and $\vec{y}$ (two most significant bits from carrier NCO). Both vectors are represented in rectangular coordinates. The output from the rotator is further used in an* integrate and dump unit *(not shown in the figure). That unit requires its input to be also in rectangular coordinates. The rotator can be implemented in the following two ways:*

1. *By using the formula:*

$$\vec{x} \cdot \vec{y} = (a + jb) \cdot (c + jd) = (ac - bd) + j(ad + cb)).$$

   This requires 4 multiplications and 2 additions.

2. *By first converting the rectangular coordinates representation into polar coordinates representation, then applying the formula:*

$$\vec{x} \cdot \vec{y} = \rho_1 e^{j\phi_1} \cdot \rho_2 e^{j\phi_2} = \rho_1\rho_2 \; e^{j(\phi_1+\phi_2)},$$

and then converting it back. This requires only one addition and one multiplication, but additional $rectangular \rightarrow polar$ *and* $polar \rightarrow rectangular$ *conversion units are required.*

The designer has to select one of these two implementations. As comparison criteria the number of transistors required or the time needed to process a pair of vectors could be used. Unfortunately these characteristics will only become available after the layouts for both versions of rotator are synthesized. Still a decision has to be made.

This kind of backward dependencies in the synthesis of digital systems appears unfortunately quite often. Given a design space we usually do not know the exact coordinates of all points (recall figure 0.1). Optimization on incorrectly specified design space may lead to very disappointing results. In the past people tried so solve this problem in the following ways:

1. *By re-designing a part or the whole circuit.* If at a certain design stage we discover that continuing the current line will not deliver a circuit, which satisfies the design constraints (for instance it will be too large to fit on a single chip), we can always go several steps backward and do everything again. This is often a time consuming process and it may be difficult to retain what is useful.

2. *By predicting the values of not yet know circuit parameters.* It is usually not easy to come up with good predictions. Prediction of interconnections length alone forms already a quite difficult problem [Don79][Sec87]. The result may be that the decision made on the basis of incorrect information may turn out to be wrong, and finally we may obtain a circuit which is unnecessarily inefficient or does not satisfy the design requirements. In most cases this also means time consuming redesign and iteration. Thanks to advances in computer technology and numerical methods it is nowadays often possible to obtain globally optimal solutions with respect to some cost functions. What is however the use of this potential in case that our predictions are quite far from real values?

3. *By using a knowledge database.* In this approach we try to use previously gathered knowledge, often gained from other designs. For instance, if we had already used phase rotators from example 1 in previous designs and had stored them in a database, we have easy access to all their characteristics. The database can also be built by a priori considering consequences of all possible design decisions. For example we could build beforehand several versions of every logic gate (with different shapes or driving capabilities). This opens possibility of applying the concepts of dynamic programming [Bel57]. They were successfully applied to floorplan optimization [Ott83],

but there additive property of the circuit cells' area could be employed. In performance optimization direct extension of the floorplaning algorithms does not lead to efficient programs. This is due to the fact that circuit performance is an *intensive* quantity. It is not additive (like area). Its computation requires analyzing the structure and behavior of the circuit. Another disadvantage of the method is that quite some work overhead may be required before the real design can start.

4. *Stochastic modelling.* Another approach could be to assume that all parameters can be represented as random numbers with given expected values and probability distributions and modify existing algorithms to take this random nature into account. There is no doubt that algorithms based on probabilistic information would be much more powerful. We can see here however several problems: good methods for generating these probabilistic distributions are needed and secondly, we should avoid increasing the, already high, computational complexity. Both problems are serious. Generating good probabilistic distributions of circuit parameters is, if feasible, difficult and expensive. Also, methods of probabilistic programming [Sen72] [SM84] do not seem to satisfy our needs: using them in combination with integer linear programming formulations means that only problems of very limited size can be solved. Finally, but maybe most important, the uncertainties in performance and other circuit characteristics are not really captured by randomness.

What do we want to show in this thesis?

In this thesis we would like to present two solutions to the problem of performance driven synthesis of digital systems. The first approach which we call here *the time-budgets* method belongs to the third category described above. It is aimed at removing the need for redesigns, but it cannot be applied at all the stages of the synthesis process. We propose therefore also another solution, which is wider applicable and more practical. It decreases significantly the risk that a redesign is necessary. The approach is based on possibilistic programming [Zad78], a branch of fuzzy mathematical programming. As the name says methods belonging to this category have their roots in the theory of fuzzy sets and solve optimization problems on fuzzy numbers.

Why using possibilistic programming?

We claim that there are several good reasons for using possibilistic programming at many stages of the automated synthesis process. Here follows the list of the most important ones:

1. Using the methods of possibilistic programming we decrease significantly the risk that a redesign is necessary.
2. Possibilistic programming provides good means to model performance imprecision.
3. The methods of possibilistic programming are flexible, making the process of adopting the existing synthesis algorithms easy.
4. Gathering possibilistic information is much easier than stochastic data, and fits better into the overall synthesis process.
5. The computational complexity of the possibilistic methods is often low.

We would like to emphasize that possibilistic programming should not be viewed as "cheap replacement" of stochastic programming. The possibilistic approach does not assume anything about probabilistic properties of the system. It is also not true, that formulating a possibilistic program we loose information. Usually probability distributions are not available, at least without additional (large) effort to calculate them. We assume that the only information to expect is something like possibility distributions (as will be introduced in chapter 3). Therefore we do not loose any information. We just start from different kind of information and use different (more suitable) its model and methods.

Organization of this thesis

First, in chapter 1 we will give an overview of past and present design automation systems. Four self contained **CAD** systems will be described (selected on the basis of local usage). They cover together a whole design trajectory starting from a description in high-level language to a working layout.

Chapter 2 is devoted to the issue of *design trade-offs*. It will be illustrated by the so called "high C" circuits (a transmitter-receiver pair sending a signal over high capacitance wire). Several "high C" circuits will be presented with different performance, power dissipation and area characteristics. After defining a delay model the circuits will be compared based on simulation results, and design *trade-off curves* will be generated. At the end we will list additional sources of the imprecision of circuit delay models. It will be argued that there are enough reasons to treat the problem seriously. This conclusion forms one of the motivations for the algorithms presented in this thesis.

Since most of the methods described in this thesis are based on *integer programming* (shortly **IP**), chapter 3 will survey some **IP** methods. After that, in the same chapter, the concepts of possibilistic programming will be described.

In chapter 4 our first solution to the problem of performance driven synthesis (called the *time-budgets method*) will be introduced.

The next chapter will be devoted to three successful applications of the possibilistic programming. We selected three subtasks belonging to different stages in the automated synthesis trajectory: the problem of *retiming*, the problem of *simultaneous scheduling, selection and allocation of functional units*, and the problem of *hardware-software partitioning of embedded systems*.

In chapter 6 the implementation details and experimental results for all proposed algorithms will be presented. In addition, we will describe our general **PIP/PLP** solver called FSOLVE.

Chapter 1

OVERVIEW OF AUTOMATIC CIRCUIT SYNTHESIS SYSTEMS

Contents

1.1 Introduction

In this chapter we will try to give an overview of past and present systems for automatic circuit synthesis. Four complete **CAD** systems will be described, covering the whole design trajectory starting from a description in high-level language to a working layout.

1.2 A historic perspective

First attempts to automate design of electronic circuits date already from 1950s [Qui52][McC56]. The complexity of the systems in these days was not large, the nature of the systems different. These programs and algorithms are therefore only of historical value to us.

Automatic layout systems started appearing in 1970s, and were oriented for gate-array and standard-cell design styles. First placement and routing algorithms were developed. With the advent of the **LSI**[1] chips the situation had changed. Many designers found entering large designs at the physical level tedious and error-prune.

[1]Large Scale of Integration

It resulted in a shift of interest from layout to logic view. In the early 1970s another important change in the attitude towards the automatic synthesis took place: people realized that previous exact optimization methods were useless in solving practical problems. The circuits became simply much too large. This awareness resulted in the development of many heuristic algorithms, which at the price of giving up the global optimality were able to cope with **LSI** [HCO74]. The theory of computational intractability was developed [GJ79]. First "serious" logic minimizers were developed, most of them for two-level logic forms, to be implemented in PLA style. As example we can give ESPRESSO from IBM and University of California at Berkeley [BHMSV84].

In the beginning of the 1980s, when the first **VLSI**[2] chips started to appear on the market, new problems had to be addressed. There was an increased need for multiple-level logic optimization tools. Second, representation of the circuits at even higher, architectural level became necessary. The former issue resulted in several multiple-level logic optimization programs, which were able to cope with large number of logic equations. As examples we can give YLE system (Yorktown Logic Editor) from IBM, or MIS (Multiple-level Interactive System) [BRSVW87] from the University of California at Berkeley. To address the second issue, a shift in the input specification for the synthesis programs was observed. Large **VLSI** chips could not be modelled anymore in terms of logic-level primitives. Also entering large circuits in schematic editors became impractical. This resulted in development of high-level description languages (as VHDL [Maz92] or Verilog [TM91]) and architectural synthesis techniques. Fully integrated systems targeted at instruction-set processor design became available, for example YORKTOWN SILICON COMPILER [BBC$^+$85] from IBM and the CATHEDRAL [JCV$^+$86] from the Catholic University of Leuven (Belgium).

Recently a lot of research on yet higher level synthesis systems is going on. In the design automation community they are known under the name systems for *embedded systems* design. *Embedded systems* could be defined as combinations of hardware with general purpose computational capabilities and more dedicated modules, together performing a function carefully partitioned in software and hardware to obtain the optimum trade-offs between the various performance metrics. These types of implementations have become increasingly popular as advances in ic-technology and processor architectures allow for flexible computational parts and high-performance modules integrated on a single carrier. An example of such system is the CASTLE hardware-software co-design system from GMD, Germany [TSV94].

[2] Very Large Scale of Integration

1.3 Selected systems overview

At this moment many commercial and research systems from various vendors and universities are available. We will concentrate on four typical ones: the CASTLE hardware-software co-design system from GMD, Germany, the OLYMPUS high-level synthesis system from Stanford University [KDM92], the SIS logic synthesis system from University of California at Berkeley [SSL+92] and the OCEAN *sea-of-gates* layout system from Delft University of Technology, the Netherlands [GS94]. The main criteria for choosing these systems was that they were available to the author and cover most of the features typical for nowadays state-of-art synthesis systems.

Very general design-flow, including all mentioned above design stages is presented in figure 1.1. In the sections to follow we will try to elaborate on all the components.

1.3.1 Hardware-software co-synthesis

The main objective of a design environment for HW/SW co-design is to help a designer in finding suitable and cost-effective structure of his system, carefully partitioned between software and hardware. The CASTLE system accepts as its input algorithmic specifications in C++ [Str91b], in VHDL [Maz92], or a mix of both. The design flow in CASTLE can be divided into two stages:

- **Design analysis.** During this stage the system collects static and dynamic information about given algorithmic description. The static information includes communication statistics for calling functions, amounts of data transferred between functions and dependencies between operations. The dynamic information is obtained by profiling the C++ code and/or simulating VHDL code. This results in execution times and frequencies of all operations. It is important that the profiling and simulation is performed with data that the designer believes to be relevant for his system.

- **Hardware-software partitioning.** In this stage the designer interactively partitions the systems between software and hardware, simultaneously trying different target architectures. The system helps by estimating the consequences of each partitioning step. Finally the VHDL code is generated for all operations assigned to hardware and C++ code for all operations remaining in software.

The generated C++ code is now compiled with a C++ compiler, while VHDL code forms an input to a high-level synthesis system, as described in next sections. As can be seen in figure 1.1 the whole process is potentially iterative. After synthesizing the hardware, both components are usually simulated in a hardware-software simulator, for example in PTOLEMY [BHLG94]. If the system does not satisfy the design constraints the whole process has to be repeated.

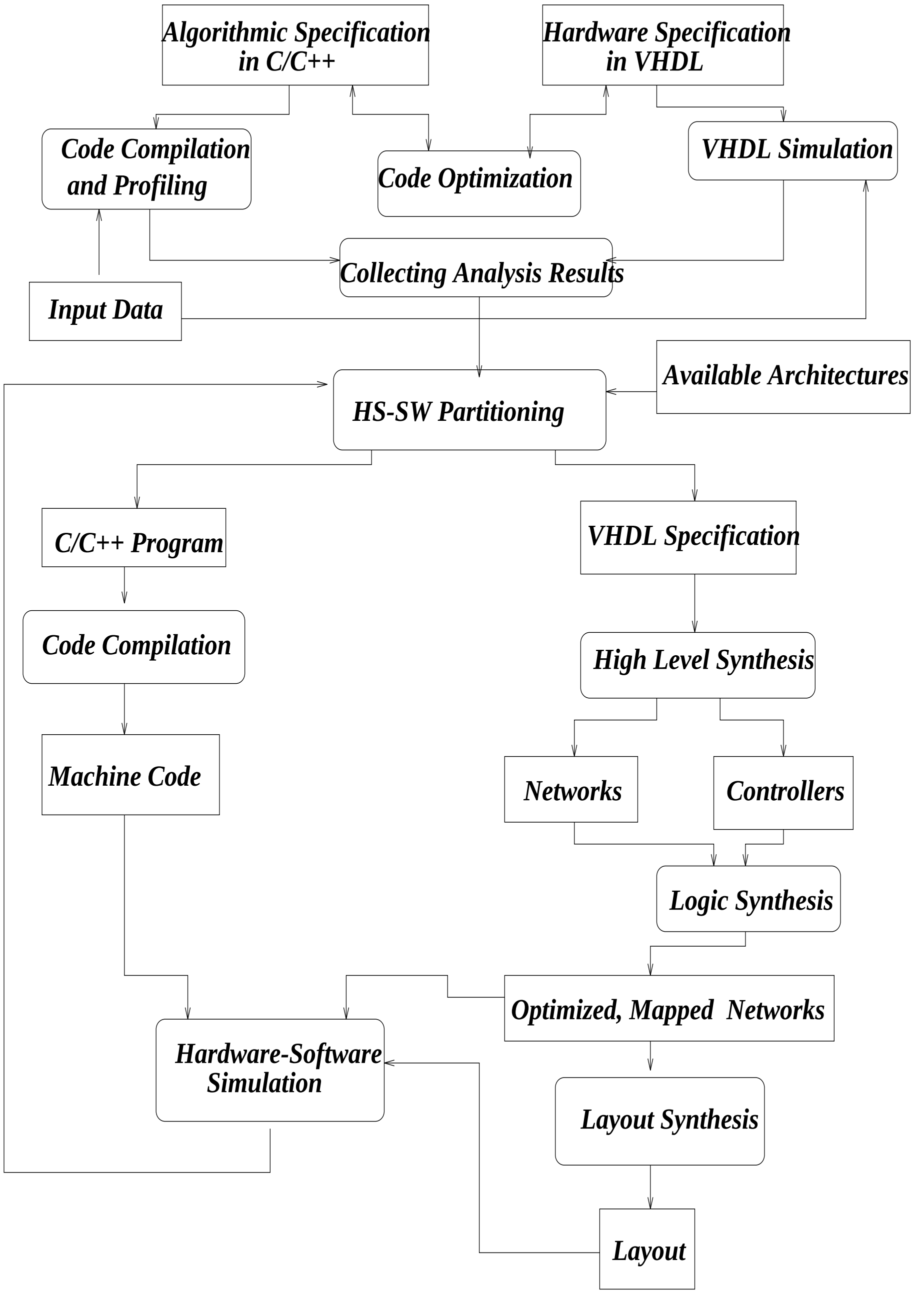


Figure 1.1. Very general design flow of the ICs

1.3.2 High-level synthesis

The high-level (architectural) synthesis starts usually from an algorithmic description in a high-level description language, for example in VHDL, or as in case of the OLYMPUS system, in *HardwareC* [KDM92]. *HardwareC* is an augmented version of ANSI C [KR88], which facilitates also hardware description. The synthesis is performed in two steps. First the front-end parsing and behavioral optimization is performed. This results in an implementation-independent graph-based description. In the second step the scheduling and binding is performed, producing a synchronous logic implementation. In OLYMPUS the functionality of the first step is implemented in a program called *Hercules*, while a program called *Hebe* does the second step. Here follows a list of synthesis steps and optimizations performed by *Hercules*:

- **expansion of selected model calls** - this enables optimizations across the call hierarchy,
- **mapping of selected operators to library models** - for example "+" can be assigned to a specific adder from the library,
- **for-loop unrolling**,
- **constant and variable propagation** - replacing a reference to a variable by its last assigned value,
- **common sub-expression elimination** - removing redundancy in operations,
- **dead-code elimination** - removing parts of code which do not have any influence on the behavior of the whole model,
- **data-flow analysis** - identifying the data and control dependencies among the operations,
- **collapsing conditionals** - the conditionals containing only combinational logic are collapsed to increase the scope in which logic synthesis can be applied,
- **combinational coalescing** - grouping together combinational logic operations into *expression blocks*. The *expression blocks* define the largest scope in which logic synthesis can be applied.

The result of all these translations and optimizations is a so called *sequencing graph*, see example in figure 1.2. The vertex set V of this graph is in one-to-one correspondence with the operations. The edge set E models the dependencies due to the data flow or serialization. In practice this graph is often hierarchical. In addition we have a set of functional resources (with area and delay information) and a set of design constraints. These three elements form an input to *Hebe*. The objective of the program is to explore the design space to obtain a suitable structure which satisfies all design constraints (area, timing). This is done in three stages:

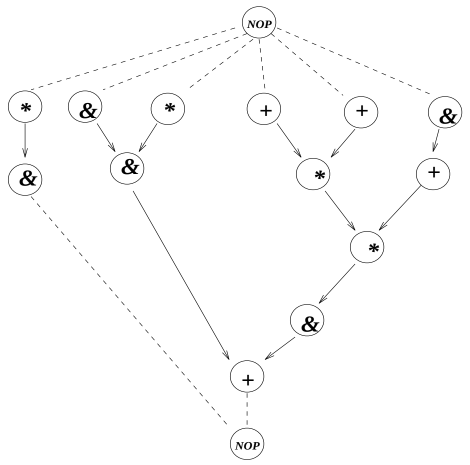


Figure 1.2. An example of a sequencing graph

- **Resource binding.** Based on cost criteria that are used to evaluate the design space, operations are assigned to available hardware resources. An example binding for the sequencing graph from figure 1.2 is shown in 1.3.
- **Resolving resource conflicts.** The operations which have been assigned to the same resource have to be serialized, to ensure that the calls do not simultaneously activate the same resource.
- **Relative scheduling.** Scheduling assigns operations to control steps. Figure 1.4 shows an example scheduling of the sequencing graph from figure 1.2. Since the sequencing graph is free from resource conflicts this can be done exactly and efficiently. A complication arises from the fact that some operations may have unbounded execution delays. A technique called *relative scheduling* is used [KDM90] to address that problem.

The approach in the OLYMPUS system differs from the approach in the OASIC system [GE90], which performs the binding and scheduling simultaneously. In addition in the OASIC system, the global optimality of the generated structure is guaranteed.

Finally logic description of both the datapath and the control logic is generated, to be used by a logic synthesis system.

1.3.3 Logic synthesis

The logic synthesis usually starts with the STG (State Transition Graph) specification, for example in kiss format [Lis89]. In figure 1.5 an example STG is presented,

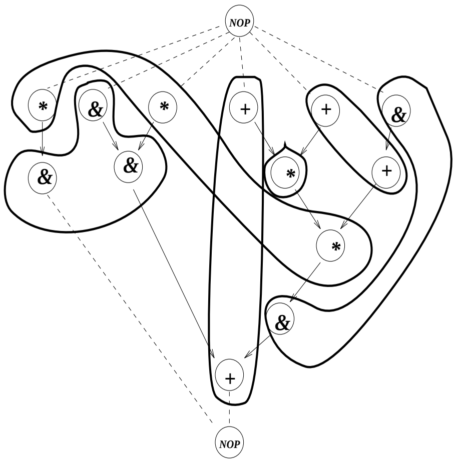


Figure 1.3. An example binding. Operations which are assigned to the same resource are placed within ovals.

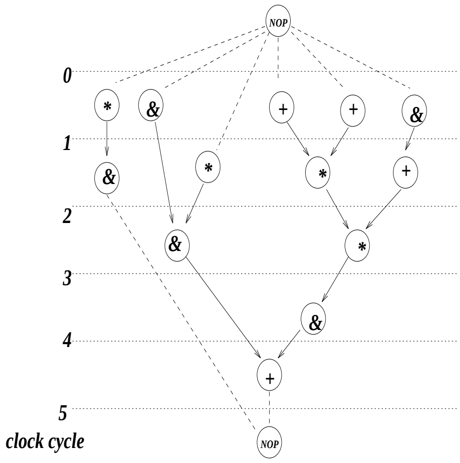


Figure 1.4. A schedule

in figure 1.6 its description in that format. Usually the following design steps are performed:

- **State minimization.** The input STG often contains some states that produce the same outputs under the same inputs. This can be exploited to generate another machine with fewer states [HRSJ91].
- **State assignment.** Initially all states are given symbolic names. These names have to be replaced by binary codes in an optimal way. For example to obtain minimal number of registers or low power consumption [BDM94].
- **Combinatorial optimizations.** This step has many substeps, all combined to find a logic representation with a minimal number of implicants and literals while preserving the functionality. For overview of these methods see [BHMSV84]. Most of the programs use so called **BDD**'s (Binary Decision Diagrams) [Ake78] [Bry86], which make efficient storage and manipulation of boolean functions possible.
- **Technology mapping.** All previous optimizations are made at the technology independent level, using very simple models of the logic elements. Now the logic functions have to be mapped into the real gates specified in a technology library. Usually this is done using tree-covering algorithms [Tou90]. Frequently the optimization criteria are the minimum area or minimum total delay. Recently also attempts to map for low power were reported [TAM93] [TPD93].

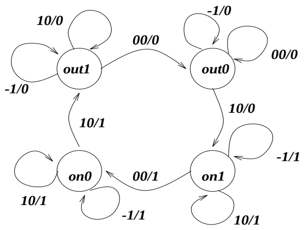


Figure 1.5. An STG - the hotel switch example

```
.model hotel
.inputs in1,in2
.outputs out
.i 2
.o 1
.s 4
.p 12
.r out1
-1 out1 out1 0
10 out1 out1 0
00 out1 out1_0 0
-1 out0 out0 0
00 out0 out0 0
10 out0 on1 0
-1 on1 on1 1
10 on1 on1 1
00 on1 on0 1
-1 on0 on0 1
00 on0 on0 1
10 on0 out1 1
```

Figure 1.6. A kiss2 code description. After definitions of inputs, outputs, and reset state (.r statement) the list of all transitions follows.

- **Restructuring for performance.** After technology mapping, we have already a legal mapped network, which however may violate design constraints (i.e. delay, power, area). Several transformations can be applied to improve the network:

  1. retiming [LRS83],
  2. peripheral retiming [MSBSV91],
  3. collapsing sections along the long paths and resynthesizing them to reduce the depth of the circuit (assuming that smaller depth represents smaller delay) [SWBSV88],
  4. clustering nodes according to some criteria and then collapsing each cluster into a single node [TSB91].
  5. repowering - only changing gate selection, structure remains the same,
  6. applying buffering to the multi-fanout nodes,
  7. dualizing - replace a gate by its dual.

1.3.4 Layout synthesis

In this section we will concentrate on the OCEAN sea-of-gates design system. Sea-of-gates is a special design style where we deal with a prefabricated transistor pattern on the chip. To Implement a circuit, a designer only has to interconnect the transistors with metal wires. The sea-of-gates style has several advantages:

- Because the chips are prefabricated and only metal layers have to be layed down a short fabrication time results.
- Wavers with the prefabricated image may be initially produced in large quantities, what dramatically reduces chip cost.
- Because the designer does not have to spent time designing all masks (he even does not have to know much about process design) a dramatic speedup in design time occurs.
- There are no restrictions on routing area, what makes implementation of structured logic (RAM, PLA, etc.) possible.

The OCEAN system has the following features:

- It accepts network descriptions from the following synthesis systems: SYNOPSIS, CADENCE, EUCLID, SIS, EDIF.
- Works with arbitrary sea-of-gates images, for example with 1.6μ CMOS *fishbone* process [Phi88] (figure 1.7), 0.8μ CMOS *octagon* process, even with traditional 4μ Philips *gate-array*.
- It supports hierarchical (full-custom-like) layout style on sea-of-gates.
- Allows for combination of any automatic and interactive manual layout.
- All programs can handle even the largest designs.

A block diagram of the OCEAN system is shown in figure 1.8. Let us shortly describe the design trajectory in OCEAN.

1. We start with a network specification in blif or sls format. This may be automatically generated by a logic synthesis system, or entered by hand in the schematic editor called SCHERZO [Kar94].
2. Now usually placement and routing takes place. Placement is performed by the MADONNA placer [Str91a] [Kar91]. After that the TROUT router [Gro91] makes interconnections.

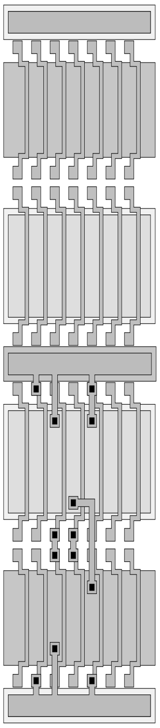


Figure 1.7. A 2-input nand gate on fishbone image

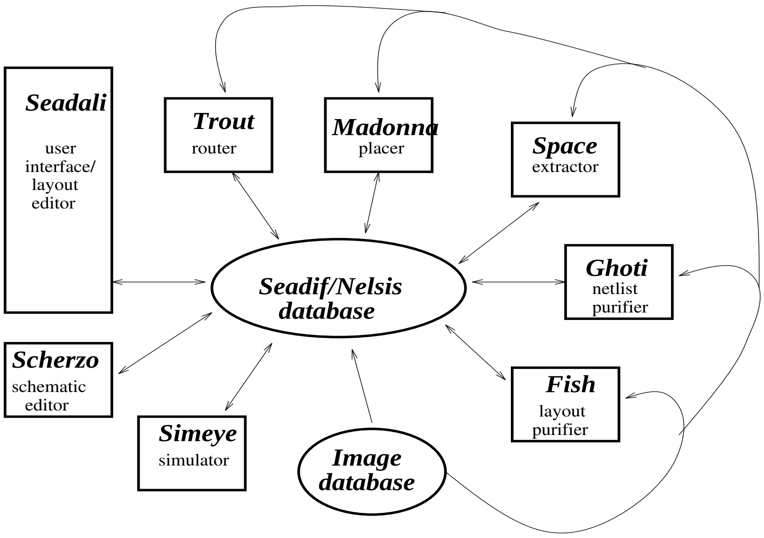


Figure 1.8. The OCEAN sea-of-gates design system.

3. The results can be immediately examined in the SEADALI layout editor/viewer, which serves also as a general user's interface in OCEAN. Also some changes can be made by hand. The correctness of the layout can be checked by the FISH layout purifier [GS91].

4. We can extract a network from the existing layout using the SPACE [vGvdMB95] layout extractor. Superfluous (unconnected) transistors can be removed with the GHOTI netlist purifier [GS91].

5. Both extracted and original networks can be simulated in the *Simeye* simulator, which provides an interface to the SLS switch level [vGdG87] simulator or to SPICE [VZN$^+$81] (the only non-Delft tool).

All these steps result in a nice, perfectly working chip, for example as the one shown in figure 1.9.

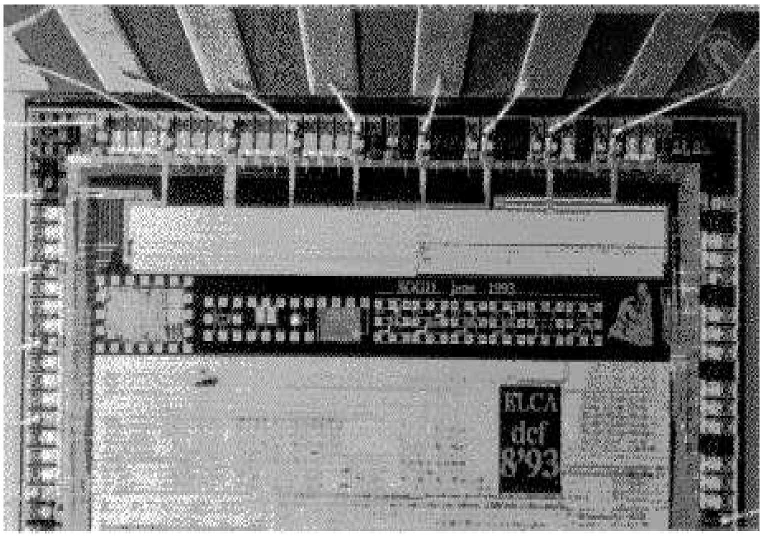


Figure 1.9. Microphotograph showing one of the many chips which were made with OCEAN

Chapter 2

DELAY MODELS AND DESIGN TRADE-OFFS

Contents

2.1 Introduction

Most design automation programs accept as their input the following three kinds of data: information about the structure of the problem (for example in form of data flow graphs, boolean expressions or network description), information about design alternatives and information about design constraints. In this chapter we will concentrate on the design alternatives, usually called *design trade-offs* [Sin92], [SDM91]. They appear often in the form of so called *trade-off curves* (figure 0.1 from the preface). Also our algorithms described in chapters 4 and 5 will use them as input data. The *trade-off curve* expresses the relation between selected figures of merit at the Pareto points. In our case, since our primary concern is the performance of the digital system, it will always represent the *trade-offs* between time delay and some other criteria, for example area, power, noise margin, number of pins, etc.

The main purpose of this chapter is to introduce the reader to the concepts of design trade-offs. This however requires defining first the time delay models. Therefore sections 2.2 and 2.3 are devoted to models of delays in cells and interconnections. Models of the chip area or power dissipation are relatively simple. This is not the case with the circuit performance (for example speed). It is not additive (like for example area). Its computation requires analyzing the structure and behavior of the circuit. Many different time delay models have been proposed in the past. The more elaborate ones can offer higher precision, but unfortunately, usually increase the complexity of used analysis and synthesis methods, making them impractical even for very small examples. This is one of the reasons why in practice rather simple delay models are used. Moreover, circuits are usually not available in the early stages. Most of their parameters are difficult to predict. Using uncertain values in detailed models does not make much sense. These parameters may depend on many factors, as for example the final chip processing, also on logic and layout synthesis methods used. Throughout sections 2.2 and 2.3 some simple numerical examples will be presented. They serve two purposes. The reader should get an impression of the order of magnitude of some of the model's components. Secondly, they introduce a real-life technology - the C3TU process [Phi88]. All circuits presented in this chapter are implemented on that process.

After that, to demonstrate the issue of the design-tradeoffs we will concentrate on so called "high-C" circuits (a transmitter-receiver pair sending a signal over high capacitance wire). Most presented circuits were taken from the book [Bak90]. The circuits will be shown to have different performance, power dissipation and area characteristics. They will be compared based on simulation results, and design *trade-off curves* will be generated.

Simultaneously we will try to direct the reader's attention to approximate character of these curves. A fact that a given circuit represents a Pareto point in a *trade-off curve* does not mean that it is an absolute Pareto point. What we only can state is that a better solution is not known, or has not been considered. A Pareto point is invalidated if we come out with another, better circuit. Current sense amplifier circuit will be used to demonstrate this issue. The circuit will be shown to have such superb characteristics that it invalidates nearly every Pareto point in previously generated curves. To put things into perspective, in section 2.5 selected other design problems, for which several implementations having different delay-area-power characteristics are available, will be presented.

At the end of this chapter we will try to concentrate on another, from our point of view very important , issue - the imprecision of circuit delay models. It will be shown that there are enough reasons to treat the problem seriously. This conclusion forms one of the motivations for the algorithms presented in this thesis.

2.2 Preliminaries

For the purpose of fair comparison of the “high C” circuits (section 2.4), in this section we will define some important notions - the time delays of the cells and their noise margins.

Imprecise definitions of the cell delays may be a source of many ambiguities. The imprecision may for example lay in not specifying the input threshold voltage (it differs among cells), or ignoring the fact that cell delay time is dependent on the input signal slew rate. The first problem is solved by assuming that the input threshold voltage of all cells is $V_{dd}/2$. In addition it is assumed that the input signal rate of change is more than a certain, technology dependent value. Finally, the input signal mentioned in the definitions, is monotonously rising or falling. This simplifies modelling and optimization. The definitions below follow these assumptions:

DEFINITION 1 *The maximum time interval between the moment when the input signal reaches 50 % of its maximum voltage swing and the moment when the output signal passes 50 % point will be referred as the* delay *of a cell,* $t_{50\%}$.

Further we could also define separate delays for the case when the output of a gate is rising or falling:

DEFINITION 2 *The time interval between the moment when the input signal reaches 50 % of its maximum voltage swing and the moment when the rising (falling) output signal passes 50 % point will be referred as the* rise (fall) delay *of a cell,* $t_{rise}(t_{fall})$.

Obviously the *delay* of a cell ($t_{50\%}$) is a maximum of t_{rise} and t_{fall}. These definitions are illustrated in figure 2.1.

Another notion that needs to be defined is the *noise margin* of a cell. It represents the allowable noise voltage at the input of a cell, which does not affect the output. The following definition follows after [WE92]:

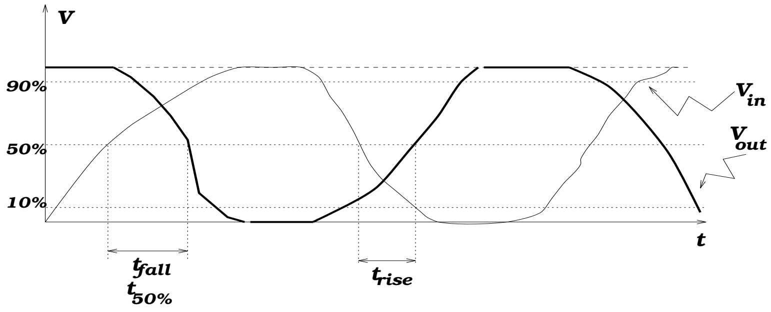


Figure 2.1. Illustration of definitions of $t_{50\%}$, T_{rise} and t_{fall} delays

DEFINITION 3 *The LOW noise margin, the HIGH noise margin and the noise margin are:*

$$NM_L = |V_{ILmax} - V_{OLmax}| \tag{2.1}$$
$$NM_H = |V_{OHmin} - V_{IHmin}| \tag{2.2}$$
$$NM = \min(NM_L, NM_H) \tag{2.3}$$

where:

V_{IHmin} *= minimum HIGH input voltage*
V_{ILmax} *= maximum LOW input voltage*
V_{OHmin} *= minimum HIGH output voltage*
V_{OLmax} *= maximum LOW output voltage*

These definitions are illustrated in figure 2.2. In case of CMOS circuits the value of $V_{OLmax} = V_{ss} = 0V$ and V_{OHmin} is equal to the supply voltage $V_{dd} = 5V$.

2.3 Delay model

2.3.1 A configuration with two gates

A typical **CMOS** logic gate is composed of a series of complementary MOS transistors. As an example, in figure 2.5 a static nand CMOS gate is presented. In practice a gate can contain many parallel-series transistors. Depending on the logical values of the inputs some of transistors are conducting current from source or from drain to the output of the gate. This current charges or discharges internal capacitances of the transistors in the gate itself, of the interconnections and at the inputs of next logic stage.

In the simplest case one transistor is charging a single gate. Such configuration is shown in figure 2.3. Only a part of both gates is drawn - the pullup transistor of gate 1 and an input transistor of the gate 2. This circuit is modelled by a RC network with an ideal switch as shown in figure 2.4.

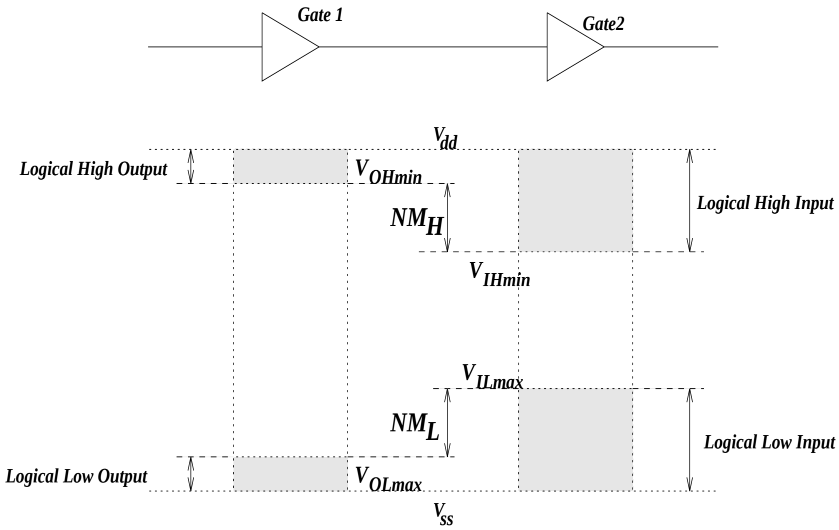


Figure 2.2. Illustration of definitions of NM_L and NM_H

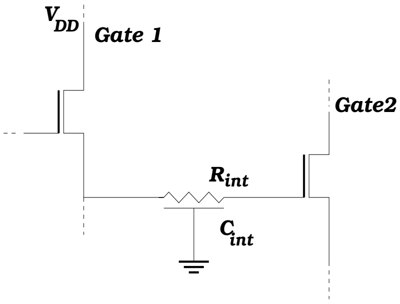


Figure 2.3. A simple network

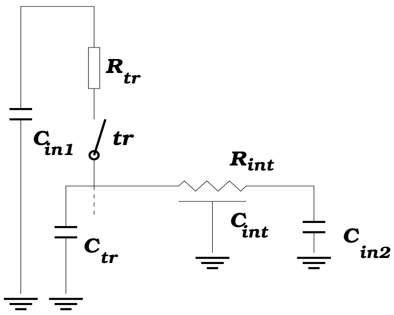


Figure 2.4. A RC network model

To calculate the delay of the circuit several components, like the on-resistance of the transistors R_{tr}, internal capacitance of the transistor C_{tr}, the resistance and capacitance of the interconnections R_{int}, C_{int}, and input capacitances of the fanout

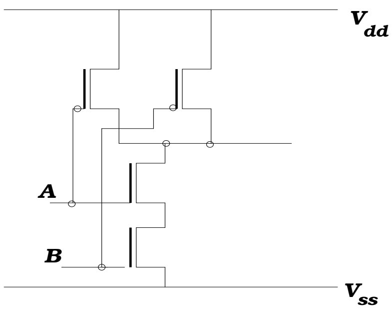


Figure 2.5. A two-input CMOS nand gate

transistors C_{in} have to be taken into account. The simplified Sakurai's formula [Sak83] for the 50% delay from the first to the second gate is:

$$T_{50\%} = 0.4 R_{int} C_{int} + 0.7(R_{tr} C_{int} + R_{tr} C_{in2} + R_{int} C_{in2}) \tag{2.4}$$

Let us now concentrate on all the components.

2.3.2 The transistor resistance

The transistor on-resistance can be derived from the general model of the MOS transistor and changes depending on its working point. The average value of the R_{tr} can be expressed in the integral:

$$R_{tr} = \int_{V_1}^{V_2} \frac{dV}{I} \tag{2.5}$$

where V_1 and V_2 are boundary values of the voltage range that we average over and I is the transistor's current (being function of V).

In practice the following formula is often used to approximate this value (transistor in the linear region):

$$R_{tr} \approx \frac{L/W}{\mu C_{gox}(V_{GS} - V_T)} \tag{2.6}$$

where L is the length of the active transistor area, W its width, C_{gox} gate capacitance per unit area, μ the effective mobility of the carriers, V_T the threshold potential and V_{GS} the gate-source voltage.

Example 2 *Consider a simple inverter from figure 2.7. We can measure its output resistance as a function of input voltage. The result of simulations with* Spice *(C3TU process [Phi88]) are presented in figure 2.6. We take $L = 1.6\mu, W_N = 5.1\mu, W_P = 13.3\mu, V_{TP} = -1.1V, V_{dd} = 5V$. As can been seen, according to our expectations, the value of R_{out} is not constant but it changes from $1.7k\Omega$ to $7.3k\Omega$.*

We can also use the formula 2.6 to calculate the value of R_{out}, for example for input voltage $V_{in} = 0.5V$. In this case the p-transistor is in the linear region while the n-transistor in the cutoff region ($I_{dsn} = 0$). The on-resistance of the p-device should be therefore the same as measured before output resistance of the inverter for this voltage. We take again data for the C3TU process: $KP_P = \mu C_{gox} = 28.9 \frac{\mu A}{V^2}$:

$$R_{tr_P} = \frac{L/W_P}{\mu C_{gox}(V_{GS} - V_T)} = \frac{1.6\mu}{13.3\mu \; 28.9\mu \; 1.6} = 2.6k\Omega$$

This value is very close to the value of $2.7K\Omega$ from figure 2.6.

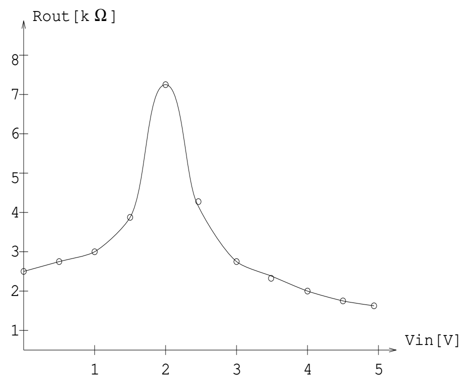


Figure 2.6. The output resistance of the inverter as a function of input voltage

It is not really clear which value of the on-resistance should be selected as the R_{out} to be used in our model. One possible way is to determine the value of the equivalent R_{eq} resistance in the circuit 2.8 in such way that the resulting delay until the voltage reaches a defined voltage is the same as the circuit from 2.7. As such defined voltage we can for example use 3.15V (which is 63% of the V_{dd}). In figure 2.9 the input and output signals of the circuit from 2.7 are presented. The time that it takes for the output to reach the value of 3.15V is $t_d = 3.1ns$. Now we can calculate:

$$R_{eq} = \frac{t_d}{1pF} = 3.1k\Omega$$

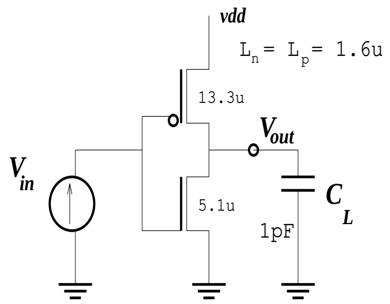


Figure 2.7. An inverter driving a single capacitive load

2.3.3 Gate input capacitance

Another important factor (dominating total capacitive load for short interconnections) is the capacitance of the inputs of all the cells which are fed by the output transistor. This capacitance is proportional to the dimensions of the transistor. The following expression gives its value:

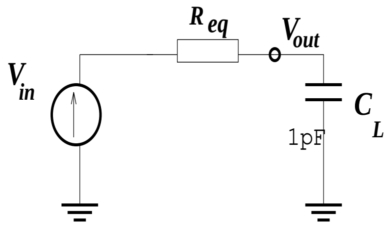


Figure 2.8. An equivalent RC circuit

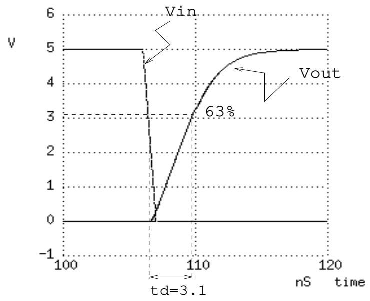


Figure 2.9. Waveforms at the output of the inverter

$$C_g = \varepsilon_{ox} \frac{WL}{t_{gox}} \tag{2.7}$$

where W, L are the width and length of the gate, t_{gox} the gate oxide thickness and ε_{ox} the dielectric constant. In case of complex cells the capacitances of all gates connected to a particular input should be added.

Example 3 *Let us calculate the input capacitance of the inverter from figure 2.7. It is a sum of the gate capacitances of the n and p transistors. Using formula 2.7 for the C3TU process [Phi88] we obtain:*

$$C_g = C_{gn} + C_{gp} = \varepsilon_{ox} \frac{W_n L_n + W_p L_p}{t_{gox}} =$$

$$= 3.45 \cdot 10^{-11} \frac{F}{m} \frac{1.6\mu m\ 5.1\mu m + 1.6\mu m\ 13.3\mu m}{250 \cdot 10^{-10} m} \approx 40pF$$

2.3.4 Interconnections

Resistance of a wire is inversely proportional to the cross-sectional area of the conductor, while the capacitance is proportional to the plate area and is inversely proportional to the distance between the plates. Exact modelling of the wire's capacitance is quite complicated. This is due to the fact that besides the standard, so called parallel plates, component, also other components have to be taken into account (the fringing field and the inter-wire capacitance). In practice however, the following approximate formulas giving the wire capacitance and resistance per unit length are used:

$$C_{int} = \varepsilon_{ox} \frac{W_{int}}{t_{ox}} \tag{2.8}$$

$$R_{int} = \rho_{int} \frac{1}{W_{int} H_{int}} \tag{2.9}$$

where ε_{ox} is the dielectric constant of the insulator and ρ_{int} is the resistivity of the conductor. Other parameters are illustrated in figure 2.10.

In practice, in technology books, the resistors are characterized by their sheet resistivity (given in $\Omega/\Box$). Then the resistance is given by the following formula:

$$R_{int} = R_{\Box} \frac{l}{W_{int}} \tag{2.10}$$

where l is the wire length.

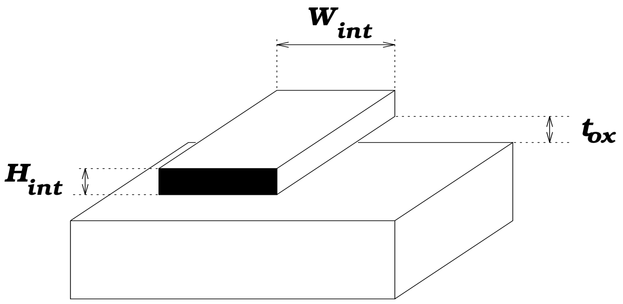


Figure 2.10. A wire segment

Example 4 *Consider an interconnection in* Metal1 *layer, which has length of $l = 7.5mm$. For C3TU [Phi88] process we have $W_{int} = 2.4\mu m$, $R_{\Box} = 45\frac{m\Omega}{\Box}$:*

$$R_{int} = \frac{45m\Omega \; 7.5mm}{2.4\mu m} \approx 140\Omega$$

The wire resistance and capacitance have a distributed character, yet they are sometimes modelled by a lumped RC network as shown in figure 2.11. The response of such lumped network is:

$$U_{out}(t) = 1 - e^{-t/RC} \tag{2.11}$$

As can be seen in figure 2.12 it differs from the response of the distributed RC network. Nevertheless it is quite commonly used. Also more elaborate RC networks as that shown in figure 2.13 can be used.

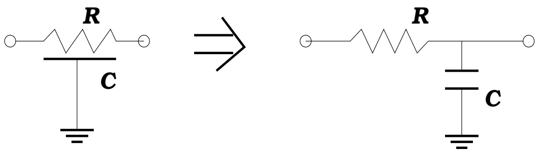


Figure 2.11. Distributed and lumped RC networks

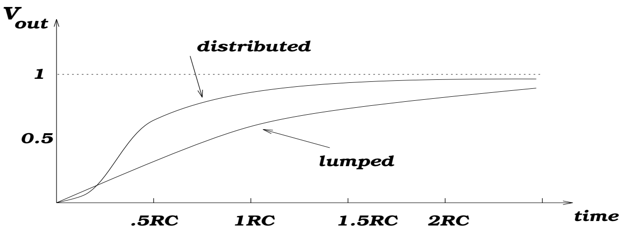


Figure 2.12. Comparison of responses of distributed and lumped RC networks

Now we already have a model for a simple two terminal interconnection. This is however not enough. In practice many multi terminal interconnections exist. They form so called fanout networks, or fanout trees (see figure 2.14).

Computing exact delay of such interconnections is very difficult because it requires solving a set of differential equations for the distributed RC sections, and no closed form time-domain solution exists. Instead we can approximate it by a RC tree network with discrete elements. For example for the fanout tree from figure 2.14 we get the network in figure 2.15. Every wire segment is modelled by a distributed or lumped resistor and capacitor.

Let us assume that we have n resistors and capacitors. Let C_i be the capacitance of node i, R_i the resistance of branch i. Let R_{jk} be the sum of all resistances common to the path from root to node i and from root to node j. For example $R_{45} = R_1 + R_3$ (see figure 2.15). The delay from the root to sink i is a sum over all sections s in the tree:

$$T_i = \sum_s R_{si} C_s \tag{2.12}$$

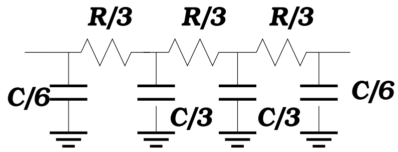


Figure 2.13. The β_3 lumped circuit approximation for a distributed RC line

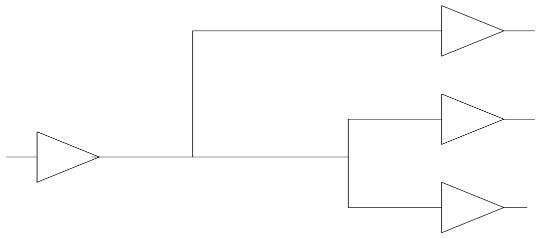


Figure 2.14. A fanout network

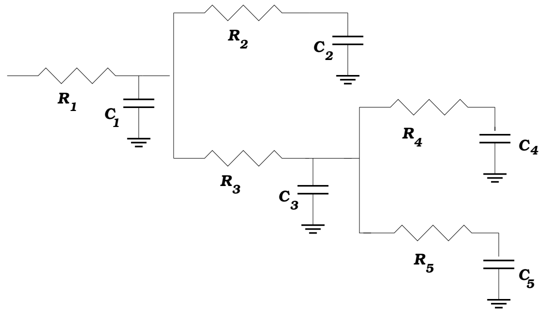


Figure 2.15. RC tree with discrete elements

2.3.5 Macro cells

In the previous section we discussed modelling of delays in circuits containing logic gates. Most of that applies for also more general cells, here called macro-cells (see example in figure 2.16). A macro-cell can be seen as a "black-box" having a certain number of inputs and outputs. The inputs and outputs will be electrically connected to some transistors within the cell in the same fashion as with gates described in previous section. Therefore all the equations giving the delay between the macro-cells still apply. There appears one additional component - internal delay of the cell itself. Therefore a delay along any internal path of the macro-cell can be given as:

$$D_t = D_b + D_o = D_b + FL \tag{2.13}$$

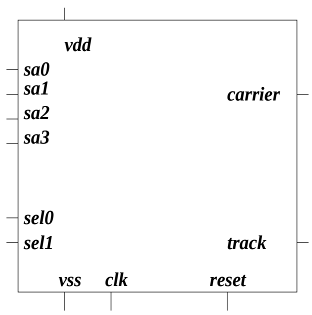


Figure 2.16. A macro-cell

where D_b is the inherent delay, D_o the fanout delay, F the drive coefficient (given in time units per unit load) and L the fanout load. The two last parameters are a linear approximation of all fanout delays. This model is simple and often used in practice by synthesis systems. For example in SIS system [SSL$^+$92] all cells are modelled in this way, also simple logical gates. An example definition of a 4-input multiplexer on a *Sea of Gates* chip [GS93] in **Genlib** (SIS) format is presented in figure 2.17. As can be seen the format distinguishes fall and rise delays. In many other systems only one (worst case) of these two values is used.

```
# Process  fishbone
# Lambda   0.2
# Made by Patrick Groeneveld and Irek Karkowski
# sis-genlib description for the libraries of august 1993
#
# time units used   1 unit == 1 ns
#
# chip area  used   1 unit == 1/2 of image's basic cell
#
# capacitance units : 1 pF == unit of fanout
#
#    cell        pin id        inp. max   rise-delay  fall-delay
#    name     function type    load load block fanout block fanout
GATE mu210 16 Y=A*!S1*!S2+B*S1*!S2+C*!S1*S2+D*S1*S2;
                 PIN A NONINV  0.12 3.0  1.30   1.3   2.10   1.4
                 PIN B NONINV  0.12 3.0  1.30   1.3   2.10   1.4
                 PIN C NONINV  0.12 3.0  1.30   1.3   2.10   1.4
                 PIN D NONINV  0.12 3.0  1.30   1.3   2.10   1.4
                 PIN S1 UNKNOWN 0.36 3.0 1.30   1.3   2.10   1.4
                 PIN S2 UNKNOWN 0.24 3.0 1.20   1.3   0.86   1.4
```

Figure 2.17. A *Genlib* description of a 4-input multiplexer

2.4 The "high C" circuits - design trade-offs example

The design of a digital system is a process in which the designer selects a proper implementation for all system components. The implementations can be compared based on some criteria, like for example area, power, timing. As an example in this section we will present different implementations of so called "high C" circuit - the transmitter-receiver pair sending a digital signal through a high capacitance interconnection (see figure 2.18). The working principles, weak and strong points of all circuits will be referred. Simultaneously we will present results of simulations performed with the SPICE3E2 [VZN$^+$81] program. At the end different *trade-off curves* for the problem will be presented. They can be later used by the design automation software.

We consider the following five comparison criteria:

criterion	symbol
total 50% delay from $V_{in}(V_{CLKrel})$ to V_{out}	T_p
total power dissipation	P
noise margin voltage	NM
maximum current	I_{mx}
relative area (reference A_{CASCA})	A

where V_{CLKrel} is the most relevant clock edge (in case of clocked circuits). To obtain a fair comparison, in all circuits the same values of parameters are used. They are listed in table 2.1. The basic configuration is shown in figure 2.18. When

Parameter	Description	Value
C_{int}	interconnection line capacitance	$10pF$
R_{int}	interconnection line resistance	$\approx 0\Omega$
C_i	input capacitance of the min.size gate	$40fF$
R_{out}	output resistance of the min.size gate	$3.1K\Omega$
C_{out}	output capacitance of the min.size gate	$150fF$
C_L	load capacitance	$200fF$
V_{dd}	drain voltage	$5V$
V_{ss}	source voltage	$0V$
L	transistor length	1.6μ

Table 2.1. Selected circuit parameters used in all simulated circuits

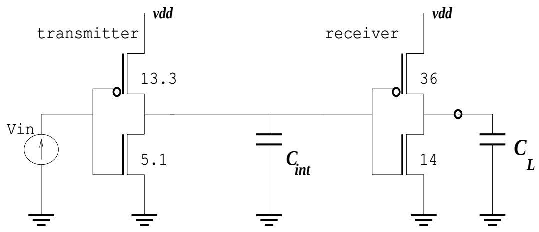


Figure 2.18. The basic configuration

comparing the size, the cascaded drivers circuit from section 2.4.2 is used as a reference. The sizes of transistors in all circuits were optimized to obtain minimum delay. The circuits were simulated in configuration shown in figure 2.20. The 50% delay was measured from the node V_{in} to V_{out}, or from the most relevant clock edge V_{CLKrel} to V_{out} in case of clocked circuits. The input inverter and output load were selected in such way that they represent typical values of capacitances. Input and clock waveforms used are presented in figure 2.21. All rise and fall times are equal $0.5ns$. The values of the parameters shown in table 2.1 have been obtained by simulating typical gates with SPICE3E2 (recall examples 2 and 3). For simulations we used the "typical" (C3TU process) MOS transistors model (given in figure 2.19). For very small transistors of the input inverter a slightly modified model was used. The difference lies however only in one parameter (we add `lambda=0.01`). The

```
*Typical - model
.model  Nenhtyp  NMOS(level=2 ld=0.325u vto=0.7 ucrit=24.3k
+ vmax=54k nfs=0e11 cjsw=620p pb=0.68v tox=250e-10 uo=510
+ delta=0.4 neff=4.0 js=2u mj=0.53 cgdo=320p  nsub=2e16
+ uexp=0.22 xj=0.4u rsh=.55 cj=130u mjsw=0.53 cgso=320p)

.model  Penhtyp PMOS(level=2 ld=0.3u   vto=-1.1 ucrit=51k
+ vmax=47k nfs=0e11 cjsw=590p pb=0.78v tox=250e-10 uo=210
+ delta=0.4 neff=0.88 js=10u mj=0.46 cgdo=320p nsub=5e16
+ uexp=0.33 xj=0.5u rsh=.75 cj=490u mjsw=0.46 cgso=320p)
```

Figure 2.19. MOS transistors models used in SPICE.

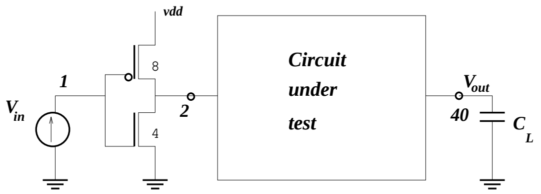


Figure 2.20. Test configuration used for all circuits

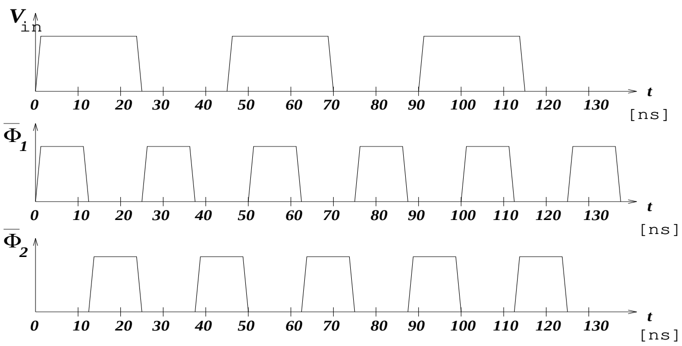


Figure 2.21. Input and clock waveforms used during simulations

noise margin NM (as in definition 3) was measured in the same simulation configuration. We take $V_{OLmax} = 0V$ and $V_{OHmin} = 5V$. Very slow saw input signal was applied to the input of the circuit. The LOW and HIGH *noise margins* were measured, then minimum of these two was used as the NM. An example waveforms during such measurement can be seen in figure 2.22.

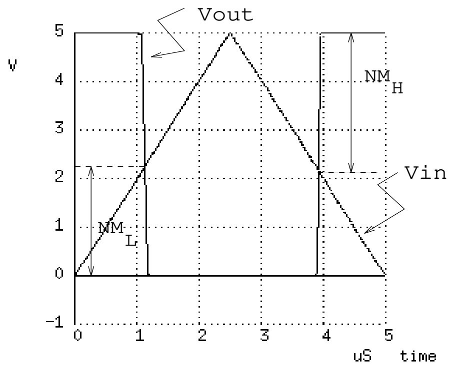


Figure 2.22. Waveforms during the noise margin measurement

2.4.1 Transistor sizing

By increasing dimensions of the output transistor[1] we increase the driving capability of it. The method has however some limitations:

- Increasing size of the transistors, we increase the area required to implement the circuitry.
- Power consumption grows.
- Larger size of the transistors causes the input capacitance to grow. As result while the delay at this stage decreases, the total delay in the path may remain constant or even increase, because the transistors of the previous stages have to drive larger load.

The simulated circuit called TRSIZ is presented in the figure 2.23. The limitations of the method can be clearly seen in figure 2.24. We present there the results of simulations for different driver size. The Pareto points are connected with a line. As can be seen, the minimum delay through the circuit is about $5ns$ and cannot be further improved by increasing the driver size. The reason for it can be easily seen in figure 2.25. Because of the large input capacitance of the driver, the waveform at node 2 becomes very slow and participates most in the total delay T_p. This is exactly opposite to the situation when the driver has minimum size (see figure 2.26).

[1]this method is sometimes also called “powering up the circuit”

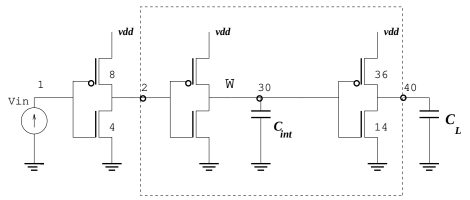


Figure 2.23. Simulated transistor sizing configuration (TRSIZ circuit)

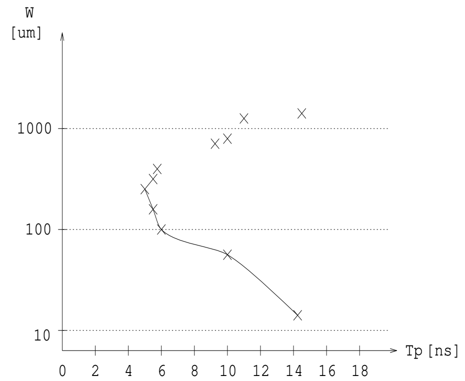
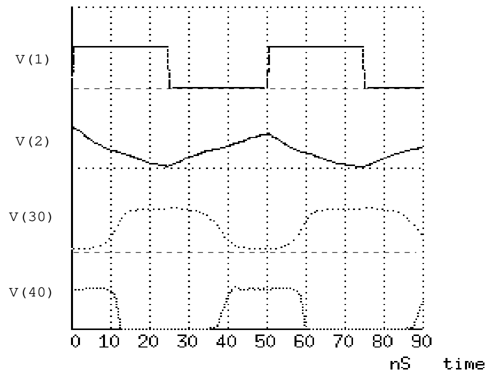


Figure 2.24. Results of simulations for different sizes of the driver in TRSIZ

In that case the node 30 constitutes most of the total delay. The delay is minimal for the driver size $W = 252\mu$. This configuration was simulated in detail. The table 2.2 shows the results.

There exist many algorithms, which can be applied to find optimal transistor sizing in the whole circuit. The simplest are based on the Newton's method [RPG76]



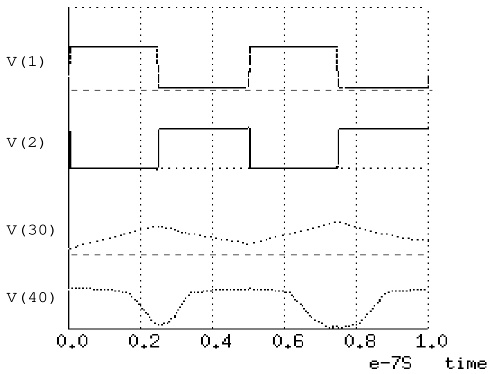


Figure 2.25. Waveforms in the TRSIZ circuit for $W = 1476\mu$

Figure 2.26. Waveforms in the TRSIZ circuit for $W = 18\mu$

$T_p[ns]$	$P[mW]$	$NM[V]$	$I_{mx}[mA]$	$A[1]$
5.1	8.2	2.0	13.5	0.47

Table 2.2. Simulation results for the TRSIZ circuit with driver having $W = 252\mu$

and have some convergence problems. Heuristic approaches have also been tried [Kao85]. The methods of [FD85] and [SSVFD88] are based on the mathematical programming but because of the size problems, they give up the global optimality and resort to iterations. Also non-linear programming has been tried, again with limited applications [Mar89]. Finally, in [BJ90] an approach based on linear programming has been presented.

2.4.2 Cascaded drivers

The method is a direct extension of the transistor sizing approach. Instead of sizing a single driver, a chain of drivers that increase in size is inserted. The number of drivers depends on the size of the load to be charged. The last driver has to be large enough to drive the load, while the input capacitance remains low (the first driver is small). Example circuit called CASCA can be seen in figure 2.27. We can

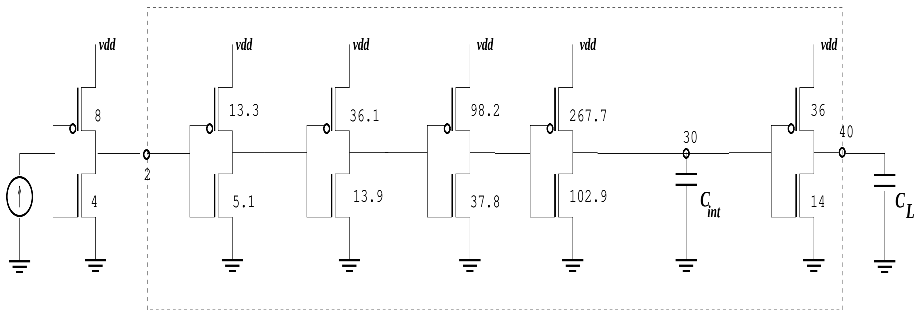


Figure 2.27. The CASCA circuit with cascaded drivers

formulate the following optimization problem: What should be the sizes and how many drivers do we need? If we assume that the output capacitance of the driver can be neglected in comparison with the input and interconnection capacitance the solution is [MC80] [Lew84]:

$$f = e \tag{2.14}$$
$$n = \ln\left(\frac{C_{int} + C_L}{C_o}\right) \tag{2.15}$$
$$\tag{2.16}$$

where e is the base of the natural logarithm, n the number of drivers, f the size increment ratio, $C_o = C_i + C_{out}$ the capacitance driven by of the minimum size driver and C_L the load capacitance. In our example the optimal number of drivers is 4, because:

$$n = \ln(\frac{10pF + 200fF}{190fF}) \approx 4$$

If the minimum size driver has the size of $W = 18.4\mu$ (p-mos transistor of $W = 13.3\mu$ and n-mos transistor of $W = 5.1\mu$), then the other 3 drivers should have sizes: 50μ, 136μ and 370.6μ.

The waveforms in the circuit are presented in figure 2.28. The summary of the simulation results is shown in table 2.3.

$T_p[ns]$	$P[mW]$	$NM[V]$	$I_{mx}[mA]$	$A[1]$
3.8	12	2.1	12.8	1

Table 2.3. Simulation results for the CASCA circuit

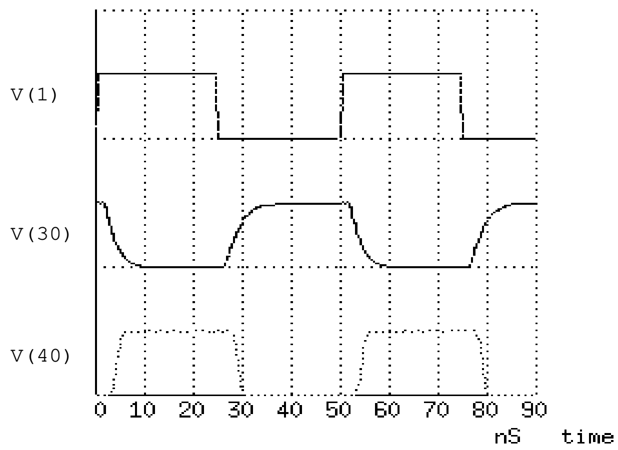


Figure 2.28. Waveforms in the cascaded drivers circuit

We can compare this results with the result of a formula from [Bak90]:

$$T_p = 0.7eR_0C_0\ln(\frac{C_{int} + C_L}{C_0}) + 0.4R_{int}C_{int} + 0.7R_{int}C_L$$

After substituting the real values we obtain:

$$T_p = 0.7e\ 3.1 \cdot 10^3\ 190 \cdot 10^{-15}\ 4 + 0 + 0 = 4.5 \cdot 10^{-9}$$

The delay of $4.5ns$ is a slightly pessimistic value compared to the delay of $3.8ns$ obtained in the simulations. This means error of about 18%.

2.4.3 Precharge to V_{dd}

This technique requires two-phase non-overlapping clocking scheme. During active period of the ϕ_1, the interconnection is precharged to V_{dd} through the *pullup* transistor (see figure 2.29). In the same time all *pulldown* transistors are forced into non-conducting state. During ϕ_2 the logic function is calculated and the interconnection is only discharged depending on its value. This is the same as in the similar, but single-phase, *domino logic* [KCH82]. The disadvantage of this circuit is that it is reported to be sensitive to glitches and subject to charge-sharing mechanisms. On the positive side, the circuit requires the clock signal only on one side of the interconnection (no problems with clock skew). The simulated circuit called PRECH is presented in figure 2.29. The results of the simulation are shown in table 2.4. It is very important that the input signal has obtained its final value

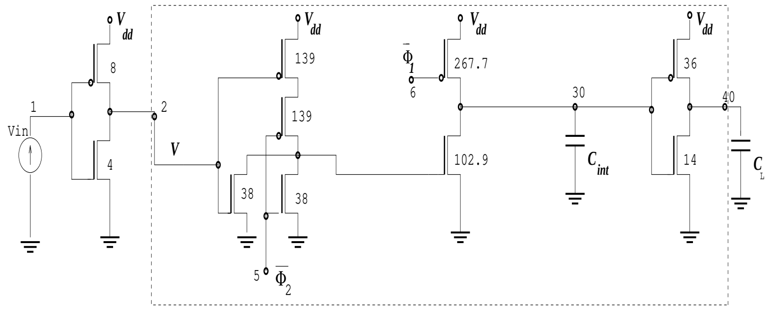


Figure 2.29. Precharge to V_{dd} circuit (PRECH)

$T_p[ns]$	$P[mW]$	$NM[V]$	$I_{mx}[mA]$	$A[1]$
2.2	10.5	2.5	28.1	1.2

Table 2.4. Simulation results for the PRECH circuit

before the evaluation phase ($\overline{\Phi_2}$ high). Otherwise the signal on the interconnection can assume incorrect value. Two different scenarios are presented in figures 2.30 and 2.31. In the second one the input signal arrives too late (in the middle of the evaluation phase). In that case discharged capacitance could not regain its charge because the pullup transistor is in the non-conducting state.

2.4.4 Threshold sense amplifier

This is a modification of the *precharge to V_{dd}* circuit. The standard inverter at the receiving side is replaced with a special receiver (circuit THRSA in figure 2.32), which is activated when the interconnection potential falls one threshold voltage (V_{TP}) below V_{dd}. As result, the circuit is faster than the normal precharge circuit, where the voltage has to reach $V_{dd}/2$. Power consumption remains the same, but

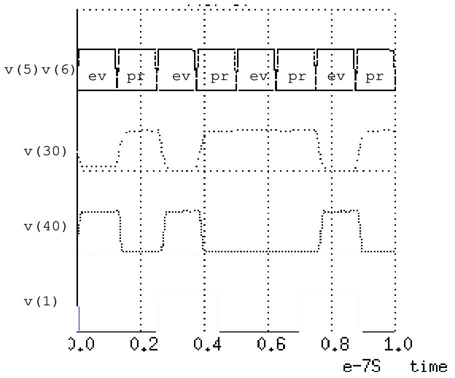


Figure 2.30. Scenario 1: input signal arrives in time

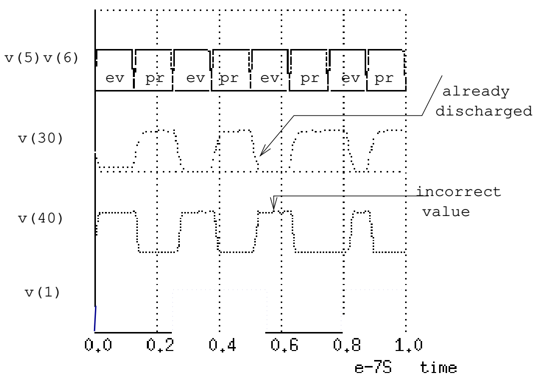


Figure 2.31. Scenario 2: input signal arrives too late

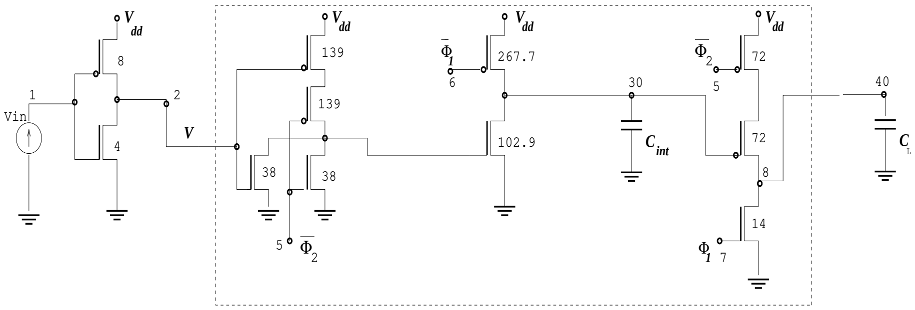


Figure 2.32. Threshold sense amplifier (THRSA)

the noise margins decreases. Because both the transmitter and the receiver are clocked, potential clock skew problems exist.

The waveforms in the circuit are presented in figure 2.33, the results of the simulations are shown in table 2.5

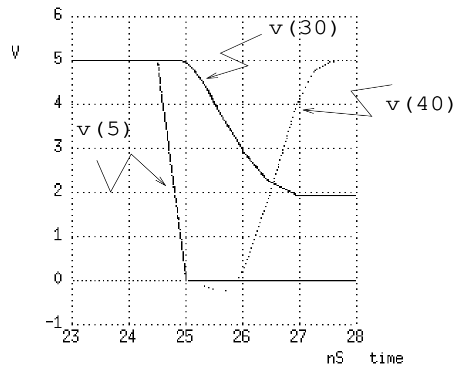


Figure 2.33. Waveforms in the THRSA circuit

$T_p[ns]$	$P[mW]$	$NM[V]$	$I_{mx}[mA]$	$A[1]$
1.75	11.5	2.0	25	1.23

Table 2.5. Simulation results for the threshold sense amplifier (THRSA) circuit

2.4.5 Precharge to high gain

In these circuits the interconnection is precharged to the high-gain region of the receiver circuit. Therefore the inverter does not require much voltage change at its input to detect state changes (see figure 2.35). This is obtained by making a feedback between the output and input of the receiving inverter (see circuit PREHG in figure 2.34). The advantage of this configuration is also that at high clock frequencies, its power consumption is low. The reason for it is that the potential on the interconnection will be confined around $V_{dd}/2$. The circuit has two limitations: it is sensitive to variations in process parameters and, it is difficult to build reliable fanout-out trees with it, because of the threshold potential matching problems for the long wires. The simulation results are shown in table 2.6.

$T_p[ns]$	$P[mW]$	$NM[V]$	$I_{mx}[mA]$	$A[1]$
1.25	11.5	2.3	25	1.7

Table 2.6. Simulation results for the precharge to high gain circuit (PREHG)

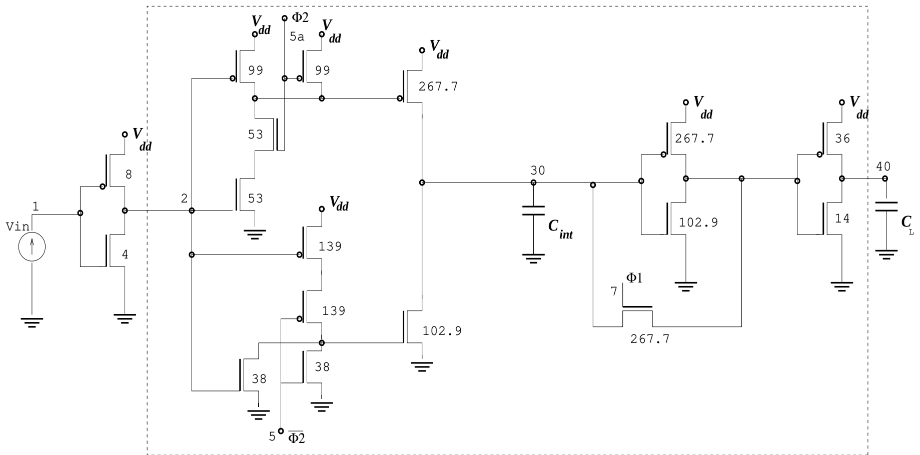


Figure 2.34. The PREHG circuit

2.4.6 Static sense amplifier

It is a *single-ended* modification of the *precharge to* $V_{dd}/2$ circuit. It does not require any clocking, therefore all transistors used for gating the signals with various clocks are not necessary (see STASA circuit in figure 2.36). When V_{in} is low the

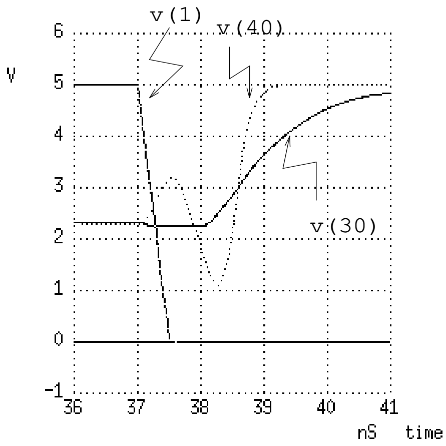


Figure 2.35. Waveforms in the precharge to high gain (PREHG) circuit

voltage is divided between transistors M1 and M2, what causes V_8 to go low. If V_{in} is high the line goes to $\approx V_{dd}/2 + \Delta$ what causes V_8 to go high (see figure 2.37). The circuit is quite immune to noise, and lends itself well for large fan-in circuits. On its bad side it is again quite sensitive to small variations in process parameters, because good matching between pullup pMOS and pulldown nMOS transistors is necessary. The simulation results can be seen in table 2.7.

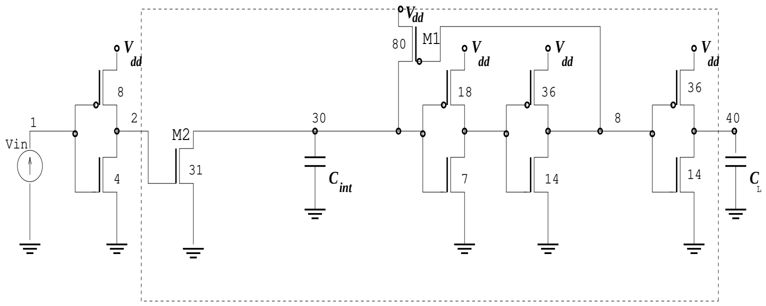


Figure 2.36. Static sense amplifier (STASA)

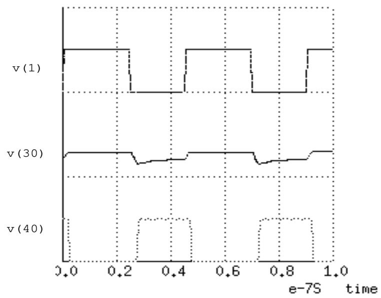


Figure 2.37. Signals in the static sense amplifier (STASA) circuit

$T_p[ns]$	$P[mW]$	$NM[V]$	$I_{mx}[mA]$	$A[1]$
2.8	19.6	2.2	9.1	0.4

Table 2.7. Simulation results for the static sense amplifier (STASA) circuit

Simulations also show that the circuit exhibits some stability problems. If for example a short pulse of 2.2V is applied to its input the circuit starts to oscillate (see figure 2.38)

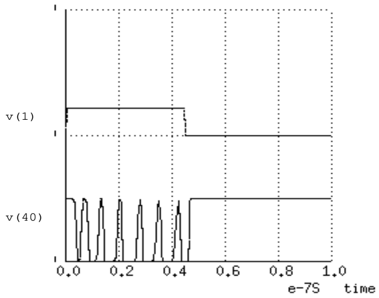


Figure 2.38. Oscillations in the static sense amplifier (STASA) circuit

2.4.7 Clocked sense amplifier

A clocked version of the *static sense amplifier* is presented in figure 2.39. When the Φ_1 is low the M1 and M2 transistors are in the non-conducting state and therefore interconnection is floating (see figure 2.40). In the evaluate phase (Φ_1 is high) the pulled up or down depending on the value of the input signal. It might even be pulled completely to ground. This will result in longer charging time. The advantage of the circuit is that the ratio between pullup and pulldown transistors is not critical. In addition, it consumes less power, because there is no continuous current flow, what is confirmed by simulations (see table 2.8). On the negative side, it exhibits some reliability problems.

$T_p[ns]$	$P[mW]$	$NM[V]$	$I_{mx}[mA]$	$A[1]$
3.2	9.4	2.1	5.8	0.7

Table 2.8. Simulation results for the clocked sense amplifier (CLOSA) circuit

2.4.8 Current sense amplifier

This is a modification of the so called "clamped bit-line current mode sense amplifier" [BJ91], which is typically used in CMOS dynamic memories. It has been

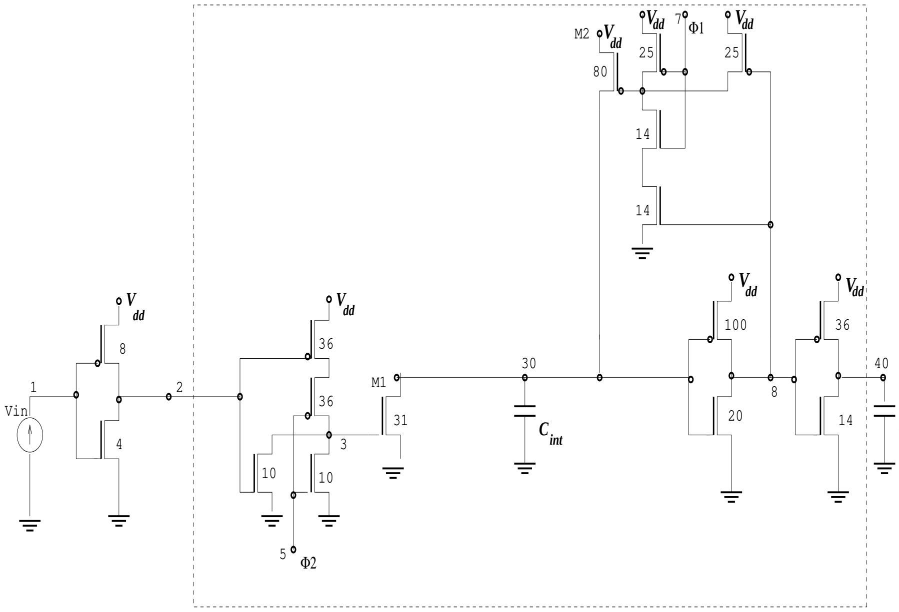


Figure 2.39. Clocked sense amplifier (CLOSA)

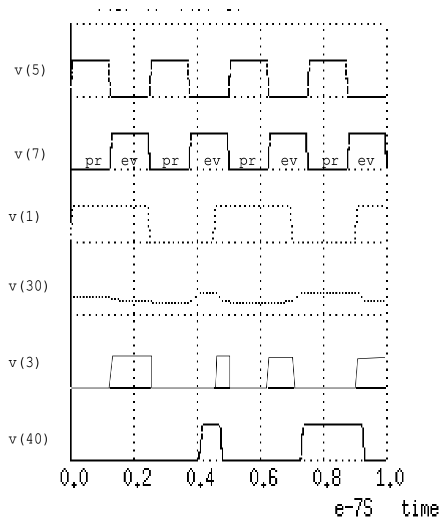


Figure 2.40. Waveform in the clocked sense amplifier (CLOSA)

adapted to work with single line, by introducing asymmetry in the voltage sensitive latch of the receiver.. The CURSA circuit (shown in figure 2.41) requires one additional signal (V_{sample}). When this signal is "1" the sense amplifier is brought to its meta-stable state (voltage at nodes V_5 and V_6 about 2.5V (see figure 2.42)). Simultaneously the n-transistor in the driver is conducting and the interconnection is slightly charged (if also the V_2 input signal is low). Consider now the latch itself. Because the load transistor M1 is larger than transistor M2, the voltage V_{L1} will be lower than V_{L2}. During the sample pulse the gate voltage of the n-transistors M3 and M4 is equal. This means that after this pulse ends, there will be a larger current flowing through M3 than through M4. Therefore the latch will always flip to one side. If however the C_{int} has obtained enough charge, the V_{L1} voltage will be higher than V_{L2} and the latch will flip to the other side ($V_5 = 0V$). The circuit has several advantages. First of all, because the sizes of all transistors and the voltage swing on the interconnection are small, its power consumption is low. It is also very fast. Another advantage is that the output signal remains stable between the “sample” pulses. The simulation results are presented in table 2.9.

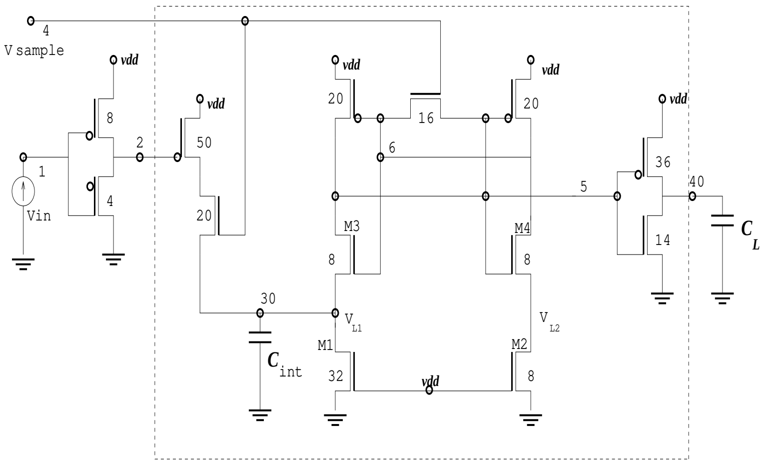


Figure 2.41. The current sense amplifier (CURSA) circuit

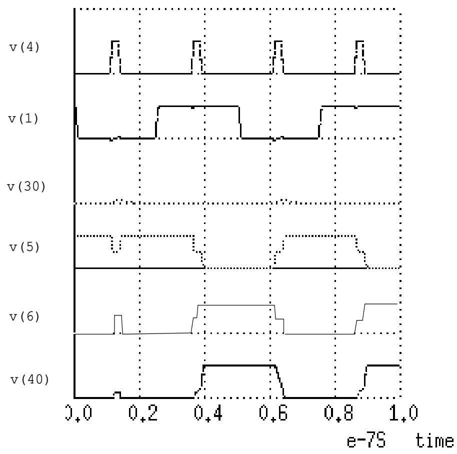


Figure 2.42. Waveforms in the current sense amplifier (CURSA) circuit

$T_p[ns]$	$P[mW]$	$NM[V]$	$I_{mx}[mA]$	$A[1]$
0.35	1.8	2.11	4.9	0.4

Table 2.9. Simulation results for the current sense amplifier (CURSA) circuit

2.4.9 Comparing the "high C" circuits

Let us summarize obtained results. For convenience we gathered all the numbers in table 2.10.

Circuit	$T_p[ns]$	$P[mW]$	$NM[V]$	$I_{mx}[mA]$	$A[1]$
TRSIZ	5.1	8.2	2.0	13.5	0.47
CASCA	3.8	12	2.1	12.8	1
PRECH	2.2	10.5	2.5	28.1	1.2
THRSA	1.75	11.5	2.0	25	1.23
PREHG	1.25	11.5	2.3	25	1.7
STASA	2.8	19.6	2.2	9.1	0.4
CLOSA	3.2	9.4	2.1	5.8	0.7
CURSA	0.35	1.8	2.1	4.9	0.4

Table 2.10. Summary of the simulation results for all "high C" circuits

All the circuits can be seen as points in the multi-dimensional (here $\mathbb{R}^5$) design space (recall figure 0.1 from the preface). Optimization problems over this space can be formulated and solved. The full design space can be projected into 2 or 3 dimensional design spaces, by selecting 2 or 3 criteria which are most important from our point of view. We show several such design spaces curves in figures 2.43 to 2.47. In all plots the point representing the current sense amplifier (CURSA) has been skipped. In the 3-dimensional plot of figure 2.43 every valid point is represented by a shaded cuboid. The part of the design space outside the cuboids represents the area of suboptimal solutions. In the 2-dimensional plots *Pareto* points have been connected with a line. Recall that a *Pareto* point is a point that corresponds to a global optimum in mono-dimensional evaluation space (there is no other point in the design space with at least the inferior objective). Only these points form the real *trade-off curve*. For example in figure 2.44 the point for CLOSA does not represent a Pareto point because it is worse, both in delay and area, than the STASA circuit. Also in 3-dimensional design-space (figure 2.43) the point for CASCA can be skipped because it is invalidated by the CLOSA circuit in both dimensions.

We would like to point out the approximate character of all presented curves. A fact that a given circuit represents a Pareto point in a *trade-off curve* does not mean that it is an absolute Pareto point. What we can only state is that a better solution is

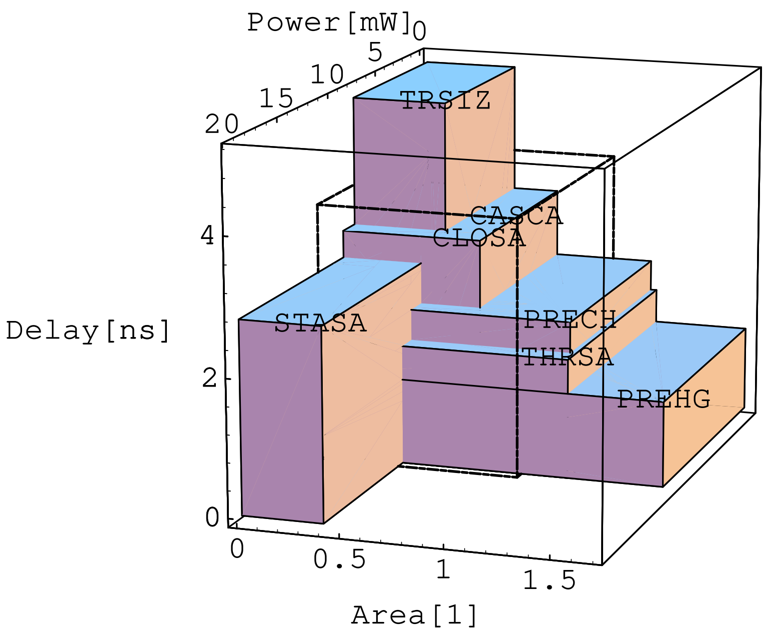


Figure 2.43. The design space for 3 criteria (delay,power,area)

not known, or has not been considered. A Pareto point is invalidated if we come out with another, better circuit. The newly developed current sense amplifier **CURSA** is a perfect example. If we forget about the noise margin then the circuit represents a perfect choice. It is fast, small, dissipates low power. Therefore it invalidates nearly every Pareto point in previously generated curves (as demonstrated in figure 2.48). This is the reason why it has been skipped in all our plots.

The *trade-off curves* of figures 2.43 to 2.47 can be used by the design automation software to guide the design. For example the area-delay *trade-off curve* from figure 2.44 can be directly used in the time budgets method, as we will see in chapter 4. The *trade-off curves* can be easily parameterized for different values of the interconnection capacitance. This way we could obtain the whole family of *trade-off curves*.

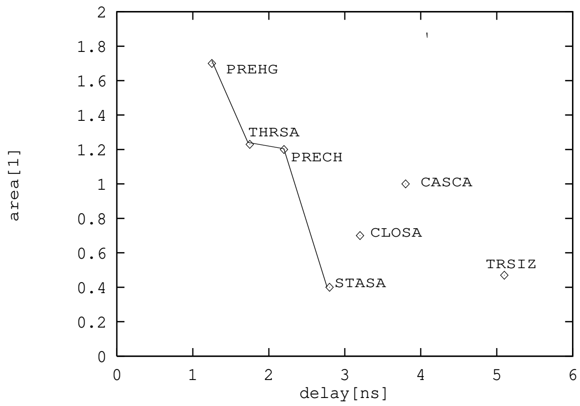


Figure 2.44. The delay-area trade-off curve

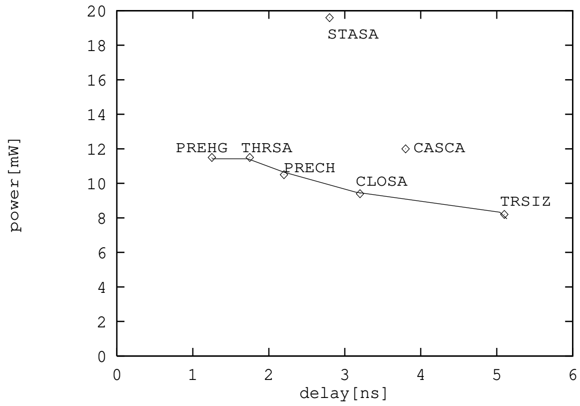


Figure 2.45. The delay-power trade-off curve

2.5 Other examples of design trade-offs

In the previous section we have presented several *transmitter-receiver pair* implementations which can be used to send digital signals through high capacitance wires. As could be seen every implementation had its weak and strong points. The *trade-off curves* were shown to be good means of gathering the information about them. The curves are advantageous also because the design automation software usually expects them as their input.

The problem of driving high capacitance nodes is of course not the only problem where several implementations having different delay-area-power characteristics are possible. In this section we will list selected other methods, which can be used to change delay-power-area properties of parts, or of whole digital system. In any

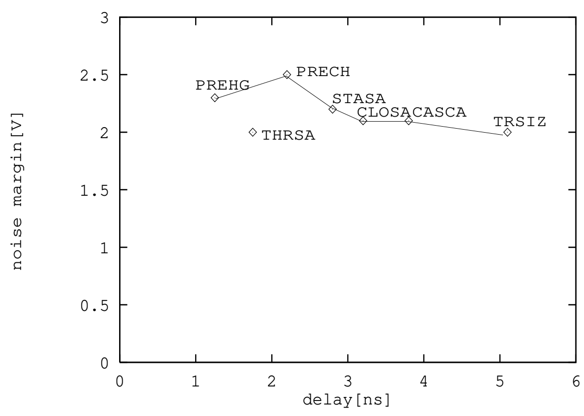


Figure 2.46. The delay-noise-margin trade-off curve

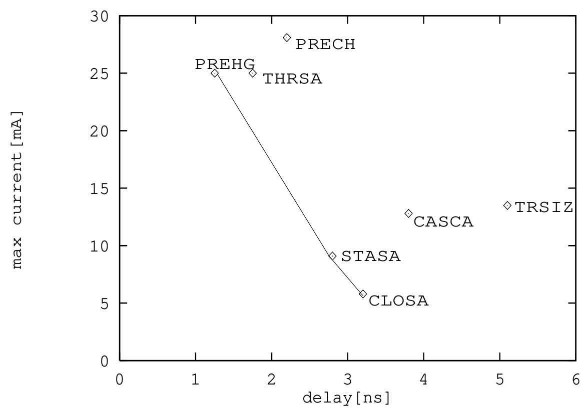


Figure 2.47. The delay-maximum-current trade-off curve

case, all available hardware or software alternatives must be considered and included in the *trade-off curves*.

2.5.1 Repeaters

For long interconnections the resistance may become comparable or even larger than the on-resistance of the driver (recall that in previous section we assumed $R_{int} \approx 0$). This problem can be solved by use of buffers which divide the wire into equal sections. We call such buffers repeaters. The use of repeaters makes delay grow linear with wire length. An example configuration with k repeaters is shown in figure 2.49. The following optimization problem can be formulated: Having an interconnection with R_{int}, C_{int} how large and how many repeaters should be inserted to obtain minimal delay. The solution is (see for details [Bak90]):

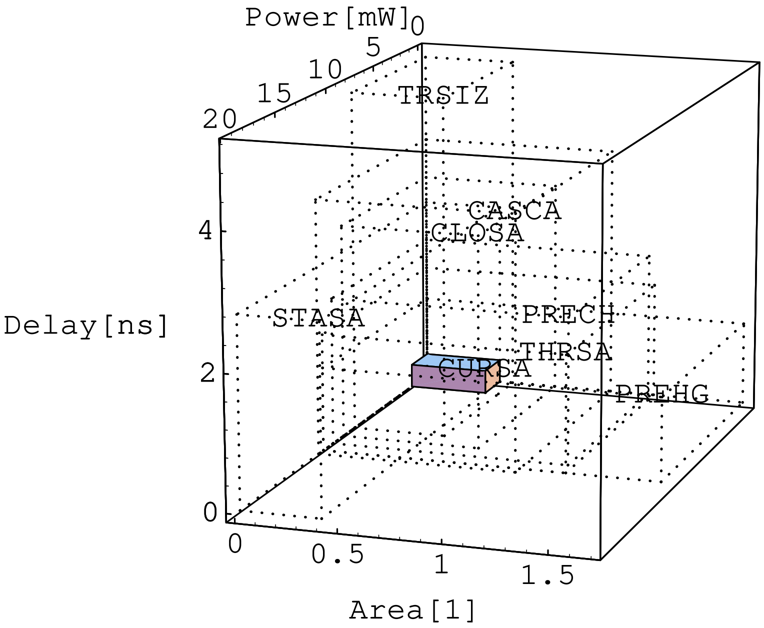


Figure 2.48. The design space for 3 criteria, including the CURSA circuit

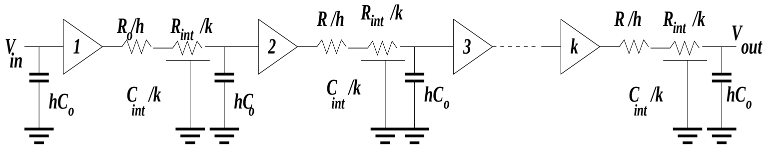


Figure 2.49. Driving long interconnections with repeaters

$$k = \sqrt{\frac{0.4R_{int}C_{int}}{0.7R_oC_o}} \tag{2.17}$$

$$h = \sqrt{\frac{R_oC_{int}}{R_{int}C_o}} \tag{2.18}$$

$$\tag{2.19}$$

where R_o and C_o are output resistance and input capacitance of the minimum size repeater, k the number of repeaters and h the size factor for the transistors in the repeaters. Then the expression for the delay becomes:

$$T_{50\%} = 2.5\sqrt{R_o C_o R_{int} C_{int}} \tag{2.20}$$

The method has of course its limitations too. The RC constant of the wire ($R_{int}C_{int}$) must be at least seven times [Bak90] the delay of the minimum size repeater (R_oC_o). Otherwise adding repeaters will increase the total delay instead of decreasing it. We can obtain implementations with different area-power-delay characteristics by varying number and size of used buffers.

2.5.2 Scaling down the IC

By scaling down, the size of all transistors and wires may be reduced by a factor of S. This should lead to gains in circuit speed, because the input capacitance of the gates decreases. Unfortunately, this is not the case with interconnection's delay. When the width of a wire is decreased the capacitance decreases but the wire resistance grows by the same amount. Consequently the RC constant cannot be reduced by adjusting mask dimensions. In addition, chip size has increased continuously throughout the evolution of the ICs. This can be modelled as a chip scaling factor S_c, which represents the increase in the length of one side of a die. This leads to the increase in the average global interconnection length. In case of wires going from one corner of the chip to the other the delay worsens by factor of $S^2 S_c^2$.

2.5.3 Wire sizing

Instead of changing the dimensions of all wires, we can modify only some of them. Also separate segments of a given interconnection may be assigned different widths and lengths. An example can be seen in figure 2.50. This technique is can be used for design of so called *zero clock skew* trees. The clock skew is defined as the maximum difference of the delays from the clock source to the clock pins on latches. Example of routing algorithm for clock lines is described in [RS91]. For a regular interconnection it may be more difficult because unlike clock or power lines it has to share the metal layers with all other interconnections in sometimes highly congested areas. This makes routing combined with wire sizing very difficult.

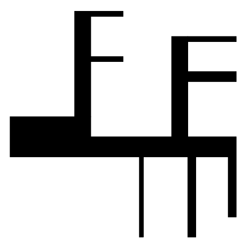


Figure 2.50. An interconnection after wire sizing

2.5.4 Transmission protocols

Concentrate now on the problem of signal transmission between, for example, computers, telecommunication or measurement equipment. Several implementation options are available. On the hardware side, beside the options from section 2.4 we can choose analog or digital transmission (which can be again serial or parallel). Also a number of software options is available. We call them transmission protocols [Phi84] [Ins78] [Ins89] [Ins85]. Very simple and cheap ones just send the signal over the transmission media without checking anything. More complex have usually a build-in intelligence, are able to check for, or even correct transmission errors, compress the data etc. The hardware and software costs of all these alternatives may differ significantly.

2.5.5 Synthesis techniques for improving circuit delays

There exist several other methods for generating circuits with different delay-area-power characteristics, which are commonly used in logic-synthesis and layout-synthesis systems [SSL$^+$92], [GS93]. They can be divided into two groups. Methods of the first group transform a given working circuit into another functionally equivalent circuit, which has some better properties. Retiming [LS91] and calculating optimal clock schedules [Szy92] are well known examples here. To the second group belong modifications of all well known logic and layout synthesis methods, which have an objective quantifier *timing-driven* in name. For example most logic minimization programs optimize the total number of literals, which is related to the area and power consumption of the circuit (for example ESPRESSO minimizer [BHMSV84]). This may be combined with timing driven cofactoring methods, for example as in [GPB92]. Also most technology mapping programs (for example **map** in SIS) accept options controlling whether it should be optimized for area or for speed (-n 1 option is **map**). There exists a huge number of *timing-driven* placement and routing programs, for example [JE89] and [SFHS91]. Some post-placement programs were also reported [KSF94].

2.6 Imprecision of circuit delays

The model of time delays in cells and interconnections presented in sections 2.2 and 2.3 is simple and therefore well suited for use in algorithms of high-level, logic and layout synthesis. Unfortunately there is a price that we have to pay for it. The price is the accuracy. Here we will try to list the most important factors which are ignored in this model:

- In our definition of time delay of a cell we ignore the fact that the delay depends on the slew rate of the input signal.

- As could be seen in subsection 2.3.2 a MOS transistor is modelled as an ideal switch, a resistor and a capacitor with a given constant values of resistance and capacitance. In practice they change depending on the working point of the transistor.

- Due to variations in the production process, the actual parameters of the transistors can vary significantly from wafer to wafer, in given technology. The production variation can cause even up to 10% variation in performance.

- No distinction is made between rise and fall delays. Just a worse of these two values is considered. In practice the rise and fall delays may differ significantly.

- It is ignored that the logic cells calculate specific logic functions and depending on the values at inputs delays may differ. Also so called *false-paths* are ignored. A path is a *false-path*, when no event can propagate along it. Example of *false-path* is shown in figure 2.51.

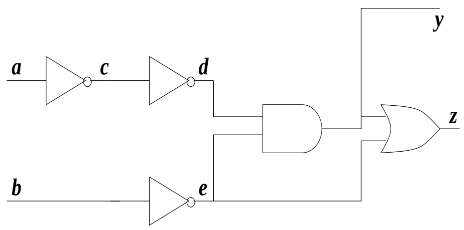


Figure 2.51. Example of false path. No event can propagate along path a, c, d, y, z and therefore it is a *false-path*.

- We model interconnections using a lumped RC model. This is far from reality since the resistance and capacitance have a distributed character.

- When modelling capacitance of the interconnections two very important factors are ignored. First of them is the *fringing field* component. It becomes more important when the size ratio of the wire approaches value of 1. Second effect which is neglected by our model is the capacitance between neighboring wires. This component is extremely difficult to predict, since it depends on the final configuration of wires on the chip. Both factors are becoming significant in nowadays ICs. Combined they might even become larger than the basic *parallel plates* component.

- At very high clock frequencies the inductance of the interconnections becomes significant. If also the wires are sufficiently long and the rise and fall time of the waveforms is comparable to the time of the flight across the line simple model should be replaced by another one - the transmission line. Fortunately this is only necessary with some very fast gallium arsenide chips having rise times of about 100ps.

All listed above factors, only by itself, can already be a reason of a large modelling imprecision. This imprecision can sometimes be avoided by using more detailed models. For example we could take the fringing fields into account when modelling the interconnection capacitance and replace the formula 2.8 by the following one:

$$C_{int} = \varepsilon_{ox} \left\{ \frac{W_{int}}{t_{ox}} - \frac{H_{int}}{2t_{ox}} + \frac{2\pi}{\ln\left[1 + \frac{2t_{ox}}{H_{int}}\left(1 + \sqrt{1 + \frac{H_{int}}{t_{ox}}}\right)\right]} \right\} \tag{2.21}$$

Unfortunately taking other factors into account may be much more difficult. For example the third component of the capacitance - the wire-to-wire capacitance depends on the final routing and therefore is unknown before the final layout is generated.

As can be seen, there are more than enough reasons to treat the problem of the imprecision of circuit delay models seriously. Ignoring it we run a large risk of generating non-optimal circuits, or even worse - ones which violate design constraints. As already stated in the preface, the solution to this problem proposed in this thesis is based on so called “possibilistic programming” [Zad78], a branch of fuzzy mathematical programming. The possibilistic programming will be introduced in chapter 3, and applied to selected problems in synthesis of digital systems in chapter 5.

Chapter 3

MATHEMATICAL PROGRAMMING METHODS OVERVIEW

Contents

3.1 Introduction

Most of the algorithms to be described in chapters 4 and 5 are based on some kind of mathematical programming, namely on integer and possibilistic programming. Therefore in this chapter, we will give an overview of integer programming (**IP**), its basic properties and the more popular methods used for solving **IP** problems. After that, the reader will be introduced to the concepts of possibilistic programming. First, possibility distributions will be defined, followed by some real life examples. After a short comparison of stochastic and possibilistic programming, selected methods of possibilistic programming will be outlined.

3.2 Integer programming problems

The following notation will be used

1. Vectors will be denoted by small letters without any additional under- or overlining, for example x, b. If there is no ambiguity no special symbols are used for transposed vectors. By x_j we denote $j-th$ component of vector x. A vector symbol with an asterisk (for ex. x^*) denotes the optimal vector for a given optimization problem, while the same symbol with an overline ($\bar{x}$) is used to denote an optimal vector for the linear relaxation of that problem.

2. Matrices will be denoted by capital letters without any additional under- or overlining. a_{ij} is the element from the $i-th$ row and the $j-th$ column of matrix A.

3. For integer programming problems we use bold capital letters, for example $\mathbf{P}$. Their linear relaxations are denoted by the same letter but overlined. The dual problem of **P** is denoted by a bold capital $\mathbf{D}$.

   If $\mathbf{D}$ appears together with a subscript this subscript will be the name of the original problem of this dual problem.

4. $v(\mathbf{P})$ denotes the optimal value of cost function (score for x^*).

5. $F(\mathbf{P})$ denotes the set of feasible solutions to our problem.

The integer programming problem $\mathbf{P}$ can be formulated as follows. Find the minimum of the function:

$$f(x) = \sum_{j=1}^{n} c_j x_j \tag{3.1}$$

subject to the set of constraints:

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i,\ i = 1, \ldots, m, \tag{3.2}$$

$$x_j \geq 0,\ j = 1, \ldots, n,$$

$$x_j \in \mathbb{Z}, j \in N = \{1, \ldots, n\}.$$

Using the introduced notation : Find

$$f(x^*) = v(\mathbf{P}) = \min\{f(x) \mid Ax \leq b,\ x \geq 0,\ x \in \mathbb{Z}^n\} \tag{3.3}$$

where : *x* is the vector of the unknowns,
c is the vector of cost coefficients,
A is the matrix of constraints coefficients,
b is the constraint vector (right hand side).

If we remove the constraint that $x \in \mathbb{Z}^n$ we will obtain a linear relaxation of the problem ($\bar{\mathbf{P}}$). Such a problem is called also the *linear program*.

Integer programming (**IP**) is unfortunately in general $\mathcal{NP} - hard$ [GJ79]. However if a program has a particular structure, often polynomial algorithms are known to exist. Also for unstructured problems, a careful program construction [CJP83] [GE90] in combination with a good **IP** solver [CPL93] can result in very low running times even for a quite large programs. Simultaneously the **IP** methods have several salient features. First of all provided enough time they guarantee reaching the globally optimal solution. It is also important that they offer quite large degree of flexibility. Many different constraints can easily be formulated, the same is true for the cost functions. Also the possibilistic theory (see chapter 3) can be directly applied, without an increase in the number of variables.

The *linear programs*, from the other side, are relatively easy to solve, for example by use of the simplex [KM72] (in practice polynomial) or ellipsoid [Cha79] (provably polynomial) algorithms.

3.3 Related integer programming problems

Having a matrix A $(m \times n)$ and vectors $b \in R^m$, $c \in R^n$, we can formulate two different linear programs:

$$v(\mathbf{P}) = \max\{cx \mid Ax \leq b,\ x \geq 0\} \tag{3.4}$$

and

$$v(\mathbf{D}) = \min\{ub \mid uA \geq c,\ u \geq 0\}. \tag{3.5}$$

The problems are said to be each other "duals". When problem **P** is called the *primal problem*, then **D** will be called the *dual problem*. In this notation u is an m-dimensional row vector of dual variables. Nice thing about these problems is that a dual problem to **D** is equivalent to the problem **P** in the sense that we can solve the dual problem **D** (which might be easier) and convert the obtained solution $\bar{u}$ to $\bar{x}$.

Definition 1 *Problem:*

$$v(\mathbf{P_1}) = \min_{x \in F(P_1)} f(x) \tag{3.6}$$

is the relaxation of problem

$$v(\mathbf{P_2}) = \min_{x \in F(P_2)} g(x) \tag{3.7}$$

if:

1. $g(x) \leq f(x)$ *for each* $x \in F(\mathbf{P_1})$,

2. $F(\mathbf{P_2}) \supseteq F(\mathbf{P_1})$.

The main reason for introducing relaxed problems is that they usually are easier to solve than the original integer program and provide estimates of the optimal solution. The most popular relaxation is obtained by ignoring the constraint that all variables are integers. We denote the thus obtained *linear relaxation* by $\overline{\mathbf{P}}$. If the solution of the relaxed problem is an integer vector then it is also the solution to the original problem $\mathbf{P}$. If it is not, $f(\bar{x})$ can still be useful as an estimate of the value of cost function in the optimal point of $v(\mathbf{P})$:

$$v_{est}(\mathbf{P}) = \lceil f(\bar{x}) \rceil \leq v(\mathbf{P})$$

Let us consider the following integer programming problem:

$$v(\mathbf{P}) = \max\{cx | Ax \leq b, 0 \leq x \leq d, x \in Z^n\}, \tag{3.8}$$

The *Lagrangian relaxation* of this problem is defined:

$$v(\mathbf{Lu}) = \max\{cx + u(b - Ax) | 0 \leq x \leq d, x \in Z^n\}, \tag{3.9}$$

where $u \geq 0, u \in \mathbb{R}^m$ is a constant vector. In other words we add all our constraints with appropriate coefficients to the cost function of the problem. Problem $\mathbf{Lu}$ is of course a relaxation of $\mathbf{P}$, because $F(\mathbf{Lu}) \supseteq F(\mathbf{P})$, and for each $x \in F(\mathbf{P})$,$b - Ax \geq 0$. Consequently $cx + u(b - Ax) \geq cx$. The feasible solution space for $F(\mathbf{Lu})$ is very simple, therefore the relaxation is easy to solve.

3.4 The knapsack problem

Consider the following problem. A scout has to load n boxes with weights $a_1, a_2, \ldots, a_n$ and values $c_1, c_2, \ldots, c_n$ into his knapsack with capacity b in such a way that the total value of loaded boxes is maximized. This problem is known as a *knapsack problem* and can be formulated as follows. Let x_j be a binary variable, taking value 1 if j-th box is loaded and 0 if not. Then we can write:

$$v(Z) = \max \sum_{j=1}^{n} c_j x_j \tag{3.10}$$

$$\sum_{j=1}^{n} a_j x_j \leq b, \tag{3.11}$$

$$x \in \mathbb{B}, \quad j = 1, \ldots n. \tag{3.12}$$

As can be seen the the program has only one constraint. Its importance comes from the fact that several other problems can be transformed to it. It is also worth mentioning that its linear (non-integer) version can be efficiently solved.

3.5 Basic properties of the IP programs

In **IP** methods we often convert (transform) a problem into another equivalent problem, which is easier to analyze or solve. The example below will demonstrate this issue, together with other notions introduced in the previous section.

Example 5

$$v(\mathbf{P}) = \max(2x_1 + x_2) \quad (3.13)$$
$$s.t \quad 3x_1 + x_2 \le 7, \quad (3.14)$$
$$2x_1 + 3x_2 \le 10, \quad (3.15)$$
$$x \ge 0, \quad x \in Z^2. \quad (3.16)$$

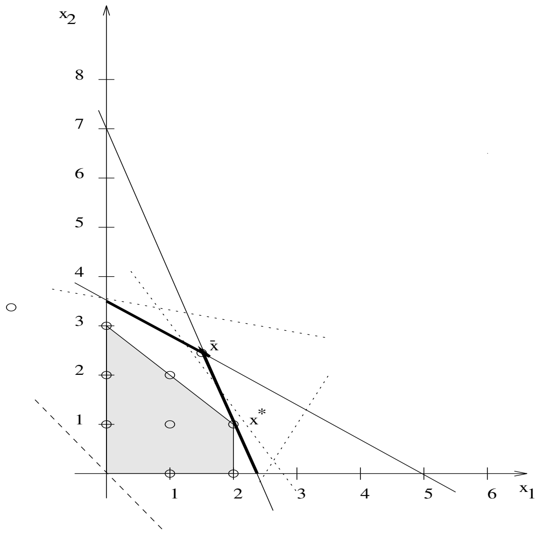


Figure 3.1. Example optimization problem

The graphical illustration of the problem is shown in figure 3.1. The feasible solutions are drawn with small ovals, dashed lines represent the sections of the cost function. First observation is that there exist an infinite number of equivalent formulations of our problem. For example the 3 dotted lines represent constraints with the same set $F(\mathbf{P})$. *The set* $F(\mathbf{P})$ *can be described by its* convex hull*, i.e. the smallest convex set that includes* $F(\mathbf{P})$. *We will denote it as* $\mathbf{conv}F(\mathbf{P})$. *The solution to the problem* $\mathbf{P}$ *denoted with* x^* *is not the same as the solution to the linear relaxation of the problem (denoted with* $\bar{x}$*). Rounding* $\overline{x}$ *up and down will not provide the same point as* x^*.

In figure 3.1 the set $\mathbf{conv}F(\mathbf{P})$ *has been hashed. Recall that the set* S *is convex, if from the fact that* $x_1, x_2 \in S, x_1 \neq x_2$ *we can conclude that a segment between* x_1 *and* x_2 *belongs to* S*, i.e.* $x = (\alpha x_1 + (1-\alpha)x_2) \in S$ *for arbitrary* $0 \le \alpha \le 1$.

Looking at figure 3.1 we can easily write down other constraints describing the $\mathbf{conv}F(\mathbf{P})$ *and obtain the following program:*

$$\begin{aligned} v(\mathbf{P}') &= \max(2x_1 + x_2) && (3.17)\\ s.t \quad & x_1 + x_2 \leq 3, && (3.18)\\ & x_1 \leq 2, && (3.19)\\ & x \geq 0, \quad x \in Z^2. && (3.20) \end{aligned}$$

The problem $\mathbf{P}$ *is equivalent to the problem* $\mathbf{P}'$*. In other words solving* $\mathbf{P}'$ *we will obtain the same solution, because* $F(\mathbf{P}) = F(\mathbf{P}')$ *and therefore* $F^*(\mathbf{P}) = F^*(\mathbf{P}')$*. What is even more interesting, in* $\mathbf{P}'$ *we can skip the constraint* $x \in Z^2$*, getting the linear programming problem (non-integer)* $\mathbf{P}''$*. Also* $F^*(\mathbf{P}) = F^*(\mathbf{P}'')$*.*

In this simple example we have shown that by changing the set of constraints we can obtain an equivalent but tighter formulation of **P**, which is much easier to solve. This kind of transformation is a basis of most **IP** methods. Unfortunately it is usually not trivial to find the right set of constraints. We will come back to this problem later in this chapter.

Here follow other simple transformations, which are valid for arbitrary mathematical programming problems.

1. The maximization of the cost function $f(x)$ s.t. $x \in F(\mathbf{P})$ is equivalent to minimization of a function $g(x) = -f(x)$ s.t. $x \in F(\mathbf{P})$. We get:

$$v(\mathbf{P}) = \max_{x \in F(P)} f(x) = - \min_{x \in F(P)} (-f(x)). \tag{3.21}$$

2. Given a constant $a \in \mathbb{R}$, the maximization of the cost function $f(x) = g(x) + a$ s.t. $x \in F(\mathbf{P})$ is equivalent to maximization of a function $g(x)$ s.t. $x \in F(\mathbf{P})$. We get:

$$v(\mathbf{P}) = \max_{x \in F(P)} f(x) = \max_{x \in F(P)} (g(x) + a) = a + \max_{x \in F(P)} g(x)). \tag{3.22}$$

3. Arbitrary constraint $g(x) \geq 0$ after multiplying it by -1 is equivalent to $h(x) \leq 0$ where $h(x) = -g(x)$.

4. Arbitrary constraint $g(x) \leq 0$, after adding a non-negative extra variable s, can be replaced by the equality:

$$g(x) + s = 0, \quad s \geq 0. \tag{3.23}$$

5. Every equality $g(x)$ can be replaced by two inequalities $g(x) \leq 0$ and $g(x) \geq 0$.

3.6 Overview of the IP methods

3.6.1 Structured IP problems

Structured **IP** problems have the constraints matrix A with such particular structure that a polynomial time algorithms solving it are known to exists. As examples we can give here:

- System of difference constraints: each row of the linear programming matrix A contains one 1 and one -1, and all other entries of A are 0. In other words, we have a set of constraints involving n unknowns, where every constraint has the form:

$$x_j - x_i \leq b_k, \quad where \;\; 1 \leq i, \;\; j \leq n, \;\; 1 \leq k \leq m. \tag{3.24}$$

  The same constraint set arises in shortest paths problems. Therefore such system of inequalities can be satisfied, or determined to be inconsistent in linear time by use of the **Bellman-Ford** algorithm [CLR90](p.532-536). We have to construct a special graph and run the algorithm on it. This construction can be also found in [CLR90](p.539-543).

- Problems with all elements of A equal 1 and 0, and the right hand side vector of all 1's. We can map them into, for example, the node packing problem on a graph. If the graph is claw-free then it can be solved in polynomial time using Minty's algorithm [Min80].

- The minimum-cost flow problem in graph $G(V, E)$:

$$v(\mathbf{P}) = \min \sum_{(i,j)\in E} c_{ij} x_{ij} \tag{3.25}$$

$$s.t. \sum_{j\in V(i)} x_{ij} - \sum_{j\in V'(i)} x_{ji} = \left\{ \begin{array}{ll} \leq a_i, & i \in V_1, \\ = 0, & i \in V_2, \\ \leq -b_i, & i \in V_3, \end{array} \right\} \tag{3.26}$$

$$0 \leq x_{ij} \leq d_{ij}, \qquad (i,j) \in E, \tag{3.27}$$

  where V_1 - set of sources, V_3 - set of sinks, V_2 - set of normal vertices, c_{ij} - unit cost of the flow from vertex i to j, d_{ij} - capacity of the edge (i,j), a_i - amount of flow leaving source i, b_i - amount of required flow at a sink i, $V(i)(V'(i))$ - set of predecessors/ancestors of vertex i. The problem can be solved with for example Goldberg and Tarjan algorithm [GT87] in $\mathcal{O}(|V||E| \lg |V| \lg(|V|^2/|E|))$ time.

3.6.2 Non-structured IP problems.

General (non-structured) **IP** programs may be very difficult to solve. If the problem is large or the difference between the bounds (estimates of $f(x^*)$) and the optimal solutions great, the general **IP** solvers may not produce any results in reasonable time. The working principle of nearly all methods is solving a series of temporary linear programming problems. Here follows an overview of the most popular methods:

- Solving the linear relaxation of the **P**: This method may seem attractive. If $\bar{x}$ is an all-integer solution, then it represents also an optimal solution to **P**. Unfortunately for unstructured programs it seldom happens. Rounding up and down the non-integer variables does not guarantee obtaining the optimal solution to **P** (as could be seen in example 5).

- The cutting planes method [Gom58]: We solve first the linear relaxation of **P**. If the solution is not all-integer, we construct an extra constraint (a cutting plane), which separates that solution from the set $F(\mathbf{P})$. We repeat these steps until an all-integer solution is found. It can be proven that we will succeed after finite number of cuts. Unfortunately this number can be in practice quite large and therefore the method is very seldom used.

- The branch and bound method: This is the most popular method. It creates new *linear (sub)programs* by bounding each variable towards integral values. Formed in this way search tree is expanded only if it may lead to a better solution than the current best one. From experience it has been found that usually problems of up to 200 variables can be solved using this method. Unfortunately often if the shape of the search polyhedron is similar to a narrow needle even very small problems will not converge.

- Polyhedral approaches: It has been observed that finding the facets of a given **IP** can be very advantageous. Usually the branch and bound technique would then generate fewer search nodes. In 1983 Crowder et.al. [CJP83] demonstrated it in their system, which was able to solve sparse binary **IP**'s of about 2000 variables in less than 1 CPU hour. Unfortunately there exists no formal method for obtaining facets of a given **IP**. Usually however a problem or its part can be mapped into a well known problem, as for example the node packing, whose facets are partially characterized. Also knapsack inequalities can be extremely successful in speeding up the solution. The result of all these transformations is that the solution to the linear relaxation to the **IP** contains substantially less non-integer variables. Thus, less branch and bound iterations are necessary. As one very successful application of these methods we can give the architectural synthesis system of C. Gebotys et.al. [GE90].

- Heuristics: There exists a number of heuristic approaches to solving **IP** problems. At the sacrifice of the global optimality, some of them are in limited time able to find a solution which is very close to the optimal one. One of the most successful heuristics of this kind, which can be applied to many combinatorial problems, is the so called *simulated annealing* algorithm [OvG89].

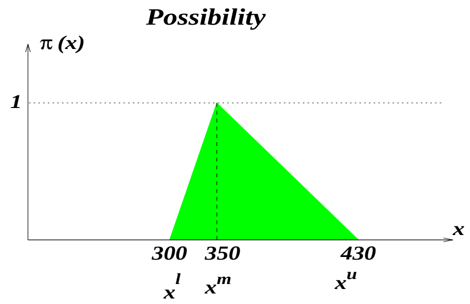


Figure 3.2. A triangular possibility distribution of fuzzy number x.

3.7 Possibility distributions

Now we will concentrate on another branch of mathematical programming - on so called possibilistic programming. We assume that the reader has already a basic understanding of fuzzy sets and possibility theory. For all who feel novice to these subjects we provide a small introduction in the appendix at the end of this book. We will start with a notion of possibility distribution.

To define fuzzy/imprecise numbers, we use possibility distributions[1], as introduced by Zadeh [Zad78], see e.g. figure 3.2. In this thesis, the possibility measure of an event might be interpreted as the possibility degree of its occurrence under the possibility distribution, $\pi(.)$, (analogous to a probability distribution).

Among the various types of distributions, triangular and trapezoid are the most common in solving possibilistic mathematical programming problems. We will concentrate only on triangular fuzzy numbers.

The triangular fuzzy number is denoted by $X = (x^m, x^l, x^u)$, where x^m is the *most possible* value (possibility function $\pi(x) = 1$), x^l and x^u are lower and upper bound values of the acceptable events, respectively. These bound values can be interpreted as the most pessimistic and the most optimistic values. Which one is pessimistic and which optimistic depends on the context.

Example 6 *Consider such a fuzzy number to be the size of the phase rotator unit from example 1 in the preface. Recall that it was difficult to estimate. Therefore, instead of a poor approximation, we represent the size as a triangular fuzzy number. For example:* $size(A) = X = (350, 300, 430)$ *means that the most possible size of* A-type *unit is 350, we do not expect it to be smaller than 300, nor larger than 430 transistors. In this context* $x^l = 300$ *is our most optimistic value,* $x^u = 430$ *most pessimistic.*

Exact values of x^m, x^l and x^u can be derived from technology parameters, placement and routing methods used, and other information about the structure of the circuit.

[1]in fuzzy mathematical programming we use membership functions instead.

Example 7 *In the* **Ocean** sea-of-gates *system [GS94] (see chapter 1) we have to decide what might be the size of our placement region before performing placement. This is not easy because of two reasons:*

1. *We want to generate layouts as small as possible.*
2. *We do not want to generate too dense layouts since later we may not be able to route them*[2].

Consequently, the existing software tries to guess this size based on factors such as the shapes of cells or the amount of wiring. Retries however are still necessary.

How can we construct the fuzzy number representing the area of such functional unit? The lower bound can be easily calculated as the sum of areas of all cells to be placed. In practice such a dense placements can hardly ever be routed successfully. Every circuit can be placed and routed if we define our placement region size to be large enough (the software can easily calculate it). This can be used as x^u. *Also the most* possible *size (*x^m*) can be estimated on the basis of experience.*

3.8 Stochastic versus possibilistic programming

Stochastic programming [Sen72] [SM84] can be applied to decision problems where input data have some given probability distributions. It is however not a very attractive approach for the problems of automatic circuits synthesis. The main reasons are

1. its low computational efficiency,
2. the inflexibility of the probabilistic doctrines which makes modeling difficult, and
3. the uncertainties in performance are not really captured by randomness.

Possibility distributions do not have these disadvantages. Therefore, since the introduction by Zadeh, there has been much research on the possibilistic theory. Possibilistic decision making models have provided an important tool in handling practical decision making problems.

To avoid any misunderstanding we would like to emphasize that possibility distributions should not be viewed as "cheap replacements" of probability distributions. The possibilistic approach does not assume nearly anything about probabilistic properties of the system. It is also not true, that using it we loose some information. Let us elaborate on that. Usually the probability distributions do not

[2] In *sea-of-gates* the routing is performed over the cells, there are no special channels for routing.

exist, at least without additional (large) effort to calculate them. We assume that the only information available is a possibility distribution (3 numbers) and nothing more. Therefore we do not loose any information. We just start from different kind of information and therefore also use different (more suitable) its model and methods.

Example 8 *The timing specification for our phase rotator (figure 0.4) contains the following timing constraint:*

> *The delay between arrival of the signal on the "control" input and the time when the data becomes available at the "MIr" output should be about 10ns. In any case the signal should not appear at the output sooner than 5ns or later than 15ns.*

The most straightforward way of modelling this constraint is to express it as the following fuzzy constraint:

$$delay_{control,MIr} \leq X = (10, 5, 15)$$

On the other hand if we were using stochastic programming, X would have to be a random variable with a given probability distribution. It is clear that such a model would not represent the real meaning of our constraint.

Of course, it does not mean that possibility theory can be a substitute in all instances for probability theory.

3.9 Possibilistic linear programs

We define the following Possibilistic Linear Program (**PLP**):

$$\min \quad \tilde{c}x \tag{3.28}$$

$$s.t. \quad \tilde{A}x \leq \tilde{b}, \quad and \quad x \geq 0 \tag{3.29}$$

where $\tilde{c}$, $\tilde{A}$, and $\tilde{b}$ may consist of imprecise numbers with possibilistic distributions.

First let us concentrate on the case when only c is imprecise. For given x, the value of the fuzzy objective function (eq. 3.28) is a fuzzy number defined by three corner points ($c^m x$,1),($c^l x$,0) and ($c^u x$,0). Thus, minimizing the fuzzy objective can be obtained by pushing these three critical points to the left. To keep the triangular shape (normal and convex) of the possibility distribution function, instead of minimizing these three objectives simultaneously, we are going to minimize $c^m x$, maximize $[c^m x - c^l x]$ and minimize $[c^u x - c^m x]$, see figure 3.3.

In this way we obtain the following auxiliary program:

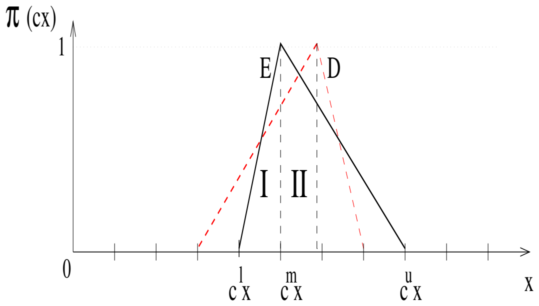


Figure 3.3. The strategy to solve " $\min \tilde{c}x$". We prefer the possibility distribution of D to that of E.

$$\begin{aligned} \max \quad & z_1 = (c^m - c^l)x \\ \min \quad & z_2 = c^m x \\ \min \quad & z_3 = (c^u - c^m)x \end{aligned}$$

$$s.t. \; x \in X = \{x : Ax \leq b \,,\; x \geq 0\} \tag{3.30}$$

This Multiple Objective Linear Program (**MOLP**) is equivalent to minimizing the *most possible* value of the imprecise cost (at the point of possibility degree = 1). At the same time, we minimize the "risk of paying higher cost" (see region II in figure 3.3), and maximize "the possibility of the lower cost" (region I).

To solve problem 3.30 we may use any **MOLP** technique such as utility theory, goal programming, fuzzy programming or interactive programming. One possibility is to use Zimmermann's fuzzy programming method [Zad65] with the normalization process proposed by Lai and Hwang [LH92].

METHOD 1

First we calculate the following Positive Ideal Solutions (PIS) and Negative Ideal Solution (NIS) of the objective functions:

$$\begin{aligned} z_1^{PIS} &= \max_{x\in X} (c^m - c^l)x \\ z_2^{PIS} &= \min_{x\in X} c^m x \\ z_3^{PIS} &= \min_{x\in X} (c^u - c^m)x \\ \\ z_1^{NIS} &= \min_{x\in X} (c^m - c^l)x \\ z_2^{NIS} &= \max_{x\in X} c^m x \\ z_3^{NIS} &= \max_{x\in X} (c^u - c^m)x \end{aligned} \tag{3.31}$$

where the feasible solution space $x \in X$ is the same as in eq. 3.30. The linear membership functions of these objective functions can now be computed as (see also figure 3.4):

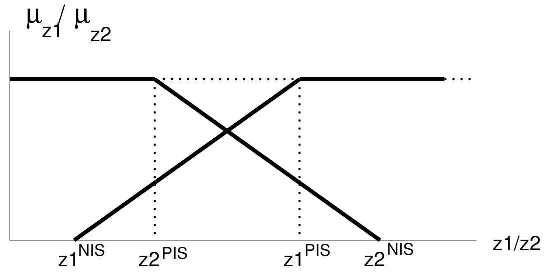


Figure 3.4. The membership functions of the objectives, z_1 and z_2.

$$\mu_{z1} = \begin{cases} 1 & if \quad z_1 > z_1^{PIS} \\ \frac{(z_1 - z_1^{NIS})}{(z_1^{PIS} - z_1^{NIS})} & if \quad z_1^{NIS} \leq z_1 \leq z_1^{PIS} \\ 0 & if \quad z_1 < z_1^{NIS} \end{cases} \tag{3.32}$$

$$\mu_{z2} = \begin{cases} 1 & if \quad z_2 < z_2^{PIS} \\ \frac{(z_2^{NIS} - z_2)}{(z_2^{NIS} - z_2^{PIS})} & if \quad z_2^{PIS} \leq z_2 \leq z_2^{NIS} \\ 0 & if \quad z_2 > z_2^{NIS} \end{cases} \tag{3.33}$$

and μ_{z3} is similar to μ_{z2}. Finally we solve the following Zimmermann's equivalent single-objective linear programming program:

$$\max \alpha$$

$$s.t. \;\; \mu_{zi}(x) \geq \alpha, \;\; i = 1, 2, 3 \; and \; x \in X. \tag{3.34}$$

The optimal solution of equation 3.34 will provide a satisfying solution under the previous strategy of maximizing the chance of higher profit, and minimizing the most possible value and the possibility of lower profit.

As can be seen the complexity of the problem does not increase - we simply have to solve 7 temporary problems. This should not be a problem if the running time is low. If however the problem is larger it makes quite a difference if we can obtain results within one or seven hours. Therefore we proposed [KO95] also a simpler version of this method, in which we do not calculate the normalization coefficients.

METHOD 2

We can replace the three goal functions of the program 3.30 by a single one:

$$\min \left[-\beta_1 (c^m - c^l) x + \beta_2 c^m x + \beta_3 (c^u - c^m) x \right] \tag{3.35}$$

where coefficients β_i represent the relative importance of the three subgoals. Without lost of generality we can assume that $\beta_1 = 1$. Then, we can rewrite the equation above as:

$$\min \left[c^l x + (\beta_2 - \beta_3 - 1) c^m x + \beta_3 c^u x \right] = \min \left[a_1 c^l x + a_2 c^m x + a_3 c^u x \right] \tag{3.36}$$

Consider the relative importance of the three subgoals of program 3.30. It is easy to see that in practice every designer will assign β_i values in such a way that $\beta_1 \leq \beta_3 \leq \beta_2$. Otherwise we could obtain solutions with, for example, small risk of obtaining a worse solution but relatively large chance of obtaining a bad solution. Because of that and since $\beta_1 = 1$ the coefficients a_1, a_3 will be positive. Con-

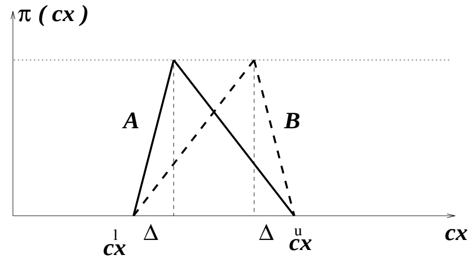


Figure 3.5. Two fuzzy numbers

sider now two "symmetric" fuzzy numbers as in figure 3.5. They have the same pessimistic and optimistic values, but **A**'s most possible value is smaller than **B**'s most possible value. If we are to prefer **A** over **B** in such case we should have

$$c^l x+(\beta_2-\beta_3-1)(c^l x+\Delta)+\beta_3 c^u x \leq c^l x+(\beta_2-\beta_3-1)(c^u x-\Delta)+\beta_3 c^u x. \quad (3.37)$$

This implies that

$$(\beta_2-\beta_3-1) \geq 0, \quad (3.38)$$

which is also equivalent to the requirement that $a_2 \geq 0$. We will use this property in the section about retiming.

If A or b is also imprecise the same methodology with only small extensions can be used. We can replace all constraints with imprecise coefficients with three aux-

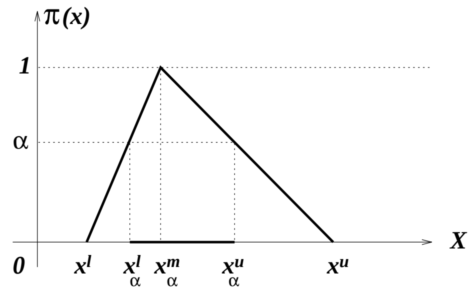


Figure 3.6. The minimal acceptance possibility

iliary inequality constraints (as proposed by Ramik and Rimanek [RR85]):

$$A^m_\alpha x \leq b^m_\alpha, \quad (3.39)$$
$$A^l_\alpha x \leq b^l_\alpha, \quad (3.40)$$
$$A^u_\alpha x \leq b^u_\alpha \quad (3.41)$$

where α is the minimal acceptable possibility. Acceptable events are all the events with the possibility $\geq \alpha$, see also figure 3.6. If the value of α is given then we obtain a crisp **MOLP** problem.

Another, even simpler, approach is to use a so called *most likely value*, which is $(4a^m_\alpha + a^l_\alpha + a^u_\alpha)/6$ where α is a minimal acceptance possibility. If we apply it to a fuzzy matrix $\tilde{A}$ we obtain the following set of constraints:

$$(4A^m_\alpha + A^l_\alpha + A^u_\alpha)/6) \leq b. \tag{3.42}$$

The concept of the *most likely value* can be even applied if constraints depend in a non-linear way on some fuzzy coefficient. We will take advantage of it in chapter 5.2

Chapter 4

TIME BUDGETS

Contents

4.1 Introduction

In this chapter we present our first approach to the problem of performance driven synthesis - the so called *time-budgets* method. The approach belongs to the third category described on page 6 in the preface. The common feature of methods in that category is that redesigns are less likely because of the use of the previously gathered knowledge. This knowledge can be extracted from other designs. The database can also be built by a priori assessing the consequences of possible design decisions, for instance by building beforehand several versions of every logic gate (with different shapes or driving capabilities). The characteristics of these versions could be collected in form of the *trade-off curves* (in the same fashion as for the "high-C" circuits in section 2.4). This approach seems to lend itself for dynamic programming [Bel57]. In case of only area/shape of the layout, very elegant algorithms are known to exist [Ott83]. If however delays and timing constraints are involved, application of dynamic programming does not lead to efficient programs. Efficient solution is however possible if the problem is formulated as a specially structured (see section 3.6.1) integer program, as will be shown below.

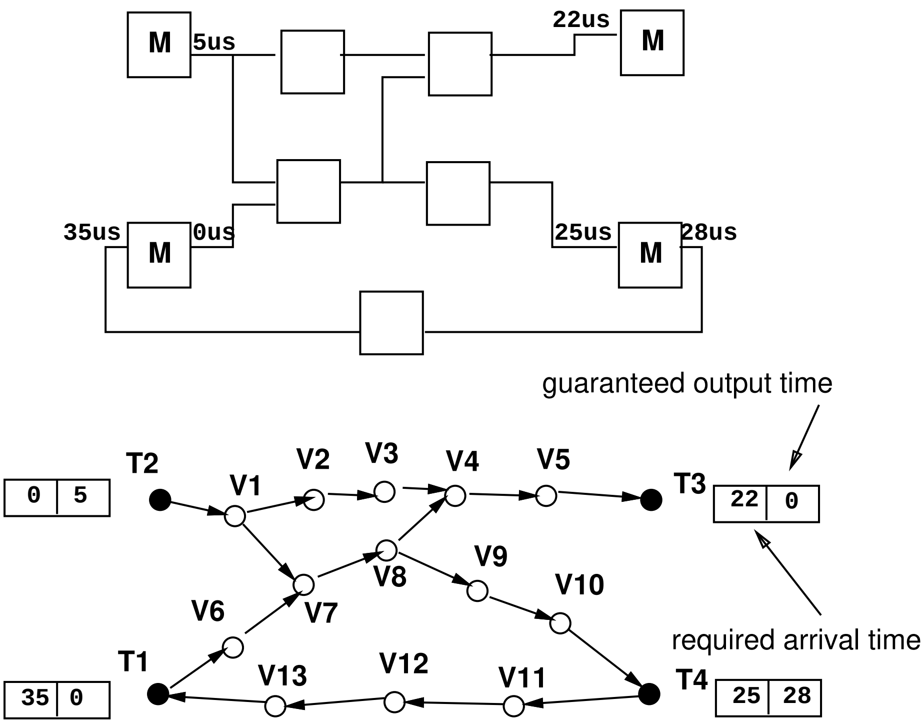


Figure 4.1. Example circuit and its graph model

4.2 Integer programming formulation.

4.2.1 The cost-delay model

An example circuit is presented in figure 4.1 (its upper part). The blocks with **M** inside are memory elements (flip-flops). The circuit is modelled by a directed graph, shown in the lower part of the figure. The vertices V2, V4, V7, V9 and V12 represent combinatorial elements and T1-T4 the *flip-flops*. The other vertices are associated with interconnections. The graph contains two kinds of vertices:

- *regular vertices* represent electronic elements (transistors, gates, modules) or interconnections. For each regular vertex a *tradeoff-curve*, or in other words a discrete function (points $(x, y) \in \mathbb{Z}^2$), expresses the relation between delay on that vertex and the cost (area, power consumption) of realizing such a delay by the electronic element it represents. For all practical purposes this function will be a non-increasing function of the delay (figure 4.2), such as the *trade-off curve* from figure 2.44 in section 2.4.

- *terminal vertices* represent primary inputs and outputs, usually from flip-flops or i/o pins. With each terminal vertex we associate two numbers : the requested arrival time and the output time.

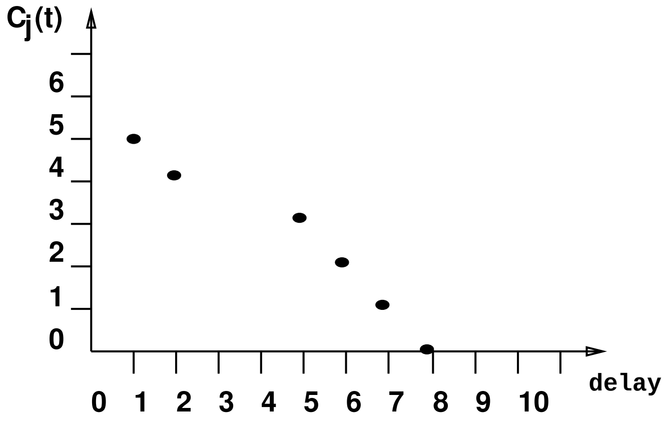


Figure 4.2. Example cost function for a vertex.

4.2.2 Integer programming formulation

The optimization problem can be formulated as follows (recall notation introduced in chapter 3): Find

$$f(x^*) = v(\mathbf{P}) = \min\{f(x) \mid Ax \leq b,\ x \geq 0,\ x \in \mathbb{Z}^n\} \tag{4.1}$$

Let us describe how the program can be constructed. We begin with the constraints. Consider the graph model of a circuit (for example one from figure 4.1). We can calculate a set of paths between all terminal vertices. Each path adds one linear inequality $a_i x \leq b_i$ to the program. Therefore, in the matrix there is one row for every path, and one column for every vertex. a_{ij} is equal to "1" if vertex j exists in path i, "0" otherwise. The right hand side of each constraint is set to the maximum allowed delay in this path.

Example 9 *For our example from figure 4.1 it is as follows:*

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

and b vector :

$$b^t = \begin{bmatrix} 17 & 17 & 22 & 20 & 25 & 7 \end{bmatrix}$$

It is well known that a circuit may have exponential number of paths. Fortunately in reality this number is usually reasonable. If not, several graph preprocessing techniques can be used. One uses the observation that only paths for which the sum of maximal vertex delays is larger than the requested delay have to be considered (so called critical paths). Another trick for reducing complexity of the problem is to treat disjoint subgraphs separately. This is often possible, because logic is naturally divided into groups between *flip-flops*. In the above example the vertices are so divided in two groups (V_{11}, V_{12}, V_{13} form one group, while V_1 up to V_{10} form the other group) that they can be treated separately. Finally, only a subset of all critical paths with a degree of criticality lower than a certain number could be considered.

The cost function of the program can be defined as follows:

$$f(x) = \sum_{j=1}^{n} C_j(x_j) \tag{4.2}$$

where C_j is the discrete cost function for vertex j (figure 4.2). The cost can have two interpretations:

1. Absolute cost. In this case the cost value represents for example total chip area occupied by a given version of the element.
2. Incremental cost. The element's version with minimum cost is used as a reference. Then the costs of all element's versions are:

   $$C_j(t) = C_{abs,j}(t) - C_{abs,min},$$

   where $C_{abs,j}(t)$ is the absolute cost of element version j and $C_{abs,min}$ is the absolute cost of the minimum cost version.

Both cost functions are equivalent in that sense that using them in **IP** program we will obtain the same solution vector x^* (property 3.22 in section 3.5). The second form is advantageous, because the incremental cost values tend to be smaller integer values than absolute ones. Smaller differences between the bounds (estimates of $f(x^*)$) and the optimal solution result. This has positive effect on the efficiency of the branch and bound techniques.

4.3 Solving the timing optimization problem

4.3.1 Calculating surrogate constraints.

In this section we will present our dedicated algorithm solving the program of section 4.2.2. We use the terminology and basic properties of the mathematical programming problems as introduced in chapter 3.

Let $X = \{x \in R^n | x \geq 0, x \in Z^n\}$. Then the timing optimization problem $\mathbf{P}$ can be written:

$$v(\mathbf{P}) = min\{f(x)|Ax \leq b, x \in X\}, \tag{4.3}$$

where $b \in R^m$. For given $w \in R^m, w \geq 0$, we define a *surrogate relaxation*:

$$v(\mathbf{Sw}) = min\{f(x)|wAx \leq wb, x \in X\}, \tag{4.4}$$

This means that the m constraints from eq. 4.3 are replaced by a single one: $wAx \leq wb$. The thus obtained surrogate problem $\mathbf{Sw}$ is a knapsack problem (section 3.4). Problem $\mathbf{Sw}$ is of course a relaxation of $\mathbf{P}$, because $F(\mathbf{Sw}) \supseteq F(\mathbf{P})$. Also the cost functions in both problems are the same. Consequently, $v(\mathbf{Sw}) \leq v(\mathbf{P})$.

We also formulate a surrogate dual problem:

$$v(\mathbf{D_s}) = \max_{w \geq 0}(\mathbf{Sw}) = \max_{w \geq 0} \min_{x \in F(Sw)} \{f(x)|wAx \leq wb, x \in X\}. \tag{4.5}$$

In other words we want to find such dual vector w for which solution of **Sw** is maximal. Each such w provides us with a (lower) bound for the solution of the original problem **P**. The higher this value, the tighter the bound obtained.

Motivation for preferring surrogate relaxation over Lagrangian or linear relaxation can be found in two theorems from literature [Geo69] [MR79]:

Theorem 1 *Let* **Lu** *be the Lagrangian and* **Sw** *the surrogate constraints relaxations for problem* **P** *then for given $w \geq 0, w \in R^m$ we have:*

$$v(\mathbf{Sw}) \geq v(\mathbf{Lu}). \tag{4.6}$$

Theorem 2 *Let $\overline{\mathbf{P}}$ be the linear and* **Sw** *the surrogate constraints relaxations for problem $\overline{\mathbf{P}}$ then for given $w \geq 0, w \in R^m$ we have:*

$$v(\overline{\mathbf{P}}) \leq v(\mathbf{D_s}) \leq v(\mathbf{P}) \tag{4.7}$$

The integer knapsack problem itself, however, is NP-hard, which leads us to considering a relaxation of that problem. A linear relaxation with $0 \leq x \leq d$ (i.e. the problem is bounded and continuous) is

$$v(\overline{\mathbf{Sw}}) = \min\{f(x)|wAx \leq wb, 0 \leq x \leq d\}, \tag{4.8}$$

the linear knapsack problem. Of course, $v(\overline{\mathbf{Sw}}) \leq v(\mathbf{Sw}) \leq v(\mathbf{P})$, and so it makes sense to formulate the following problem:

$$v(\overline{\mathbf{D_s}}) = \max_{w \geq 0} v(\overline{\mathbf{Sw}}). \tag{4.9}$$

This is a much simpler problem than $\mathbf{D_s}$, and can be solved as the multiple-choice linear knapsack program as will be shown in section 4.3.2. We have chosen a subgradient method for solving the surrogate problem as has been proposed in [Dye80]. It searches for an optimal value of vector w by traversing along a quasiconcave cost function. In each step we solve a linear multiple-choice knapsack program and calculate the direction of the next step. [Dye80] contains a proof of convergence for the algorithm.

This approach, however, has one drawback: the obtained solution vector does not necessarily belong to $F(\mathbf{P})$. Of course, it may contain a non-integer component (section 4.3.2), but even truncated down to the nearest smaller integer it may still violate the time constraints. This however is not serious because a clever heuristic can decrement some variables to move x into $F(\mathbf{P})$.

4.3.2 The multiple-choice knapsack program.

The multiple-choice knapsack problem can be formulated as follows:

$$v(\mathbf{ZW}) = \max \sum_{k=1}^{m} \sum_{j \in N_k} c_j x_j, \tag{4.10}$$

with constraints:

$$\sum_{k=1}^{m} \sum_{j \in N_k} a_j x_j \leq b, \tag{4.11}$$

$$\sum_{j \in N_k} x_j = 1,\ k = 1, \ldots, m, \tag{4.12}$$

$$x_j = 0 \; or \; 1,\ j \in N = \bigcup_{k=1}^{m} N_k, \tag{4.13}$$

$$N_i \cap N_j = \emptyset \, \forall i, j \in N_k, i \neq j.$$

So, in (4.10) we have m groups of objects and from each one of them we have to choose only one (4.12) and place it in our knapsack in such a way that the total sum of costs of elements in the knapsack is optimal. Inside the $k - th$ group we have some objects whose indices belong to N_k and which are represented by binary variables (4.13). In each solution only one binary variable in each group will be "1". If we ignore constraint (4.13) we obtain the linear multiple-choice knapsack problem $\overline{\mathbf{ZW}}$.

We now have to convert our timing constraints optimization problem to the knapsack problem. As has been said in section 4.2 each variable in problem P represents the delay at a given vertex. The relation between this delay and the cost of realizing it, is expressed by a time cost function (figure 4.2). Such a function (the *trade-off curve*) has several (usually only a few) points out of which a single one has to be chosen. The relation with the multiple-choice knapsack problem is clear.

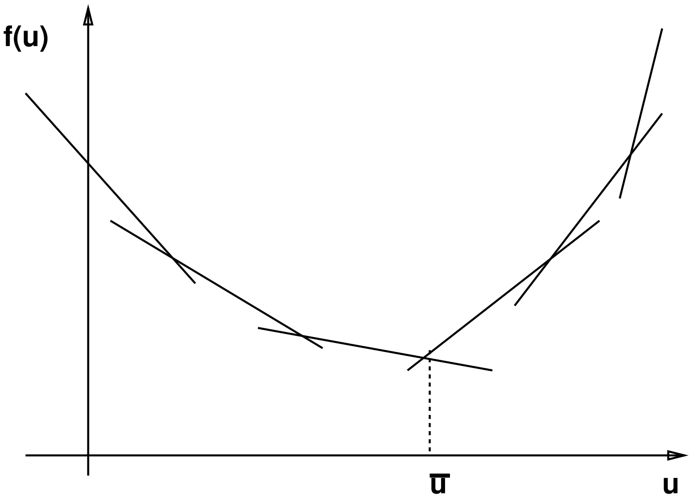


Figure 4.3. Cost function for the dual of the multiple-choice linear knapsack problem.

Each vertex in the graph represents one group of variables in the multiple-choice knapsack problem. For our example from figure 4.2 we get:

$$a_j = \begin{bmatrix} 1 & 2 & 5 & 6 & 7 & 8 \end{bmatrix}$$

$$c_j = \begin{bmatrix} -5 & -4 & -3 & -2 & -1 & 0 \end{bmatrix}$$

where a_j is a vector of a coefficients and c_j a vector of c coefficients for the $j-th$ group. The c coefficients are negative, because we want to find the minimum of (4.10).

To solve the linear multiple-choice knapsack problem we use a very nice algorithm proposed in [Dye82]. The algorithm does not solve the original problem $\overline{\mathbf{ZW}}$ but its dual $\overline{\mathbf{DW}}$. It can be formulated as follows. Let

$$g_k(u) = \max_{j \in N_k} \{c_j - ua_j\} \tag{4.14}$$

Then for a given u the optimal $g_k(u)$ represents the optimal value in $\overline{\mathbf{DW}}$ for the $k - th$ group. Therefore:

$$v(\overline{\mathbf{DW}}) = \min_{u \geq 0} f(u) = \min_{u \geq 0} (ub + \sum_{k=1}^{m} g_k(u)). \tag{4.15}$$

Note that $g_k(u)$ is a convex function of u, and so is function $f(u)$ (figure 4.3). Algorithm of [Dye82] searches for the optimal value of u for which $f(u)$ reaches minimum. It requires only $\mathcal{O}(n)$ operations (n is the number of groups). Having a solution of the dual problem, obtaining a solution to the primal problem $\overline{\mathbf{ZW}}$ is standard (for example described in [Dye82]). In this way a linear relaxation of the original problem $\mathbf{P}$ (eq. 3.1) is being solved, and in most cases our solution will therefore not be integer. It can be shown that it contains at most one non-integer variable. As mentioned in section 4.3.1 we are prepared for such an event.

4.4 The algorithm

Summarizing the above considerations in steps of a procedure yields:

THE TIME BUDGETS METHOD:

1. Make a graph model of the circuit as described in section 4.2.1.
2. Calculate set of paths, eventually apply a preprocessor (as described in section 4.2.2) to reduce their number.
3. Define the constraints matrix A in the way described in section 4.2.2. Create the cost coefficients vector of the multiple choice linear knapsack program (as described in section 4.3.2).
4. Solve the surrogate dual problem to find the value of the aggregation vector w that maximizes the bound value. Use the linear time algorithm from [Dye82] in each iteration.
5. Solve the original problem using its surrogate relaxation with the vector w obtained in the previous step. Resort to branch and bound if necessary. In every iteration use again the algorithm from [Dye82].
6. Retrieve the optimal implementation for every vertex from the solution vector x^* (the binary variables having value of "1").
7. Stop.

The actual implementation of this algorithm is slightly different (efficiency improvements) and will be described in chapter 6.

4.5 Discussions

Assuming that the cost functions for all regular vertices contain exact data the method removes completely the need for the redesign. Unfortunately, it has also some limitations:

- It is not really suitable for optimizations at too low a design level, because the size of the problem can be very large (hundreds of thousands of vertices). In such case the running times may become unacceptable. Still the internal branch and bound algorithm can be stopped after a predefined number of iterations, often providing an optimal solution. This is due to the well known fact that an optimal solution is usually found during one of the first iterations and most time is spent on proving its optimality.

- The method requires quite some work overhead before the real design can start.

- The approach is often not adequate at all. Many problems cannot be formulated as an **IP** with the same structure as the program from section 4.2.2.

Therefore, in the next chapter we propose also another solution, which is wider applicable and more practical. As will be argued, it decreases significantly the risk that a redesign is necessary.

Chapter 5

POSSIBILISTIC METHODS IN SYNTHESIS

Contents

5.1 Introduction

As we learned at the end of chapter 4 the *time budgets* method has some limitations. Therefore in this chapter we describe another solution, which is wider applicable and more practical. Its salient feature is decreases significantly the risk that a re-design is necessary. The approach is based on possibilistic programming [Zad78], a branch of fuzzy mathematical programming (as we saw it in chapter 3). This chapter is devoted to three successful applications of possibilistic programming. We selected problems belonging to different stages in the automated synthesis.

It is conjectured that any combinatorial problem can be formulated as an **IP** although it is not always easy. From our point of view we can divide all these problems into the following categories:

- Problems which can be efficiently solved using a general **IP** solver.
- Problems which cannot be efficiently solved using a general **IP** solver but for which some other efficient algorithms exist.
- Problems for which no efficient exact algorithm is known.

For the first class of problems we can just directly apply the possibilistic programming methods. As examples we will present the problem of *simultaneous scheduling, selection and allocation of functional units*, and the problem of *hardware-software partitioning of embedded systems*. The first one is an example of a problem with a solution known from literature. We extend the approach by taking imprecise information into account. The second presented problem, best to our knowledge, has not been addressed in this way before (neither for precise nor imprecise case).

We believe that also for the second class of problems the possibilistic programming can be applied. As an example we will present the solution to the problem of *retiming* of synchronous circuitry with imprecise delays.

5.2 Scheduling, selection and allocation of functional units

5.2.1 Introduction

This section is devoted to one of the subtasks within the high-level synthesis system - the *simultaneous scheduling, selection and allocation of functional units* (shortly the *scheduling* problem from now on). We will show how this problem can be solved if the information about available functional units is imprecise. The "classical" (deterministic, non-possibilistic) version of the problem can be formulated as an **IP** and efficiently solved using a general **IP** solver. Therefore our extension is quite straightforward. The possibilistic methods from chapter 3 can be directly applied. We will show that the temporary crisp programs, which we obtain in course of solving the possibilistic program, retain the good properties of the original formulation. Therefore they can also be solved by use of a general **IP** solver. We will start with a textual formulation of the "classical" problem and present its **IP** formulation. After that the "possibilistic" extension will be described.

5.2.2 Problem formulation

The organization of a typical high-level synthesis system was described in section 1.3.2. The problem that we consider here is the most important one in every such system. Let us start with a textual formulation:

PROBLEM
Given:

1. DAG (Directed Acyclic Graph) representing the partial order between the operations in an algorithm to be synthesized.
2. Interface constraints (synchronous or asynchronous).
3. User constraints.
4. Specifications of available functional units.

Produce a schedule, by mapping each code operation to a time (maintaining the partial order among operations), and assign each operation to a functional unit. Simultaneously select and allocate functional units, and schedule operations to minimize the area cost function.

5.2.3 Solution

The first attempts to solve scheduling problems of high-level synthesis date from the 70's [Bak74]. An efficient **IP**-based solution to the scheduling problem, as formulated in section 5.2.2, was first proposed by C. Gebotys et.al. [GE90]. In their approach most constraints of the **IP** form integral facets for the system of linear inequalities. Better bounds on variables and decrease in computation time result. Even quite large examples can be solved within reasonable CPU time.

Here follows the integer programming formulation of the problem. Let us start with definitions of used variables and symbols:

$x_{i,j,k}$ - binary variables which assume value 1 if code operation k has been scheduled at the j-th cstep (clock cycle), using functional unit i, 0 otherwise.
C - execution time of functional unit
L - latency time (minimum time between successive data input accepted by the unit)
$k \in Op(C, L)$ - means all codes which could be implemented by a unit with values C,L.
$j \in R(k)$ - means all csteps to which k-th code can be assigned, calculated using *critical path method* [Fou81].

$$\min f(x_{i,j,k}) \tag{5.1}$$

Minimize this linear or piecewise linear function of variables, subject to:

- Operation Assignment Constraints:

$$\forall_k \left[\sum_i \sum_{j \in R(k)} x_{i,j,k} = 1 \right] \tag{5.2}$$

They ensure that each code operation will be assigned to one control step and functional unit.

- Functional Unit Constraints:

$$\forall_{i,j} \left[\sum_{k \in Op(C,L)} \sum_{\substack{j_1=j \\ j_1 \in R(k)}}^{j+L-1} x_{j,j_1,k} \leq 1 \right] \tag{5.3}$$

Prevent that more than one code operation is assigned to the same functional unit at the same control step.

- Precedence Constraints:

$$\forall_{\substack{k_2 <\bullet\ k_1 \\ j \in R(k_z) \\ z=1,2}} \left[\sum_i \left(\sum_{\substack{j_1 \leq j+C_2-1 \\ j_1 \in R(k_1)}} y_{i,j_1,k_1} + \sum_{\substack{j \leq j_2 \\ j_2 \in R(k_2)}} y_{i,j_2,k_2} \right) \leq 1 \right] \tag{5.4}$$

Prevent that an operation, $k2$ will be scheduled after operation $k1$ whenever there is a partial order between these operations such that $k2 < \bullet\ k1$ (that means $k2$ must be scheduled before $k1$).

- Interface or user specified constraints.

5.2.4 Imprecise data about functional units

Assume now that the information about the design is imprecise. This imprecision may lay in the area of functional units, in their timing parameters, or in the user/interface constraints. Examples 6, 7, 8 in chapter 3 demonstrate it. Following our "possibilistic" strategy from chapter 3 we use the possibility distributions to model this imprecision. Consequently all these quantities will be defined in terms of triangular fuzzy numbers. Performing this simple substitution we obtain a possibilistic integer program (**PIP**). The program will contain some fuzzy numbers in the cost function (if at least one unit has imprecise area), and/or some fuzzy numbers in the constraints (if some timing parameters of our units or user/interface constraints are imprecise).

The cost function can be formulated as any linear or piecewise linear function of the variables.

$$\min \sum_i \sum_k \sum_{j \in R(k)} \tilde{s}(i) x_{i,j,k} \tag{5.5}$$

where : $x_{i,j,k}$ - the same as in section 5.2.3,
$\tilde{s}$ - vector with imprecise costs of functional units,

This first cost function (eq.5.5) is very simple and adds no extra constraints. To handle the possibility of using many units of the same type we add a small constant to the cost of every next unit used. This function has the advantage that it simultaneously optimizes for time and area.

$$\min \sum_t \tilde{s}(t) I^t \tag{5.6}$$

$$\sum_{k \in t} \sum_{i \in t} x_{i,j,k} \leq I^t, \forall_{t,j}$$

where : $x_{i,j,k}$, $\tilde{s}$ - the same as in eq. 5.5,
I^t - number of units of type t used,
$k \in t$ - means all codes which can be implemented by units of type t,
$i \in t$ - means all units of type t.

The second function (eq.5.6) has the advantage that it optimizes the area of the obtained architecture.

Because the $\tilde{s}$ coefficients are fuzzy numbers, the cost functions are also fuzzy. Recall now from section 3.9 that in process of solving the **PIP**, from this fuzzy cost function other crisp (non-fuzzy) cost functions are generated (equations 3.31 in the method 1, and equation 3.35 in method 2). It is obvious that these crisp cost functions will remain linear. Also the cost function of the "alpha" program (eq.3.34) in method 1 is clearly linear. The increase in the number of constraints in the "alpha" program is neglectable (only three additional constraints, given by equations 3.32 and 3.33).

Concerning the constraints, we learned in chapter 3 that all temporary crisp programs of method 1 and 2 will have the same set of constraints as the original problem. Examining carefully our program, we notice that no timing parameters of the functional units appear as coefficients on either side of the equations. Consider for instance equation 5.4. As can be seen, we have there two timing parameters C and L. They control the ranges over which the summing is performed. We conclude that these constraints are non-linearly dependent on the values of C and L. The only way to handle the imprecision of such parameters is to apply the concept of the most likely value (introduced at the end of section 3.9). This way we can obtain values which can be used to generate the crisp constraints. It is obvious that the constraints will retain their structure. The same is true for the user/interface constraints. For example, a minimum timing constraint between the scheduled csteps of code operations k_1 and k_2 ($k_1 \lessdot k_2$) can be formulated as follows. If the number of csteps between k_1 and k_2 should be larger than T, we have to define the following constraints:

$$\forall_{i,j_1} \left[\sum_{\substack{j > j_1 \\ j_1 \in R(k_1)}} x_{i,j,k_1} + \sum_{\substack{j \leq j_1 + T \\ j_1 \in R(k_2)}} x_{i,j,k_2} \leq 1. \right] \tag{5.7}$$

As can be seen the situation here is exactly the same as in the case of precedence constraints. The proper structure of the constraints will therefore be preserved.

At the end we would like to mention another extension to the original formulation from section 5.2.3,namely including support for multiple functions in functional units. What do we mean by that? In the original formulation every code operation can be mapped to only one type of the functional unit. In reality many functional units are able to perform several operations (they implement several functions). For instance, every ALU implements at least integer addition, subtraction, multiplication and division, not mentioning several boolean (bitwise) functions. Another observation is that several functional units may offer the same function, at different costs. Figure 5.1 gives an example. The dotted hyper-edges connect code operations with all functional units that offer the given function. To handle this obviously very important situation, we augment our program. This extension is however straightforward and causes only a small increase in the number of constraints. For example, every constraint of eq. 5.4 has to be repeated for each type of functional unit capable of providing the functionality of $k2$. Since the structure of the constraints remains the same, the good properties are preserved.

5.3 Hardware/software partitioning of embedded systems

5.3.1 Introduction

In this section we will apply possibilistic programming to another problem. Also this problem can be efficiently solved using a general **IP** solver. Since no, best to

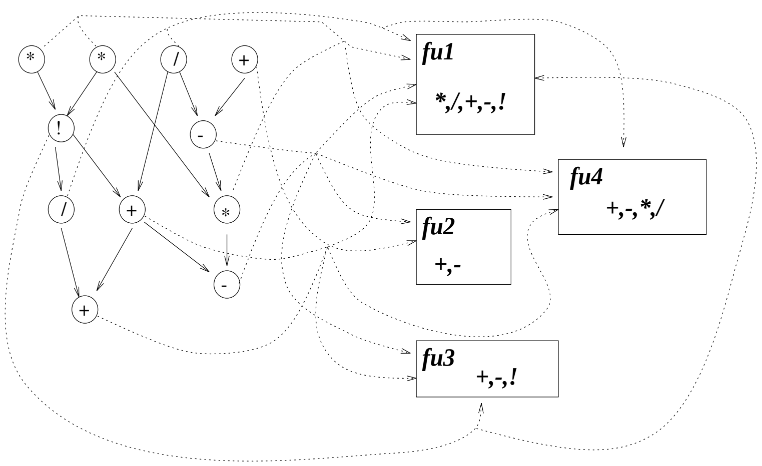


Figure 5.1. Functional units implementing multiple functions.

our knowledge, solution to this problem exists, we first have to propose it. It turns out to be interesting. As we will see in section 5.3.4, the fuzzy constraints of the program may contain possibilistic numbers on both sides of the inequalities. The constraints depend linearly on these numbers. Recall that in the *scheduling* problem from section 5.2, the constraints were non-linearly dependent on the values of the timing parameters. This means that a different possibilistic approach to handle the imprecision in the constraints has to be used.

5.3.2 Preliminaries

In this context *embedded systems* are combinations of hardware with general purpose computational capabilities and more dedicated modules. Together they perform a function carefully partitioned in software and hardware to obtain the optimum trade-offs between the various performance metrics. These types of implementations have become increasingly popular as advances in ic-technology and processor architectures allow for flexible computational parts and high-performance modules integrated on a single carrier. What is needed is an interactive environment that supports the designer in transforming an algorithmic specification into a suitable implementation [TSV94]. A central problem in automated co-synthesis of hardware and software in embedded systems is the hardware/software partitioning. Two different approaches exist. A *software-oriented* approach has been implemented in the **COSYMA** system [EHB93]. It starts from an all software solution and iteratively selects operations to be moved to hardware until all timing constraints are met. Another co-synthesis approach [GCM92], *hardware-oriented*, starts from a complete hardware description and iteratively selects components, which can be implemented in software without violating the performance constraints. Our partitioning approach is non-iterative and has the following features:

- The program accepts as input a specification in a high level language (at this moment a superset of ANSI C).

- The model is a quite straightforward mapping of the input language statements into a partitioning graph.

- It supports parallel flows of execution within the program (by adding extra constructs), loops and conditional statements.

- All information about functions and variables can be imprecise, by giving the coefficients a possibility distributions [Zad78]. This is essential! Otherwise we could be optimizing a model which is quite far from reality, or with a high probability of not meeting the constraints. We choose for possibility distributions because they model our knowledge about the design parameters better than for example probability distributions. Another consequence of this choice is that the computational complexity of the used partitioning algorithms does not have to increase. Also generating possibility distributions is much easier than deriving probability distributions.

- The partitioning program runs fully automatically, but support for user interaction and iterative improvement is provided. This allows the user to experiment with several implementation alternatives.

- In addition to partitioning, the program also automatically selects a main processor from the list of CPUs available.

- The approach allows for a large degree of flexibility - many kinds of constraints may be specified by the user. Also several optimization criteria can be investigated. It is easy to take into account many important design aspects and user preferences.

- Functional units can execute simultaneously with the main CPU. This extends the approach presented in [EHB93], where the processor activates a co-processor and then waits idling until it delivers a result. Without this extension it would be impossible to take advantage of the intrinsic parallelism usually present in the system specification.

We would like to emphasize that our approach differs substantially from previous results reported in the area of synthesis of application-specific multiprocessor systems [PP92]. Here follow main differences:

- Our program does not perform scheduling simultaneously with partitioning (in section 5.3.3 we say why). Thanks to that, the program is smaller and contains only binary/integer variables. This results in very short solving times. For instance it takes only 4 second on HP735 computer to solve an example with 38 nodes and 66 edges. The SOS system [PP92] needed 6416 minutes of CPU for an example with only 9 nodes, due to the mixed-integer nature of their formulation.

- The program partitions not only functions to be implemented, but also variables used in the program. Thanks to that, a better modelling of data transfers is obtained. For every function, variable or data transfer several implementation options are available.

There exist also several graph-theoretical approaches to the assignment problems in parallel and distributed computing. In [Sto77] Stone shows how to efficiently solve the assignment problem for a two-processor system by mapping it into a network flow problem. Also shortest tree algorithms have been employed. In [Bok81] Bokhari presents his method solving the assignment problem for unlimited number of processors in polynomial time under the constraint that the graph of the modular program is a tree. In more general case of so called series-parallel graphs, an algorithm developed by Towsley [Tow86] can be used. All these methods lack however flexibility in taking into account design constraints. No general hard timing constraints are handled.

5.3.3 Embedded system modelling

Our model is based on the following assumptions:

- An embedded system consumes some data, performs some operations on them and generates some data. The software component of the embedded system is thought to consist of a set of variables and exclusively/concurrently executing routines, called *basic blocks*. A set of single/multiple activations of basic block(s) necessary to process a given amount of input data is a *run* of the system.

- The total time (in clock cycles) spent by the processor in one *run* of the system is called the *latency* of the *run*.

- The set of basic blocks is a result of the clustering operation on all operations. This clustering can be *locked* by the user or done automatically. In the first case the highest level of the calling hierarchy in the input program is chosen as the partitioning boundary. Hardware software partitioning on too low level does not seem to be promising because of the interfacing overheads.

- The design contains at least one processing unit, selected from the list of available processor templates. (In our formulation we assume exactly one processor.)

- One of the basic blocks, called the $Supervisor$ represents the main scheduling engine of the system. It accommodates all operating system and program overheads. It must be implemented in software.

- Every basic block can have several implementation options, some of them in hardware. They may have form of multi-dimensional *trade-off curves*, as introduced in section 2.4.
- Functional units/co-processors can perform their function independently of and simultaneously with the main processor.
- Embedded system is *software dominated*. This means that majority of the basic blocks will remain in software and that the critical path will be also located there. The consequence of this assumption is that we do not have to perform scheduling simultaneously with the partitioning.

The basic blocks are vertices of a directed graph $G =< V, E, C >$ that represents the functionality of the system to be designed. The edges $e \in E$ represent data transfers between basic blocks ($e : u \xrightarrow{e} v$ means that some data is transferred from the basic block u to the basic block v). The edges $e \in C$ represent control flow. Both, vertices and edges, have a number of performance and cost parameters associated with the various realization options. The values of these parameters may be real or possibilistic numbers. They can be used as coefficients for **IP** formulations of our problem. The coefficients for vertices and edges are presented in tables 5.1 and 5.2. Further there is also some general design information, for example as in table 5.3.

Coefficient	Description
F_i	execution rate of the basic block i (how many times it executes during one run)
$T_{i,j}$	latency of the j-th hardware implementation of the basic block i, given in real time units
$L_{i,p}$	latency of the software implementation of basic block i on processor p, given in clock cycles.

Table 5.1. The vertex coefficients

Coefficient	Description
$C_{e,j}$	size of the code for an interface on the edge e if it is implemented in the j-th way
$L_{e,j}$	latency of the j-th implementation of the interface at the edge e
F_e	execution rate of the interface on edge e (the same as at vertices)
D_e	amount of data to transfer through the edge e during one transaction

Table 5.2. The edge coefficients

Coefficient	Description
P_j	the cycle time of the processor j, given in real time units (for example in ns)
A_u	size of the functional unit of type u
A	area available, for example on a master image
C	upper bound on the code size, for example in a fixed rom

Table 5.3. General design information coefficients

Example 10 *In figure 5.2 the source code of a part of the WISSCE receiver [Gla94] is presented. This standard ANSI C code is augmented with $<$ and $>$ brackets to denote that all statements within them can be executed in parallel. The system implementing this algorithm will continuously repeat operations specified by this diagram. The set of statements and variables from this figure can be directly mapped into the set of basic blocks. Basic blocks from the two compound-statements can execute concurrently since they are placed within triangular braces. The latency of a single* run *may be imprecise, because the exact time that some operations take is unknown or may vary. For example $readSample$ usually takes 1 cycle, but it can take as much as 3 cycles. Also the latency of other operations may be data dependent (for example the* **limit** *function).*

Example 11 *In figure 5.3 the graph model of the receiver from the figure 5.2 is presented. The oval vertices represent routines and the square ones the variables. The continuous arrows are used for the control flow, while the ones with dotted lines for the data flow. Only the coefficients assigned to one vertex called "bb5" are depicted. As can be seen some of them are crisp values representing the well-known values for the coefficients, some are triangular fuzzy numbers capturing the uncertainty.*

```
bit acq=0;                /* in acq-stage? (default value!) */
kasgen gen;               /* kasami-code generator */
register IQ sample;
                          /* read input from ad-converter */
        sample = readSample();
                          /* shift pn-code generator */
        shiftPNgen(&gen, trackCntr, acq);
        <                 /* beginning of the parallel block */
        {
                          /* promptCode only in this loop */
            register byte promptCode;
            register IQ promptSample;
            register short pPointer;
                          /* init pointer and sample */
            pPointer = i;
            promptSample = sample;
                          /* get prompt-code */
            promptCode = getPrompt(&gen);
                          /* do prompt-despreading */
            promptSample = IQmpy(promptSample, promptCode);
                          /* do filtering */
            promptSample = filter(promptSample);
                          /* do limiting and put in buffer */
            promptBuf1[pPointer] = limit(promptSample);
        }
        {
                          /* trackCode only in this loop */
            register byte trackCode;
            register IQ trackSample;
            register short tPointer;
                          /* init pointer and sample */
            tPointer = i;
            trackSample = sample;
                          /* get early-late code */
            trackCode = getEarly(&gen) - getLate(&gen);
                          /* do early-late despreading */
            trackSample = IQmpy(trackSample, trackCode);
                          /* do limiting and put in buffer */
            trackBuf1[tPointer] = limit(trackSample);
        }
      >                   /* end of the parallel block */
```

Figure 5.2. C source code of a part of the WISSCE receiver.

5.3.4 IP formulation

Integer programming **IP** has a number of salient features. When it finishes it is guaranteed to have reached a globally optimal solution. It offers quite some flexibility in formulating combinatorial solutions. One may even conjecture that any combinatorial problem can be formulated as an **IP** although it is not always easy.

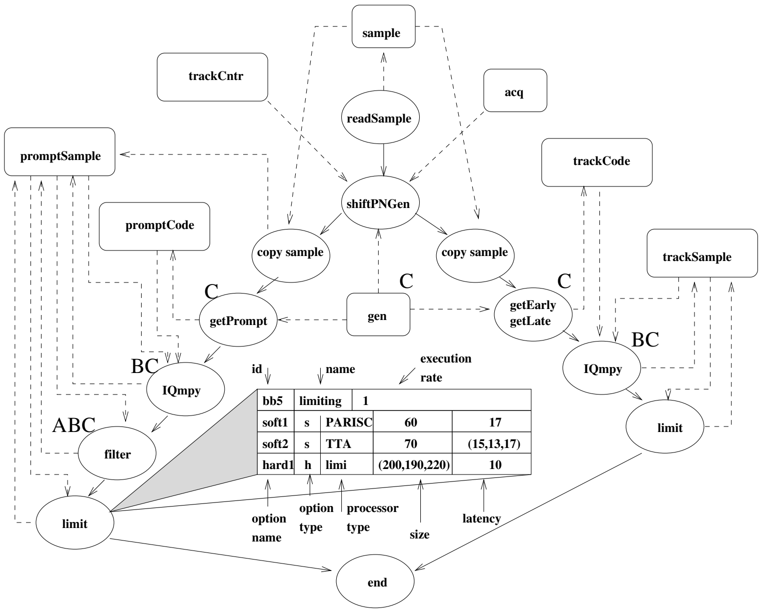


Figure 5.3. The graph model of the WISSCE receiver.

There is also a quite large degree of flexibility in cost functions. **IP**'s can be adapted to possibilistic theory in a straightforward manner, without increasing the number of variables. However, integer programming in general is $\mathcal{NP}-hard$. Many years of experience in the Operations Research have revealed that applying general **IP** methods is only recommended when:

- the problem has some kind of special structure,
- the size of the problem is moderate.

In our case the first is true, because we formulate the program in similar manner as for the *scheduling* problem from section 5.2. Since the partitioning is performed on the result of the clustering of all operations the second condition is also true (we elaborate on this in section 5.3.4.3). In this section we will formulate the problem and estimate its size.

The general symbols used are presented in table 5.4, while all decision and temporary variables are explained in table 5.5.

5.3.4.1 Constraints

The program contains the following constraints:

1. Implementation selection constraints. Every basic block i has to be implemented either in software or in hardware:

$$\forall_{i\in V} \quad \left[\sum_{p\in P} s_{i,p} + \sum_{j\in H_i} h_{ij} = 1\right], \tag{5.8}$$

$$\forall_{e\in E} \quad \left[\sum_{j\in I} m_{e,j} = 1\right] \tag{5.9}$$

Symbol	Description
P	set of available processors
H	set of hardware implementations
S	set of software implementations
U	set of available hardware units
I	set of possible ways of implementing interfaces. Some of them are between software, between hardware, other between basic blocks of mixed types
I^p	a subset of I in which the processor p participates

Table 5.4. Symbols used in the **IP** formulation

Variable	Description
$h_{i,j}$	binary variable which assumes value 1 if basic block i is implemented in hardware, using j-th implementation option
$s_{i,j}$	assumes value 1 if basic block i is implemented in software, on a processor of j-th type
p_i	assumes value 1 if we use processor i
s_e	temporary binary variable, assumes value 1 if it is necessary to provide interface in software at the edge e
h_e	temporary binary variable, assumes value 1 if it is necessary to provide interface in hardware at the edge e
$m_{e,j}$	binary variable which assumes value when 1 the j-th interfacing scheme on edge e is used ($j \in I$)
n_u	number of functional units of type u used
L	total time spent by the processor
C	software size (how much memory does the code occupy)
A	the area of the extra hardware

Table 5.5. Decision and temporary variables used in the **IP**

2. "Interface type selection" constraints. There are three kinds of interfaces: software, hardware and mixed. A software interface for edge $e : u \stackrel{e}{\to} v$ is necessary if basic blocks u and v are implemented in software. This can be expressed as:

$$\forall_{e \in E} \left[s_e = \sum_{j \in P} s_{u,j} \circ \sum_{j \in P} s_{v,j} \right] . \tag{5.10}$$

Two basic blocks in hardware generate a hardware interface:

$$\forall_{e \in E} \left[h_e = \sum_{j \in P_u} h_{u,j} \circ \sum_{j \in P_v} h_{v,j} \right] . \tag{5.11}$$

$\circ$ is the boolean *and*. This kind of constraint can be replaced by the following two arithmetic constraints:

$$\prod_{j \in N_k} x_j = y_k \quad \Rightarrow \quad \left\{ \begin{array}{ll} \sum_{j \in N_k} x_j - y_k & \leq |N_k| - 1, \\ \frac{1}{|N_k|} \sum_{j \in N_k} x_j - y_k & \geq 0. \end{array} \right\}$$

where $\prod$ is a boolean product, N_k is the index set of all variables appearing in the product.

3. Interface assignment constraints. If an interface on edge e is necessary then select exactly one implementation scheme for it:

$$\forall_{e \in E} \quad \left[\sum_{s \in S} m_{e,s} = s_e \right] \tag{5.12}$$

$$\forall_{e \in E} \quad \left[\sum_{h \in H} m_{e,h} = h_e \right] \tag{5.13}$$

Note that if $s_e = 0$ and $h_e = 0$ then because of the constraint (eq. 5.9) one of the mixed options (interface between hardware and software) has to be selected.

4. Processor selection constraints. They guarantee that only one processor is selected:

$$\sum_{j \in P} \quad p_j = 1, \tag{5.14}$$

$$\forall_{i \in V} \forall_{j \in P} \quad [p_j \geq s_{i,j}] \tag{5.15}$$

$$\forall_{j \in P} \quad \left[p_j \leq \sum_{i \in V} s_{i,j} \right] \tag{5.16}$$

$$\forall_{p \in P} \forall_{e \in E} \quad \left[p_j \geq \sum_{j \in I^p} m_{e,j} \right] \tag{5.17}$$

The second and third group of constraints ensures that p_j is equal 1 only if at least one of basic blocks is implemented in software on processor j. The fourth group guarantees that p_j is equal 1 only if at least one of the interfaces is implemented on processor j.

5. *Latency* of a *run* constraint (total time spent by the processor):

$$\sum_{i \in V} \sum_{s \in S} F_i L_{i,s} s_{i,s} + \sum_{e \in E} \sum_{j \in I} F_e L_{e,j} m_{e,j} \leq L \tag{5.18}$$

The first term includes all software basic blocks. The second one represents time spent in the interfaces.

6. Software size constraints. We assume that the code for every interface on edge e must be generated separately:

$$\sum_{i \in V} \sum_{p \in S} C_{i,p} s_{i,p} + \sum_{e \in E} \sum_{j \in H \cup S} C_{e,j} m_{e,j} \leq C \tag{5.19}$$

7. The size of the extra hardware. We assume that several basic blocks can be mapped to the same hardware unit.

$$\sum_{u \in U} A_u n_u \leq A, \tag{5.20}$$

$$\forall_{u \in U} \quad \forall_{i \in Op(u)} \quad [h_{i,u} \leq n_u]\,. \tag{5.21}$$

   In the equations above binary variable n_u takes value 1 if at least one basic block has been mapped to it. $Op(u)$ represents the set of all basic blocks that can be assigned to unit type u.

8. Many other constraints taking into account power consumption, number of pins, size of the code, design and prototyping time can be easily formulated.

5.3.4.2 Cost functions

On the basis of the constraints from the previous section the following programs among others can be constructed and solved:

- Minimize extra hardware area under total latency constraint:

$$\min \quad A \tag{5.22}$$

$$\bar{B}x \leq b \tag{5.23}$$

$$L \leq L_{max} \tag{5.24}$$

  where $\bar{B}, b$ encapsulate all constraints from the previous section, x all decision variables.

- Minimize total latency under constraint on the size of the ASIC part:

$$\min \quad L \tag{5.25}$$

$$\bar{B}x \leq b \tag{5.26}$$

$$A \leq A_{max} \tag{5.27}$$

5.3.4.3 Program size estimates

There will be $|H_i| * |V|$ $h_{i,j}$ variables, $|S_i| * |V|$ $s_{i,j}$ variables, $|E|$ s_e, h_e variables, $|E| * |I_e|$ $m_{e,j}$ variables, $|U|$ A_u and n_u variables and $|P|$ p_i variables.

Example 12 *For the "receiver", $|P| = 1, |V| = 21, |E| = 20, |H_i| = 1, |S_i| = 1, |I| = 3$ we get 159 binary variables. The program contains 195 constraints. Solving time in CPLEX IP solver [CPL93]$was \ll 1sec$.*

Because P, H_i, S_i and I will be usually small positive numbers ($\ll 10$) the size of the program will be proportional to the number of vertices and data transfer edges. Our experience learns that when the size of the program is even about 500 binary variables the optimal solution can be reached within seconds. Also larger examples were optimized in reasonable time.

5.3.5 Handling imprecision

Assume now that some parameters in the just formulated program are imprecise. Possible sources of their imprecision are:

- execution rate F_i of a basic block may be data dependent,
- latency of a hardware implementation $T_{i,j}$ may be data dependent; also it may be not exactly known if the unit has not been synthesized yet,
- size of the code $C_{e,j}$ depends on heuristic optimizations performed by the compiler,
- execution rate of an interface F_e may be data dependent,
- amount of data to transfer through an edge D_e may differ if for example we transfer a list,
- exact size of a functional unit A_u is unknown if the unit has not been synthesized yet,
- available area A may be not exactly known if the chip has to be shared with other circuit, which exact size is not yet known,
- upper bound on the code size C may be not exactly known if the ROM has to be shared with some other code of unknown size.

By substituting imprecise numbers into the integer program from section 5.3.4 we obtain a possibilistic integer program (**PIP**). Now the methods 1 or 2 (described in chapter 3) can be directly applied to solve it. Notice that all imprecise quantities appear as coefficients on either side of the constraints. Therefore, every imprecise constraint can be replaced by three crisp ones, as given by equation 3.39. Handling of imprecise cost functions is also standard.

5.4 Retiming circuitry with imprecise propagation delays

5.4.1 Introduction

This section is devoted to completely different kind of problem, as compared to previous two, described in sections 5.2 and 5.3. The difference is that it cannot be efficiently solved by use of a general **IP** solver. Fortunately a dedicated polynomial time algorithm is known, which can be used to solve it. We will show that also here the possibilistic programming can be successfully applied, without increasing the computational complexity. We start from the "classical" (non-possibilistic) version with a polynomial time algorithm solving it. After that the possibilistic extension will be described.

Retiming is a circuit transformation in which memory is added at some points in the circuit and removed from others in such a way that the sequential behavior of the circuit is preserved. It can be applied to efficiently optimize circuits for speed, assuming that the delays within a circuit are exactly known.

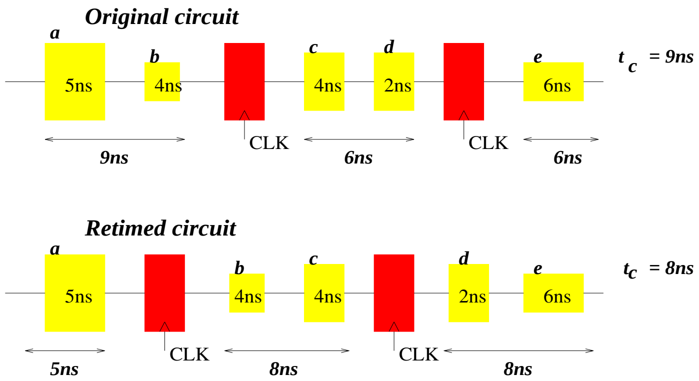


Figure 5.4. "dwareg" before place & route.

Example 13 *We want to minimize the clock cycle ϕ of a circuit called "dwareg", given in figure 5.4. Initially $\phi = 9ns$, which can be calculated by finding the longest path between flip-flops or primary inputs/outputs. A faster circuit can be obtained by shifting both latches backward (see figure 5.4). Now the circuit should have $\phi = 8ns$. After applying logic synthesis tools to the combinatorial parts of the circuit, and placement and routing, ϕ turns out to be $11ns$. This is due to fact that the delays of the combinatorial block* a,b,c *came out at different values (see figure 5.5). The original configuration would then have $\phi = 9ns$.*

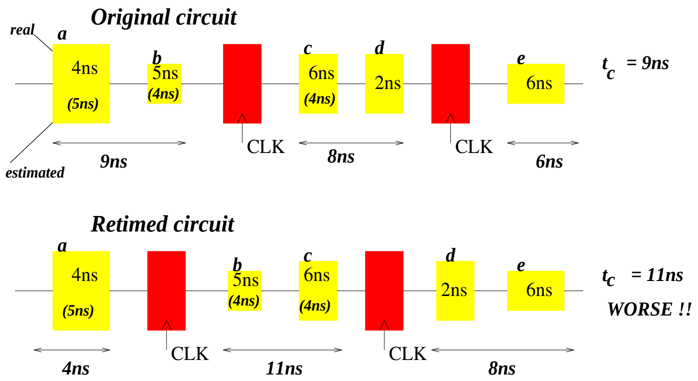


Figure 5.5. "dwareg" after place & route. The previously estimated delays are shown in braces.

5.4.2 The "Classical" model

Retiming applies to any edge-triggered, single-phase, synchronous circuit. This can be modelled as finite, directed multigraph $G =< V, E, d, w >$, with weights w on the edges representing the number of latching elements, and weights on the vertices $d \in \mathbf{R}^+$ representing the delay of the corresponding combinatorial circuit. For recalling the fundamental results of Leiserson et.al. (see [LS91]) we need some derived quantities:

$$
\begin{aligned}
W(u,v) &= \min\{w(p) : u \overset{p}{\rightarrow} v\}, && (5.28)\\
D(u,v) &= \max\{d(p) : u \overset{p}{\rightarrow} v, \quad w(p) = W(u,v)\}. && (5.29)
\end{aligned}
$$

where $p = v_0 \xrightarrow{e_0} v_1 \cdots \overset{e_{k-1}}{\rightarrow} v_k$ is a path in the graph , $d(p) = \sum_{i=0}^{k} d(v_i)$ and $w(p) = \sum_{i=0}^{k-1} w(e_i)$. The quantity $W(u,v)$ is the minimum number of latching elements on any path from vertex u to vertex v. A path p for which $w(p) = W(u,v)$ is a called a *critical path* from u to v. The quantity $D(u,v)$ represents the maximum total propagation delay on any critical path from u to v.

DEFINITION 4 *The clock period ϕ of the circuit modelled by $G < V, E, d, w >$ is:*

$$\phi(G) = \max\{d(p) : w(p) = 0\}. \quad (5.30)$$

By retiming we can obtain all equivalent synchronous circuits. The statement that enables the efficient determination of the minimum clock period over all these circuits is:

THEOREM 1 *Let $G =< V, E, d, w >$ be a synchronous circuit, let c be an arbitrary positive real number, and let r be a function from V to the integers. Then r is a legal retiming of G such that its clock period $\phi(G_r) \leq c$ if and only if:*

$$
\begin{aligned}
r(u) - r(v) &\leq w(e)\ for\ every\ edge\ u \overset{e}{\rightarrow} v, and && (5.31)\\
r(u) - r(v) &\leq W(u,v) - 1\ for\ all\ vertices && (5.32)\\
& \quad u, v \in V such\ that D(u,v) > c. &&
\end{aligned}
$$

The first group of constraints ensures that the retiming is legal, in the sense that no edge gets a negative number of latching elements assigned: $w_r(e)$ is a nonnegative integer for each edge $e \in E$. The second group of constraints guarantees that every path with delay greater than c contains at least one latching element. With the additional fact that the minimum clock period is among the values $D(u, v)$ defined by eq.5.29 the correctness of the following algorithm is guaranteed. ALGORITHM LS (Leiserson-Saxe):

1. Find the set of possible clock periods to be tried.
2. Sort the elements in the set calculated in step 1.
3. Perform binary search among the candidate values for the minimum achievable clock period. For each value apply Bellman-Ford or FEAS algorithms, as presented in [LS91], to test whether there exists a mapping r which satisfies set of constraints of theorem 1. This is equivalent to testing whether a potential clock period c is feasible.

The complexity of this algorithm is $\mathcal{O}(|V|^3 \lg |V|)$ or $\mathcal{O}(|V||E| \lg |V|)$, depending on the choice between Bellman-Ford and FEAS.

5.4.3 Imprecise propagation delays

When the combinatorial delays are not exactly known, applying the standard retiming algorithm of [LS91] may in the end lead to unnecessary violation of timing constraints and cause expensive redesigns. Instead of a single nonnegative number we assign a fuzzy number to each vertex: $\tilde{d} \in X$, where X is the space of triangular fuzzy numbers. Figure 5.6 shows an example. The vertices still represent the combinatorial elements of the circuit, but each vertex $v \in V$ now has a triangular fuzzy number, $\tilde{d}(v)$, as its weight, modelling the nonnegative imprecise propagation delay of the corresponding element. The extra vertex v_h represents interfaces with the external world, and it is given zero propagation delay. The interpretation of the edges and their weights is the same as in the classical model. How to handle multiple fanout is described in [LS91].

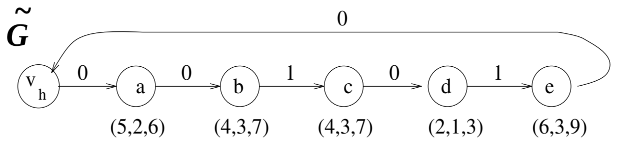


Figure 5.6. The graph model of the "dwareg" circuit.

The problem of finding a retiming r of the graph $\tilde{G}$ which optimizes its fuzzy/imprecise clock-period $\tilde{\phi}$ can be formulated as the general integer **MOLP** (analogous to the program of eq. 3.30):

$$\begin{aligned} \max z_1 &= \phi^m - \phi^l \\ \min z_2 &= \phi^m \\ \min z_3 &= \phi^u - \phi^m \end{aligned} \tag{5.33}$$

$$s.t.\ \tilde{\phi} = (\phi^m, \phi^l, \phi^u) \in Y,$$

where Y is the set of feasible imprecise clock periods. The methods described in section 3.9 can then be used to solve it. However, integer programming is in the general case $\mathcal{NP} - hard$. If it was possible to extend the standard approach while preserving its low computational complexity, an efficient retiming method with reduced risk for redesign would become available. We will show that this is possible. First however, some definitions are in order:

DEFINITION 5 *Let $f : X \mapsto \mathbf{R}^+$ be a linear mapping with positive coefficients, $\tilde{G}$ =<$V, E, \tilde{d}, w$> a retiming graph with imprecise delays. Then the crisp graph of $\tilde{G}$ under f is a graph G_f =<V, E, d_f, w>, $d_f \in \mathbf{R}^+$ obtained from $\tilde{G}$ by mapping all fuzzy delays $\tilde{d}$ of $\tilde{G}$ into crisp values: $d_f = f(\tilde{d})$.*

We can of course apply classical retiming to G_f =<V, E, d_f, w>. In particular, if $f(\tilde{x}) = f(x^m, x^l, x^u) = x^m$ the result will be the optimal clock period when all combinatorial delays are equal to the *most possible* value assigned to their vertices. The *imprecise* delay of a path is defined through the usual definition of the addition operation on triangular fuzzy numbers:

DEFINITION 6 *The imprecise delay of a path p in graph $\tilde{G}$ is:*

$$\tilde{d}(p) = \sum_{v_i \in p} \tilde{d}(v_i) = (\sum_{v_i \in p} d^m(v_i), \sum_{v_i \in p} d^l(v_i), \sum_{v_i \in p} d^u(v_i)). \tag{5.34}$$

To enable optimization we need a *max*-operator over triangular fuzzy numbers:

DEFINITION 7 *The maximum imprecise total propagation delay on any critical path from u to v is:*

$$\tilde{D}(u, v) = \tilde{\max_{p \in C}} \tilde{d}(p) = (\max_{p \in C} d^m, \max_{p \in C} d^l, \max_{p \in C} d^u). \tag{5.35}$$

where $\tilde{\max}$ is the fuzzy max operator, C is the set of paths between u and v with the lowest number total edge weight.

DEFINITION 8 *The imprecise clock period $\tilde{\phi}$ of the circuit modeled by $\tilde{G}$ is a triangular fuzzy number defined by the equation:*

$$\tilde{\phi} = \tilde{\phi}(\tilde{G}) = \tilde{\max_{p \in W^0}} \tilde{d}(p) = (\max_{p \in W^0} d^m, \max_{p \in W^0} d^l, \max_{p \in W^0} d^u), \tag{5.36}$$

where W^0 is the set of all paths that $w(p) = 0$.

LEMMA 1 *Applying a linear mapping f (as in def. 5) to the imprecise clock period of $\tilde{G}$ yields the same number as determining the clock period of the crisp graph of $\tilde{G}$ under f:*

$$\phi_f = \phi(G_f) = f(\tilde{\phi}). \tag{5.37}$$

PROOF. Consider all paths p in G_f for which $w(p) = 0$. Because the structure and the edge weights of crisp graphs are identical to those of $\tilde{G}$, this set is the same as in $\tilde{G}$. For any such path in G_f we have $d(p) = \sum_{v_i \in p} d(v_i) = \sum_{v_i \in p} f(\tilde{d}(v_i))$. Since f is linear and from definition 6 we get $\sum_{v_i \in p} f(\tilde{d}(v_i)) = f(\sum_{v_i \in p} \tilde{d}(v_i)) = f(\tilde{d}(p))$. Thus,

$$\phi_f = \max_{p \in W^0} d(p) = \max_{p \in W^0} f(\tilde{d}(p)) \qquad (\star).$$

For $\tilde{G}$ we have by definition 8 that $f(\tilde{\phi}) = f(\tilde{\max}_{p \in W^0} \tilde{d}(p))$. Again, the properties of f imply that $f(\tilde{\max}_{p \in W^0} \tilde{d}(p)) = \max_{p \in W^0} f(\tilde{d}(p))$. Combining this with $(\star)$ we obtain $\max f(\tilde{d}(p)) = \phi_f$. $\square$

THEOREM 2 *Given a crisp graph G_f obtained from $\tilde{G}$ by use of the mapping f (as defined in def. 5) the following holds:*

$$\min_{\phi_f \in K} \phi_f = \min_{\tilde{\phi} \in Y} f(\tilde{\phi}). \tag{5.38}$$

where K is the set of all feasible clock periods in G_f.

PROOF. From lemma 1 we know that applying the same legal retiming r to $\tilde{G}$ and G_f yields $f(\tilde{\phi}) = \phi_f$. From this follows that the retiming for which $\phi_f = \phi_{fmin}$ is also a retiming of $\tilde{G}$ with $f(\tilde{\phi}) = f(\tilde{\phi})_{min}$. $\square$

5.4.4 The FORTM algorithm

Theorem 2 is the basis of the following algorithm:

ALGORITHM FORTM (fuzzy optimization retiming):

1. Map the fuzzy graph $\tilde{G}$ into the crisp graph G_f using a mapping: $f(\tilde{x}) = a_1 x^l + a_2 x^m + a_3 x^u$ where a_1, a_2 and $a_3 \geq 0$.
2. Run the **LS** algorithm on the graph G_f.
3. Use the resulting configuration as optimal solution to the retiming problem of $\tilde{G}$.

THEOREM 3 *The algorithm* **FORTM** *solves exactly the program of equation 5.33.*

PROOF. We solve the program of equations 5.33 by mapping it into intermediate retiming problem, which we solve using an exact method. The mapping function is well defined in the sense of the definition 5 (see section 3.9 for discussion about how to select a_i). Therefore, from theorem 2 we get that the solution of the temporary problem represents exact solution to the original problem. $\square$

5.4.5 Computational complexity

The complexity of step 1 is $\mathcal{O}(|V|)$. In step 2 we call the standard retiming algorithm **LS**[1] of complexity $\mathcal{O}(|V||E|\lg|V|)$. This complexity remains unchanged because the graph mapping operations do not change the structure of the graph. Also the values of d_f remain well defined. Thus the total complexity is $\mathcal{O}(|V||E|\lg|V|)$.

[1]or $\mathcal{O}(|V|^3\lg|V|)$ if we use Bellman-Ford instead of FEAS

Chapter 6

SYNTHESIS RESULTS

Contents

6.1 Introduction

In chapters 4 and 5 two different approaches to the performance driven synthesis of digital systems were proposed. First one, based on possibilistic programming, substantially decreases the risk that a redesign of a system is necessary. The second one, called the *time budgets* method, removes this risk at all. On the other hand the possibilistic approach is wider applicable and more practical. In chapter 5 we described three successful applications of the possibilistic programming to problems of logic, high-level and hardware-software synthesis.

In this chapter we will review implementation details and experimental results for all these algorithms. We will start with the *time budgets* method. After that the details of the **PIP** solver, which we implemented as a separate program, will be presented. The implementation details and experimental results for the three possibilistic algorithms from chapter 5 will follow.

6.2 Time budgets example

6.2.1 Implementation details

We implemented the time-budgets method described in chapter 4 in a program called DYNAMITE. The data and control flow in DYNAMITE is shown in figure 6.1.

Continuous lines denote control flow, dotted lines data flow. The abbreviation **MCLKP** stands for Multiple Choice Linear Knapsack Program. The program consists of three stages:

1. We read the circuit from a file and make a graph model of it as described in section 4.2.1. Then, the set of paths in the graph is calculated, eventually applying a preprocessor (as described in section 4.2.2) to reduce their number. After that, we construct the constraints matrix A in the way described in section 4.2.2 and create the cost coefficients vector of the multiple choice linear knapsack program (as described in section 4.3.2).

2. The surrogate dual is solved to obtain a value of the aggregation vector w that maximizes the bound value. We use the sub-gradient algorithm as described in [Dye80] with some small modifications in the calculation of the step size. In each iteration a linear knapsack program is solved. Its solution vector is truncated or decremented if necessary. It is then compared with the best so far obtained state **BST**. Iterations are stopped as soon as the current bound value is higher or equal than **BSC** or no further improvement of it is observed.

3. *Branch and bound*. If the bound value is still lower than in the current best feasible solution the second stage is entered. As can be seen in figure 6.1, we use a priority queue to keep track of the subspaces. The priority of a subspace is calculated with the following formula:

$$z_i = v(\overline{\mathbf{PP_i}}) - v(\overline{\mathbf{P}}) + s_i \tag{6.1}$$

$$s_i = \sum_{j \in N} |\beta f_j| \tag{6.2}$$

$$f_j = \overline{x}_j - \widehat{x}_j \tag{6.3}$$

$$\beta = \frac{v(\widehat{\mathbf{P}}) - v(\overline{\mathbf{P}})}{s_P} \tag{6.4}$$

$\widehat{x}$ is a feasible integer solution obtained by truncating and decrementing some components of a solution $\overline{x}$ of **MCLKP**. Of course, subspaces with a lower value for z_i have higher priority.

Traditionally we create branches by dividing $F(\mathbf{PP_i})$ into two subspaces. As a branching variable either the non-integer variable from the knapsack problem solution, or variable which had to be decremented most is used.

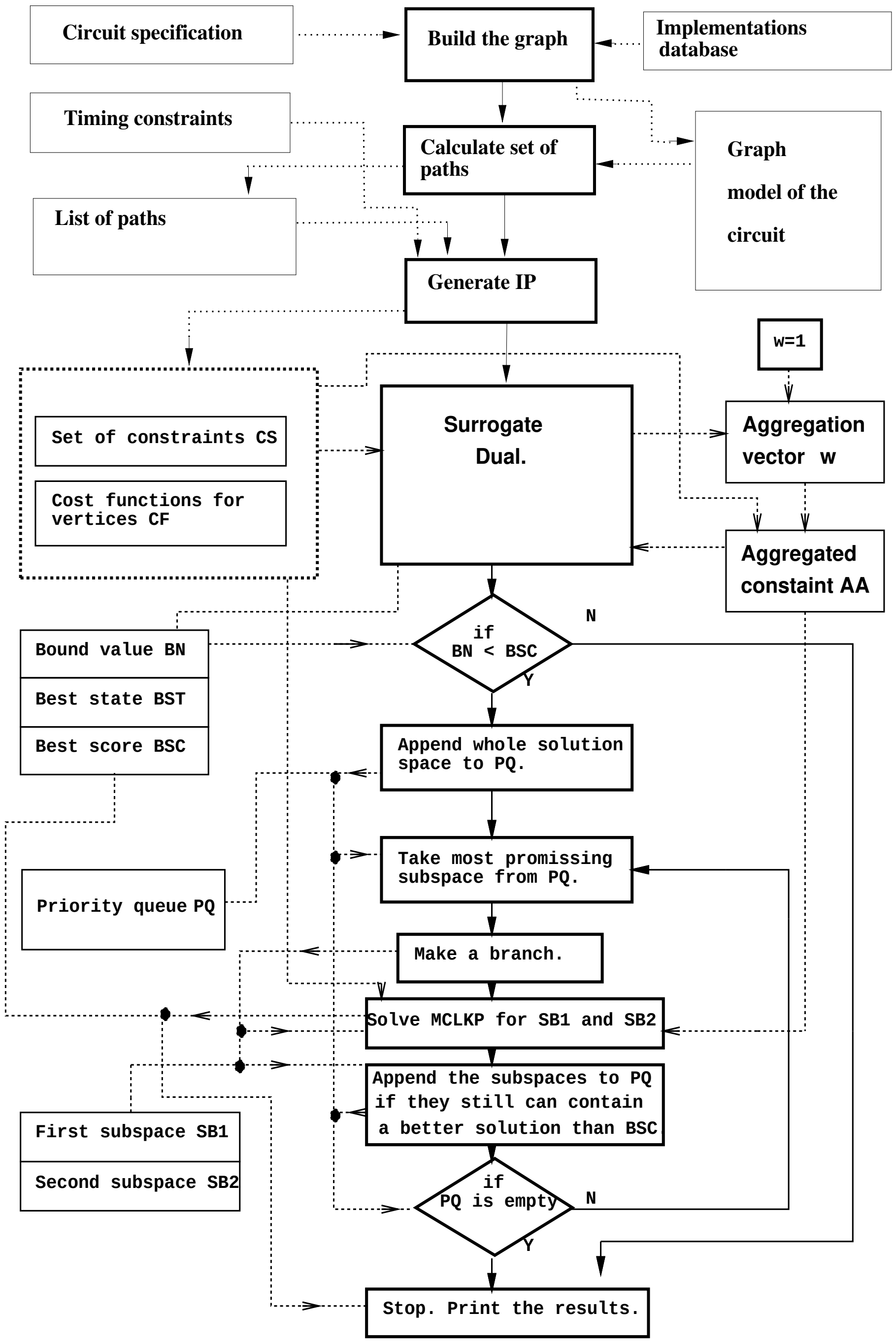


Figure 6.1. Data and control flow of the program.

If the bound value for $F(\overline{\mathbf{PP_i}})$ is smaller than the current best score **BSC**, $F(\overline{\mathbf{PP_i}})$ is appended to the priority queue **PQ**.

In each iteration the multiple-choice linear knapsack program is solved. First the knapsack problem is converted into the multiple-choice knapsack problem as described in section 4.3.2. Next we use the linear-time algorithm proposed in [Dye82] to solve it.

Another possible approach is to solve the surrogate dual during each *branch and bound* iteration assuming that the optimal value of vector w^* changes when the solution space is subdivided. We have tried this approach and the number of required iterations indeed decreased substantially, due to the larger improvement of the bound value obtained by each branch. However, we must keep in mind that solving the surrogate dual requires solving **MCLKP** several times. In our computational experiments we observed that this approach usually costs much more time than the version of program in which we solve the surrogate dual only once in the beginning.

6.2.2 Experimental results.

The algorithm has been implemented in C++ on HP9k/700 Unix machines and was tested on several sequential circuits. For each such circuit we changed the clock cycle time from maximal (the slowest realizations, that is maximal delay at each vertex) to minimal value (fastest realization, that is minimal delay at each vertex). The results in the table 6.1 and in figure 6.3 are obtained by using the example from figure 6.2. In the table, *n* denotes the number of vertices in the critical subgraph, *k* is the number of paths, b the number of binary variables.

All vertices have time cost functions with on the average 6 points.

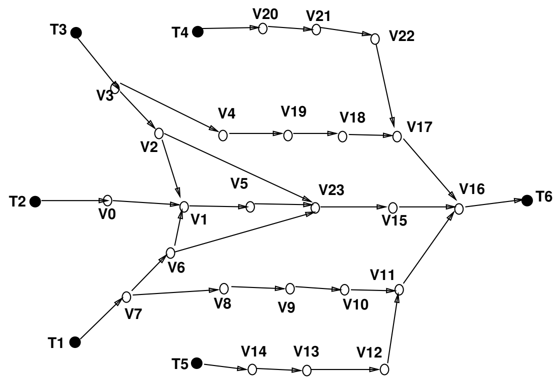


Figure 6.2. Example time constraints graph.

In four cases the execution times were above a minute. The optimal solution was however obtained after few iterations. It can be seen that the obtained bound values are usually close to the real optimal score and, consequently, only few *branch and bound* iterations are necessary. Solutions obtained after only solving the surrogate dual usually represent optimal vector x^*. In figure 6.3 both, the obtained scores after one iteration and the optimal scores, are plotted. Only in six cases was the one-iteration-result worse than the optimal score and then only by one unit.

For cycle times between 29 and 39 units, the number of binary variables for this example is 138, while the constraint matrix density is about 27%. This density is much bigger than for example problem P0201 from [CJP83] which is comparable in size (201 binary variables and density 5%). To solve it optimally they required about 1116 *branch and bound* iterations, which took about 9.9 CPU minutes on IBM 370/168.

6.3 The PIP solver

We implemented the possibilistic optimization methods described in chapter 3 as a general **PIP(PLP)** solver. The program is called FSOLVE. Its input is a comprehensive text format in which the cost function, constraints and definitions of integer/binary variables are written line by line. An example input file is shown in figure 6.4. All coefficients can be real or fuzzy numbers. We have built it on the top of very efficient optimizer called **CPLEX** ([CPL93]). Command line options control the values of the weight coefficients for z_1, z_2 and z_3 (eq. 3.31 in section 3.9). The user can also specify whether the normalization process described in chapter 3 should take place. If it is the case the solver generates 7 specially formulated crisp **IP**'s and calls **CPLEX** to solve them. We observed that most of the time is usually spend in solving z_2^{PIS} and the final "alpha" program (recall section 3.9). Since z_2^{PIS} is equivalent to solving normal non-fuzzy **IP**, to obtain the solution to **PIP** we have to spend in worst case 7 times more time. In practice it is however much less.

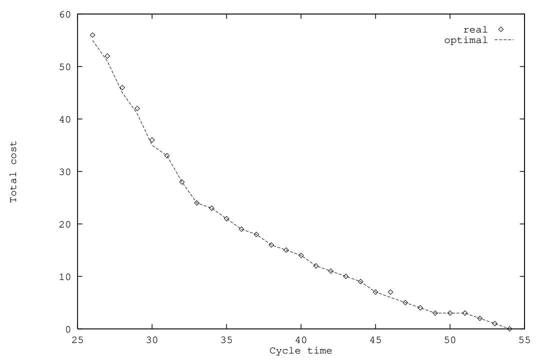


Figure 6.3. Scores obtained after single iteration together with optimal scores.

Cycle time	n	k	b	Optimal score	First bound	B&B iterations until opt. found	total	time [s]
53	11	2	69	1	1	0	0	0.01
52	16	3	103	2	1	1	2	0.04
51	16	3	103	3	2	1	2	0.42
50	17	4	109	3	3	0	0	0.04
49	17	4	109	3	3	0	0	0.02
48	17	4	109	4	4	0	0	0.04
47	17	4	109	5	5	0	0	0.05
46	17	4	109	6	6	2	3	0.63
45	17	4	109	7	7	0	0	0.15
44	21	6	132	9	8	1	113	1.92
43	21	6	132	10	9	1	134	2.32
42	21	6	132	11	11	4	5	0.78
41	21	6	132	12	12	0	0	0.19
40	21	6	132	14	13	1	280	4.12
39	22	8	138	15	14	1	33	1.25
38	22	8	138	16	16	0	0	0.23
37	22	9	138	18	17	1	168	3.44
36	22	9	138	19	19	0	0	0.34
35	22	9	138	21	20	1	61	1.91
34	22	9	138	23	22	1	136	3.18
33	22	9	138	24	24	0	0	0.27
32	22	9	138	28	27	1	407	6.79
31	22	9	138	33	31	1	2078	34.21
30	22	9	138	35	35	10	39	1.81
29	22	9	138	41	39	9	2481	41.92
28	22	9	138	45	44	1	1666	22.43
27	22	9	138	51	50	17	4414	61.87
26	22	9	138	55	54	15	572	6.73

Table 6.1. Optimization results

The possibility of defining weights for the three subgoals of the **MOLP** (eq. 3.30) turned out to be very useful. This way the user can decide what is for him more important - *most possible* cost, left or right spreads. As default we used the highest weight for z_2, then for z_3 and z_1. If the user decides to skip the normalization process then only a single crisp program is solved. Figure 6.5 contains example output generated by the program.

```
              /* The cost function  */
(5.5,5.1,6.2)F+(-1.0,-1.2,-0.85)M+(6.0,5.0,6.5)D
                                  -1.06B6 + 1.04L6;

              /* regular constraints  */
3F    + (2,1.5,3.2)M   + 2D - B1 + L1   <= 2;
1F    + 0.5M + 2D  + 1.06B1 - 1.04L1 - B2 + L2 <= 0.5;
1.8F  - 1.5M + 1.8D+ 1.06B2 - 1.04L2- B3 + L3 <= 0.4;
-0.4F - 1.5M - 1D  + 1.06B3  - 1.04L3- B4 + L4  <= 0.38;
-1.8F - 1.5M - 1D  + 1.06B4 - 1.04L4- B5 + L5   <= 0.36;
-1.8F - 0.2M - 1D  + 1.06B5 - 1.04L5 - B6 + L6 <= 0.34;

              /* Bounds  */
B1 <= 2;
B2 <= 2;
B3 <= 2;
B4 <= 2;
B5 <= 2;
B6 <= 2;
F <= 1;
M <= 1;
D <= 1;
              /* Integer variables   */
int F, M;
```

Figure 6.4. Example input file “Winston.lp” to FSOLVE

```
[moldau:fuzzy/LaiHwang 291] fsolve Winston.lp
Possibilistic Optimizer version 2.4
Solving alpha program...
Presolve Time =    0.00 sec.
B6     2.000000
M     1.000000
B1     2.000000
B2     2.000000
B3     2.000000
B4     2.000000
B5     2.000000
Done.
Value of objective = (3.12,2.97,3.32), (3.12,0.2,0.15)
```

Figure 6.5. Running FSOLVE on “Winston.lp”

6.4 Architectural synthesis examples

6.4.1 The FOAS system.

This section is devoted to implementation details and to results of our experiments with the architectural synthesis algorithms. Our system called FOAS (**F**uzzy **O**ptimization

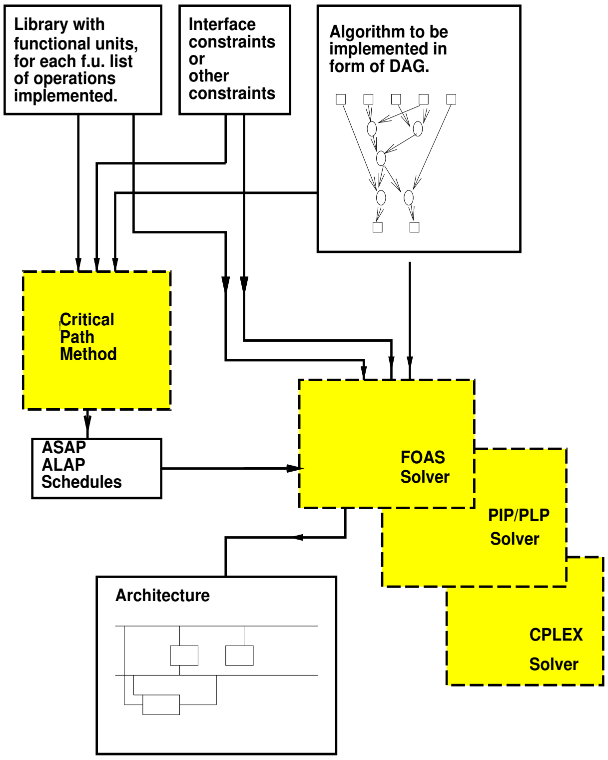


Figure 6.6. The FOAS System.

Architectural **S**ythesizer) combines methods of possibilistic programming and integer programming to efficiently solve problem defined in section 5.2.2. It has been programmed in *C++* on HP9k/735 Unix computer. Its block diagram is presented in figure 6.6.

The system reads its input data, specified in a user friendly format, from a file. This file contains descriptions of available functional units (including fuzzy cost, C and L parameters, and functions implemented), algorithm to be implemented in the form of a DAG, inputs/outputs, mapping from code operations into functional units (we allow units implementing many code operations), requested execution time and interface constraints. Based on this data, we first calculate $asap$ (as soon as possible) and $alap$ (as late as possible) schedules using the *critical path method* algorithm of complexity $\mathcal{O}(n^2)$. These schedules are later very useful because they allow us to restrict the size of the solution space (only $csteps$ between $asap(k)$ and $alap(k)$ need to be considered). Next, we generate the cost function and constraints of the **PIP**. It is then piped into the **PIP/PLP** solver. The result of this call is a 0-1 solution to our problem. We need only to scan these variables which have been assigned value of 1 (means that a code operation k, has been assigned to cstep j and to functional unit i). After sorting these variables, the system displays the resulting schedule.

Unit	C	L	Size (fuzzy)	Functions
add1	1	1	(12,8,15)	+
add2	2	1	(8,7,12)	+
mul1	1	1	(15,11,19)	*
mul2	2	1	(11,10,15)	*
alu1	1	1	$(25,21,29^A/39^B)$	+,*
alu2	2	1	$(19,18,23^A/33^B)$	+,*

Table 6.2. Functional units specifications (dod2mn ex.)

6.4.2 "Dod2mn" example

The algorithm that has to be synthesized is shown in figure 6.7. The available functional units are listed in table 6.2. In the table, C is the execution time, L - latency time (minimum time between successive data input values being accepted by the unit), both in clock cycles.

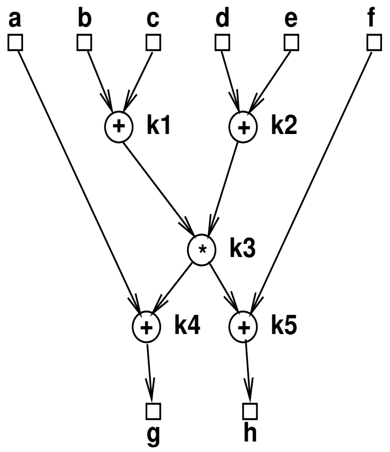


Figure 6.7. Algorithm to be synthesized (dod2mn ex.).

We compute the optimal schedule and functional unit assignment using FOAS system. We specify timing constraints on the required output time to be 4 csteps. Our program produced schedule **A** (table6.3). Notation $unitname^i(codename)$ means that *i-th* unit of type $unitname$ is used to implement code operation $codename$ at given cstep. The cost function value for this solution is equal $(37, 29, 44)$ and can be seen in figure 6.8. We conclude that the *most possible* size of this architecture is 37, it should never be lower than 29 or higher that 44 (left spread of fuzzy cost is equal 8, right 7).

Now we make a small modification. We change the characteristics of available alu's in such a way that x^o for them is much higher. It means that they could be

cstep	assignment	
1	alu1^1(k1)	add1^1(k2)
2	alu1^1(k3)	
3	alu1^1(k4)	
4	alu1^1(k5)	

Table 6.3. Schedule A.

cstep	assignment	
1	add1^1(k1)	add1^2(k2)
2	mul2^1(k2)	
3		
4	add1^1(k4)	add1^2(k5)

Table 6.4. Schedule B.

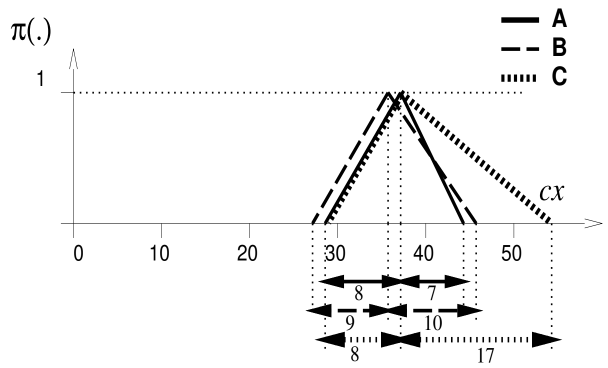


Figure 6.8. Optimal solutions (dod2mn ex.).

much bigger than the *most possible* size (see value with index B in the table 6.2). Now we run our system again without changing the other parameters. The resulting schedule is presented in table 6.4. The program detected the potential "danger" of using alu's and decided for a solution with two adders and one multiplier (cost function value $(35, 26, 45)$, see fuzzy number **B** in figure 6.8). Comparing this solution to solution **A** we can see that the *most possible* size of the architecture **B** is 35, which is better by 2 units compared with architecture **A**. The right spread, representing the risk of obtaining worse solution, is higher than in architecture **A**. This higher risk was the reason why this solution has not been chosen before. In figure 6.8 also another, 3rd fuzzy number can be seen (with name **C**). It represents that value of the cost function which we would get if we kept the architecture **A** despite the fact that characteristics of our alu's have changed. Such solution would have the same value of the *most possible* size but in the worst case it could be even 17 units bigger. Solution **B** is in that respect much better than **C**.

Concluding this very simple example, it is interesting to point out a very important characteristic of our methodology. There may exist many solutions with a similar *most possible* value of the cost function (very close to the global optimum). The system was in both cases able to choose such an architecture which had very small *most possible* size but simultaneously, had a small risk of running out of space during layout design and a large *possibility* of obtaining even smaller sizes.

6.4.3 EWF example

We also ran our program on the obligatory Elliptical Wave Filter example [Dep88]. The available functional units are listed in table 6.5.

Unit	C	L	Size (fuzzy)	Functions
add	1	1	(12,8,15)	+
mul1	1	1	(15,11,19)	*
mul2	2	1	(11,7,20)	*

Table 6.5. Functional units specifications (EWF).

Figure 6.9 shows the schedule obtained using only x^m values (traditional way). In figure 6.10 the schedule generated using fuzzy costs is shown. We optimize for area and for latency $T_e = 19$. As can be seen, both solutions use 2 adders and 1 multiplier. The first schedule uses the pipelined multiplier $mul2$ while the second the single cycle $mul1$. The fuzzy cost function values are $(35, 23, 50)$ and $(39, 27, 49)$. The most possible cost in the second solution is a little bit higher. In the same time this solution is less "risky" since the right spread is by 5 units smaller.

6.5 Retiming example

In this section we present a simple numerical example. It illustrates with how the obtained in section 5.4 results can be used to retime circuits with imprecise delays.
Recall the circuit "dwareg" from chapter 5.4. Its graph model is presented again in figure 6.11. The delay of every combinatorial block of the circuit is a triangular fuzzy number (also shown in figure 6.11). We use the following coefficients: $\beta_1 = 1$, $\beta_2 = 3.5$,$\beta_3 = 2$. The mapping function for the graph G_f thus becomes:

$$f(\tilde{x}) = f(x^m, x^p, x^o) = x^p + 0.5x^m + 2x^o.$$

The minimum clock-period retiming of the graph G_f is shown in figure 6.12. This is the same configuration as in the initial circuit. Referring to our discussion of example 13 in section 5.4.1 this is exactly the solution that we then had preferred. The reader can easily check that if we had $d(c) = (4, 3, 5)$ (the risk that the delay of the unit "c" is larger would be smaller) then we would get $d_f(c) = 15$ in G_f (see figure 6.13). The same retiming results as produced by the **LS** algorithm with exact delays equal to x^m.

In figure 6.14 two triangular fuzzy numbers are depicted. The fuzzy number **A** drawn with full lines is the imprecise clock-period of the configuration that we

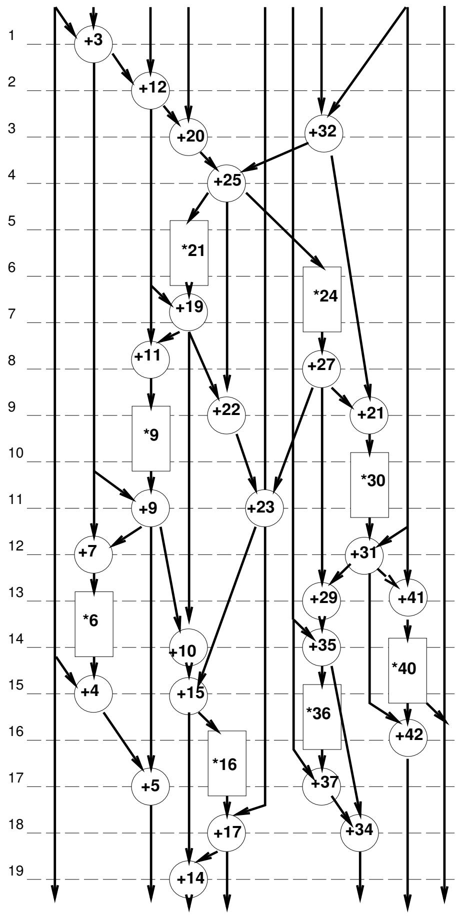


Figure 6.9. EWF schedule optimized using *most possible* costs only.

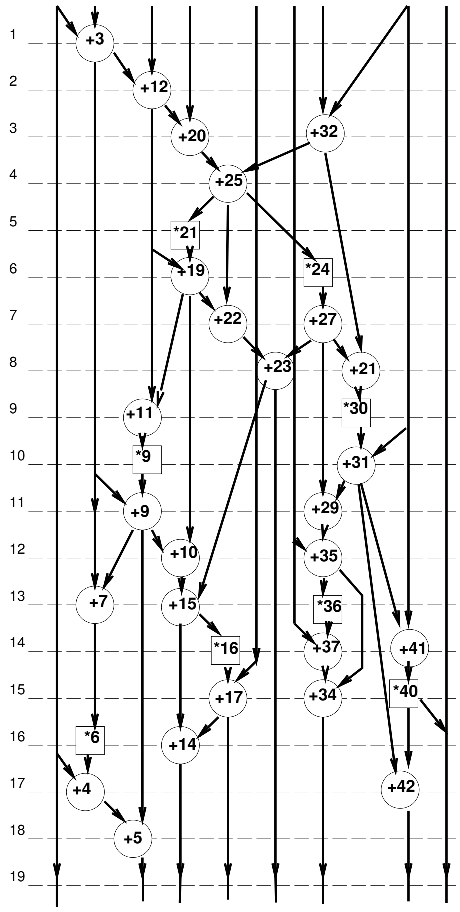


Figure 6.10. EWF schedule optimized using fuzzy costs.

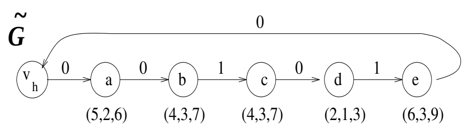


Figure 6.11. A graph model of the "dwareg" circuit

obtained using the **FORTM** algorithm. The fuzzy number **B** represents the possibility distribution of the clock cycle time in the configuration generated by the

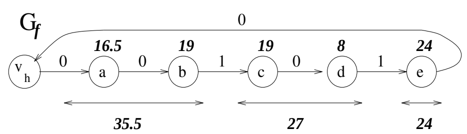


Figure 6.12. G_f after retiming if $d(c) = (4, 3, 7)$.

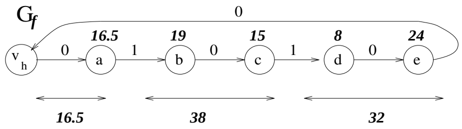


Figure 6.13. G_f after retiming if $d(c) = (4, 3, 5)$.

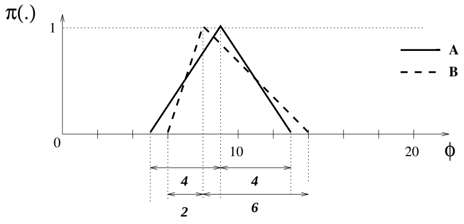


Figure 6.14. Two solutions to the retiming problem of the "dwareg" circuit.

LS algorithm. Comparing these two solutions we can see that the most possible clock-period in **A** is larger than in **B**. **B** does have a much larger right spread however, which "measures" the risk of obtaining a worse solution. Also it has much smaller left spread, which represents the chance of obtaining a better solution. We conclude that solution **A** is "better" in the sense of the strategy defined by program 5.33. There may still be design situations in which the user wants faster circuits and he is prepared to take more risk. In such case he can influence the tradeoff between the three subgoals of eq.5.33 by appropriately setting the weight coefficients β_i.

6.6 Hardware/software partitioning example

In this section we will concentrate on a third application of possibilistic programming to the problems of the synthesis of digital systems - the hardware/software partitioning of embedded systems (as described in section 5.3). The partitioner program is called HSPART and has been implemented in C++ on HP9k/735 Unix computer. We built it on the top of our general possibilistic solver called FSOLVE (see section 6.3). It forms the central part of our hardware/software co-design environment presented in figure 6.15. Let us shortly describe the function of all its components. The algorithm specification in C (augmented to allow parallel flows of execution) is parsed by the **c2sir** tool (part of the CASTLE system [TSV94]) and written into the SIR database. Then, it is converted using the **lv** program to another (more convenient for us) format, called *LV*. Data and control flow graphs from that file are parsed by a filter program called **sirFilter**. The program removes all dummy or unnecessary (not partitioned) nodes and edges. We use the call hierarchy of the program to determine the partitioning boundaries (figure 6.16). The partitioning stops at the first level. This requires the program to be written with this

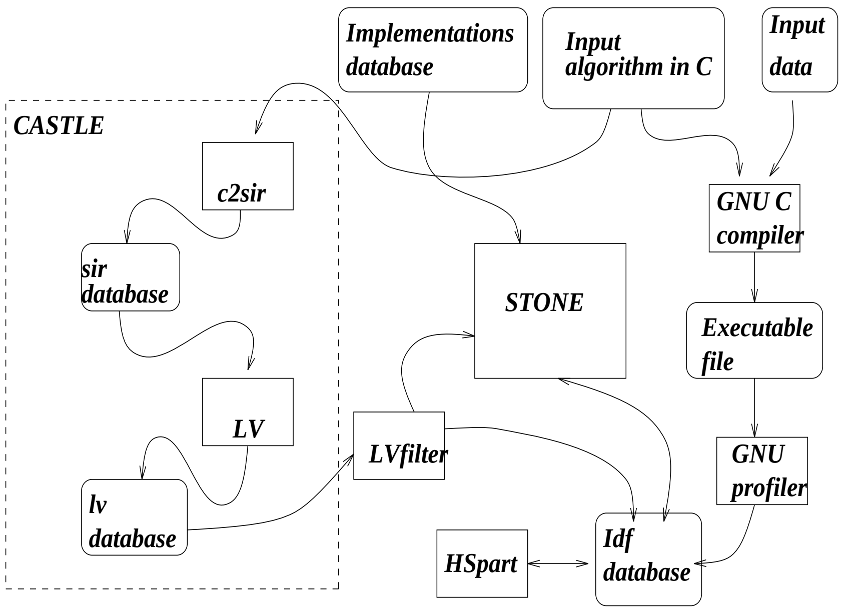


Figure 6.15. The hardware software co-synthesis environment

strategy kept in mind. If the algorithm contains parallel flows of execution, appro-

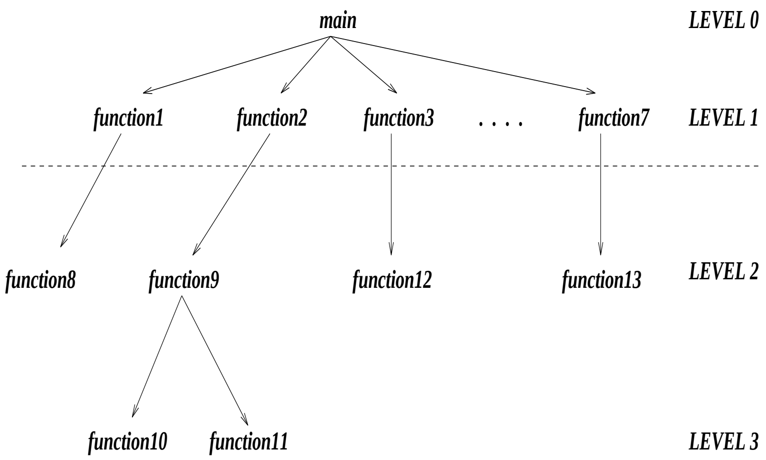


Figure 6.16. Example call hierarchy

priate paths in the control flow graph are split and put in parallel. The result is read by the STONE program [Cap95], which serves as a graph viewer and general user interface. The program uses the so called *Idf* data format to store all information. Simultaneously the input code is compiled with the GNU C compiler and profiled using provided input data. The result is written into the *Idf* database. Now there are two possibilities:

1. the user defines all possible implementations for all basic blocks and data transfers, using dialog boxes within STONE (figure 6.17),
2. the information about the implementations is imported from other projects (implementations database in figure 6.15).

After that, the user can define design constraints, types of available processors etc. Also locking of some implementations is allowed. All the information is directly written into the *Idf* database. Now the partitioner (HSPART) is called taking input from the *Idf* file. The resulting partition is written back to the database and is displayed in STONE. The user can now evaluate the obtained solution, modify it and eventually run the partitioning again, until satisfactory solution is reached.

To test the program we run it on several examples. One of them is the WISSCE receiver [Gla95]. Figure 6.18 presents the code used as the system's input. As can be seen we run the program 3 times (**dummyCnt** variable), using $N = 260$. This gives 780 executions of every node and edge within the **for** loop. Data about hardware implementations was provided by experts. The advantages of using possibility distribution instead of just a single most expected value became obvious almost immediately. It was difficult to predict exactly even the number of clock cycles that certain function will take on the CPU. Predicting the total number of transistors eventually used by the hardware is known to be a difficult problem. Allowing for fuzzy numbers has been welcomed with enthusiasm. At the same time another advantage of our approach has been confirmed - there was simply no way of generating good probability distributions other than generating every potential candidate vertex several times - a designer's nightmare.

We use the HP PARISC1.1 general purpose processor as the CPU of the receiver. To obtain data about the latencies (number of clock cycles) of the software versions of the basic blocks we compiled the code with the GNU C compiler with debug and optimization options on. Next we investigated the generated code with the gdb debugger. The number of cycles that the argument's passing takes was used as the data transfer latency. The call instruction, saving/restoring registers, allocating/deallocating local frame and the body of the callee accounted for the latency of the basic blocks. Latency of all other instructions (initialization of global variables and loops' control) constituted the latency of the *Supervisor* vertex. Figures 6.19 and 6.20 show software latencies of all non-variable basic blocks in the program. The running times were rather short. For instance for the *receiver* example they were well below 1 second. We optimized for the minimum area of the extra hardware. Three timing constrains were used:

1. *latency*: this constraint limits the latency of one *run of the system* (total time spent by the CPU in one *run of the system*).
2. *path1*: limits the total time that basic blocks **readSample, shiftPNgen, getPrompt, IQmpy, filter, limit** and associated with them data transfers take.

3. *path2*: limits the total time that basic blocks **readSample, shiftPNgen,getEarly, IQmpy, limit** and associated with them data transfers take.

We experimented with different time values in these constraints. The partitioning results can be seen in figures 6.21, 6.22 and 6.22. The basic blocks and data transfers that have been assigned to hardware are darker.

Let us shortly try to analyze the obtained results:

1. Figure 6.21: We set the timing constraints to be $1ms$. As can be seen in figure 6.20, the software version of the **filter** operation takes almost 2/3 of the total time. To satisfy the timing constraints it has been moved to hardware. All other operations could remain in software.

2. Figure 6.22: The timing constraints are set to be $600\mu s$. Now neither path satisfies the constraints and therefore something else has to be moved to hardware. **IQmpy** and **limit** are better candidates than **getPrompt** and **getEarly** because their basic blocks (belonging to both paths) can use the same unit. The gain in performance obtained by moving **IQmpy** or **limit** is sufficient to satisfy the constraints. The sizes of hardware for the operations are $(51, 45, 60)$ for **IQmpy** and $(50, 46, 65)$ for **limit**. As can be seen the most possible values are almost the same, but the hardware implementation of **limit** is slightly more "risky". Therefore the solution with **IQmpy** in hardware has been chosen. Again the important feature of the possibilistic approach is clear. If there exist solutions with a similar *most possible* value of the cost function, the solution with lower "risk" of obtaining a "worse" solution and higher chance of getting a "better" solution is selected.

3. Figure 6.23: The timing constraints are $600\mu s$. In addition we lock the implementation of the **gen** basic block to be in hardware. The Kasami-code generator (represented by this variable) is large. Only transferring it, without actually doing any other computation could violate the constraints. There are 4 data transfer edges from **gen**. Two of them require access to whole **gen** object, two other use only a small part of it (basic blocks **initGen** and **shiftPNgen**). Therefore to avoid spending too much time in data transfers the partitioner selected hardware implementations also for basic blocks **getPrompt** and **getEarly**.

As can be seen the partitioner generates very logical results. Also the imprecise nature of the system parameters is taken into account.

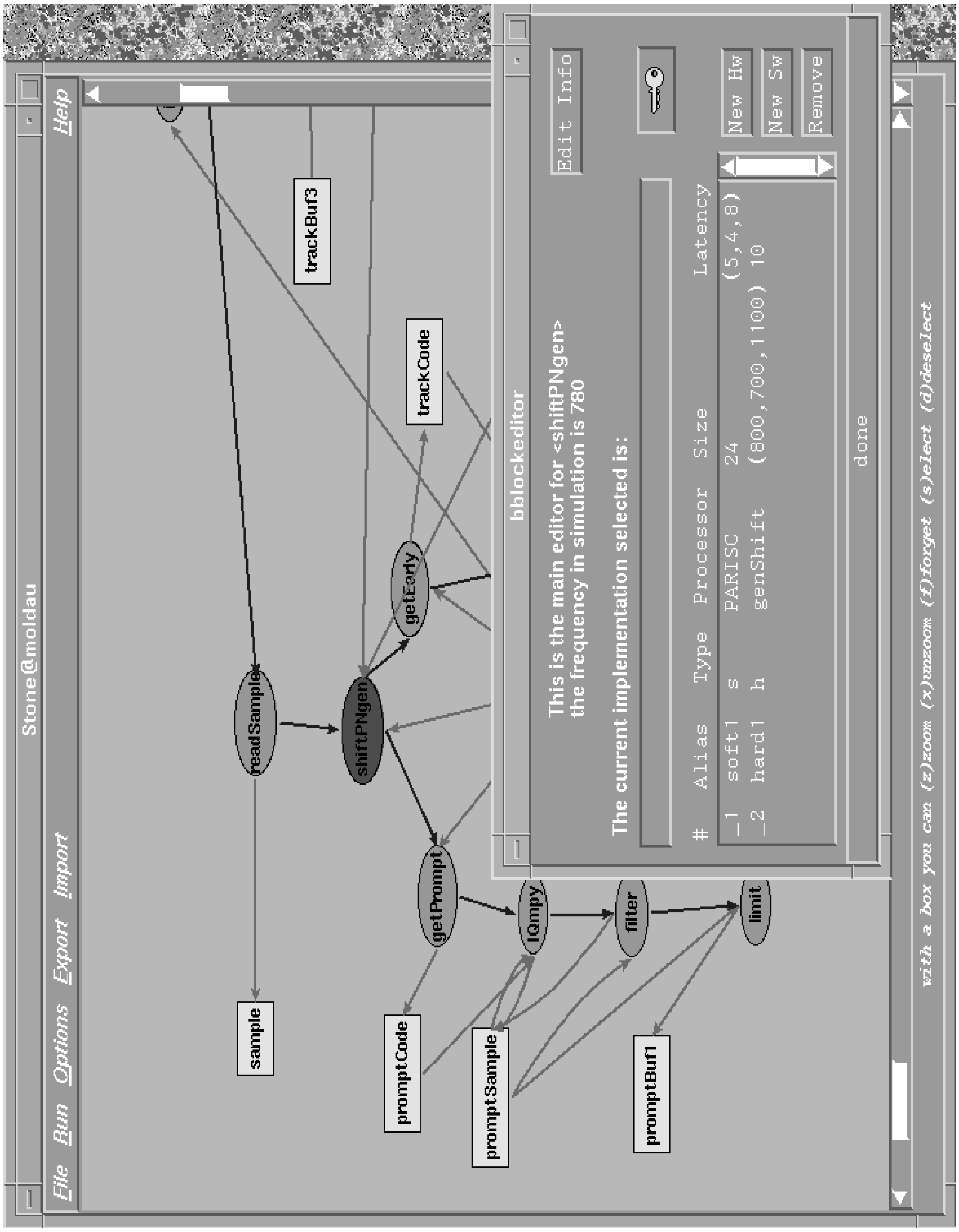


Figure 6.17. Defining implementations in Stone

```
twiddle tw;                  /* twiddle factors (for fourier) */
kasgen gen;                  /* kasami-code generator */
int dummyCnt=3;              /* we run it 3 times */
                             /* initialization of PNgen */
initGen(&gen, (short) 63, (short) 7, (short) N);
                             /* initialization of twiddle-factors */
initTwiddle(&tw);
while (dummyCnt--) {
    IQbit promptBuf1[N];     /* prompt-buffer 1 */
    IQbit trackBuf1[N];      /* track-buffer 1 */
    register short i;        /* count-vars: */
                             /* do loop (N=260)*/
    for(i=0; i<N; i++) {
        register IQ sample;
                             /* read input from ad-converter */
        sample = readSample();
                             /* shift pn-code generator */
        shiftPNgen(&gen, trackCntr, acq);
        <
          {
            register byte promptCode;
            register IQ promptSample=sample;
            register short pPointer=i;
                             /* get prompt-code */
            promptCode = getPrompt(&gen);
                             /* do prompt-despreading */
            promptSample = IQmpy(promptSample, promptCode);
                             /* do filtering */
            promptSample = filter(promptSample);
                             /* do limiting and put in buffer */
            promptBuf1[pPointer] = limit(promptSample);
          }
          {
            register byte trackCode;
            register IQ trackSample=sample;
            register short tPointer=i;
                             /* get early-late code */
            trackCode = getEarly(&gen) - getLate(&gen);
                             /* do early-late despreading */
            trackSample = IQmpy(trackSample, trackCode);
                             /* do limiting and put in buffer */
            trackBuf1[tPointer] = limit(trackSample);
          }
       >
   } /* end for */
}  /* end while */
```

Figure 6.18. The C specification of a part of the WISSCE receiver

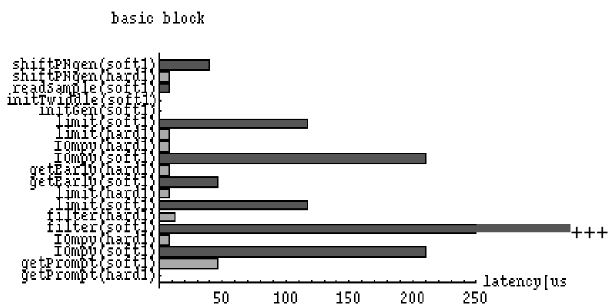


Figure 6.19. Total latencies of software and hardware implementations of basic blocks

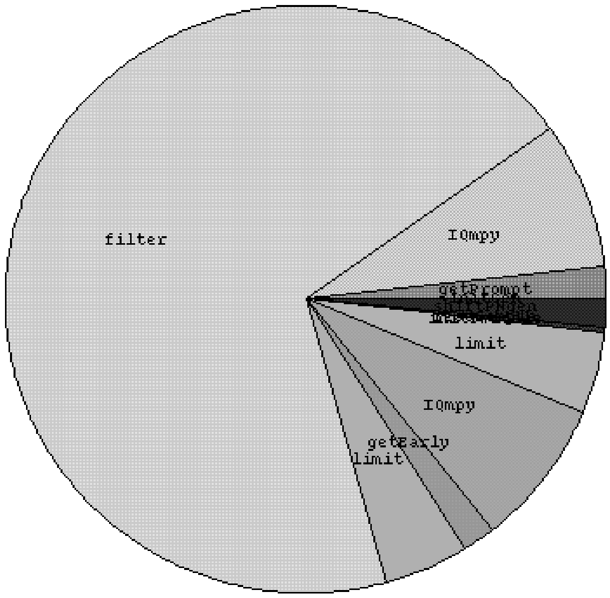


Figure 6.20. Total software latencies of all basic blocks

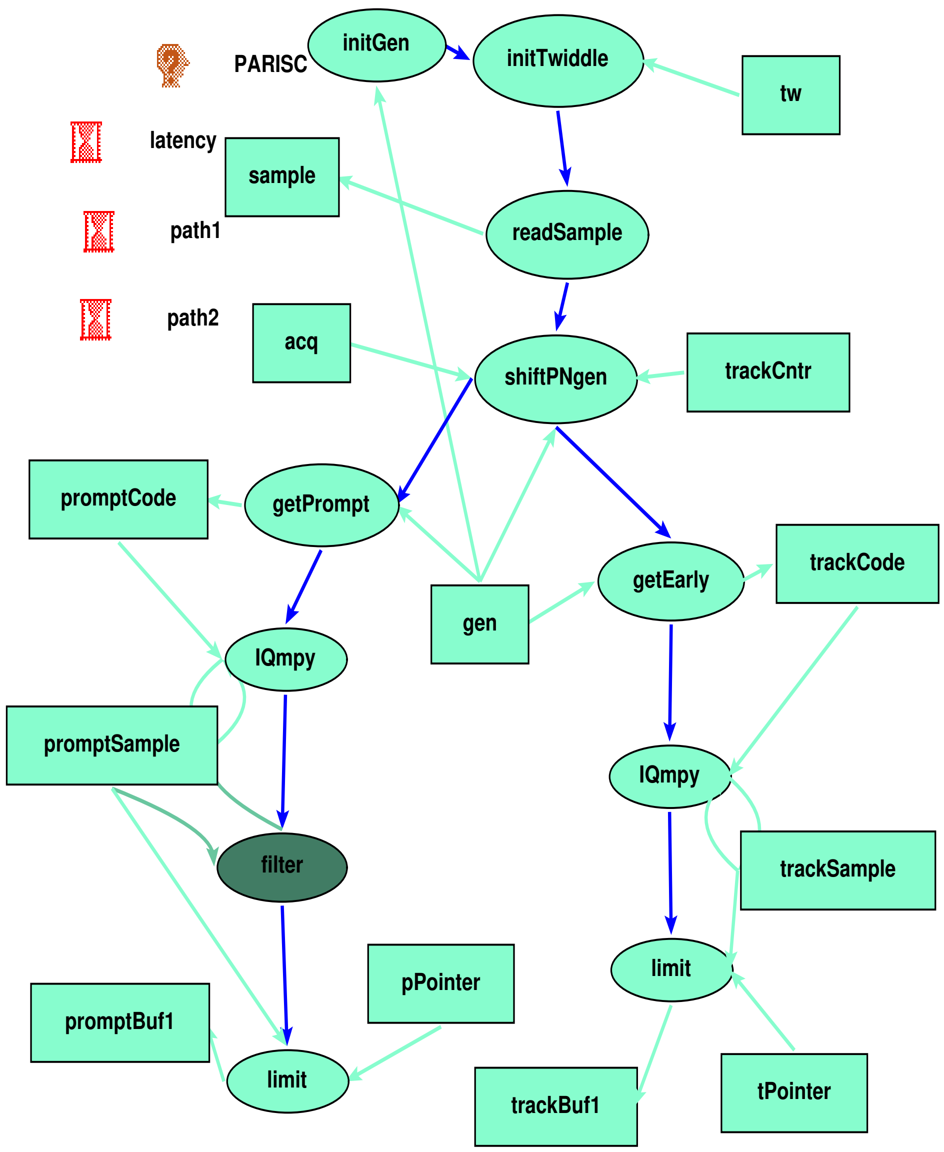


Figure 6.21. Partitioning of the WISSCE receiver for $L \leq 1ms$.

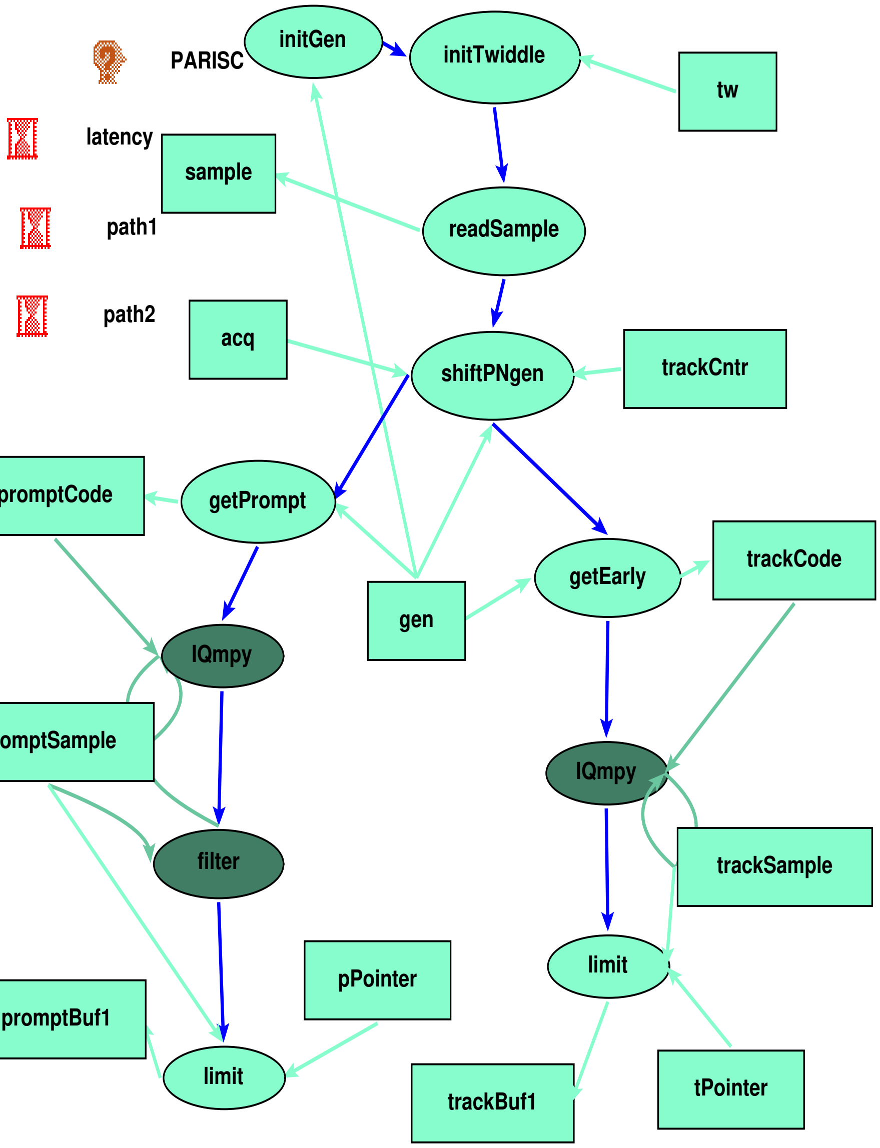


Figure 6.22. Partitioning of the WISSCE receiver for $L \leq 600us$.

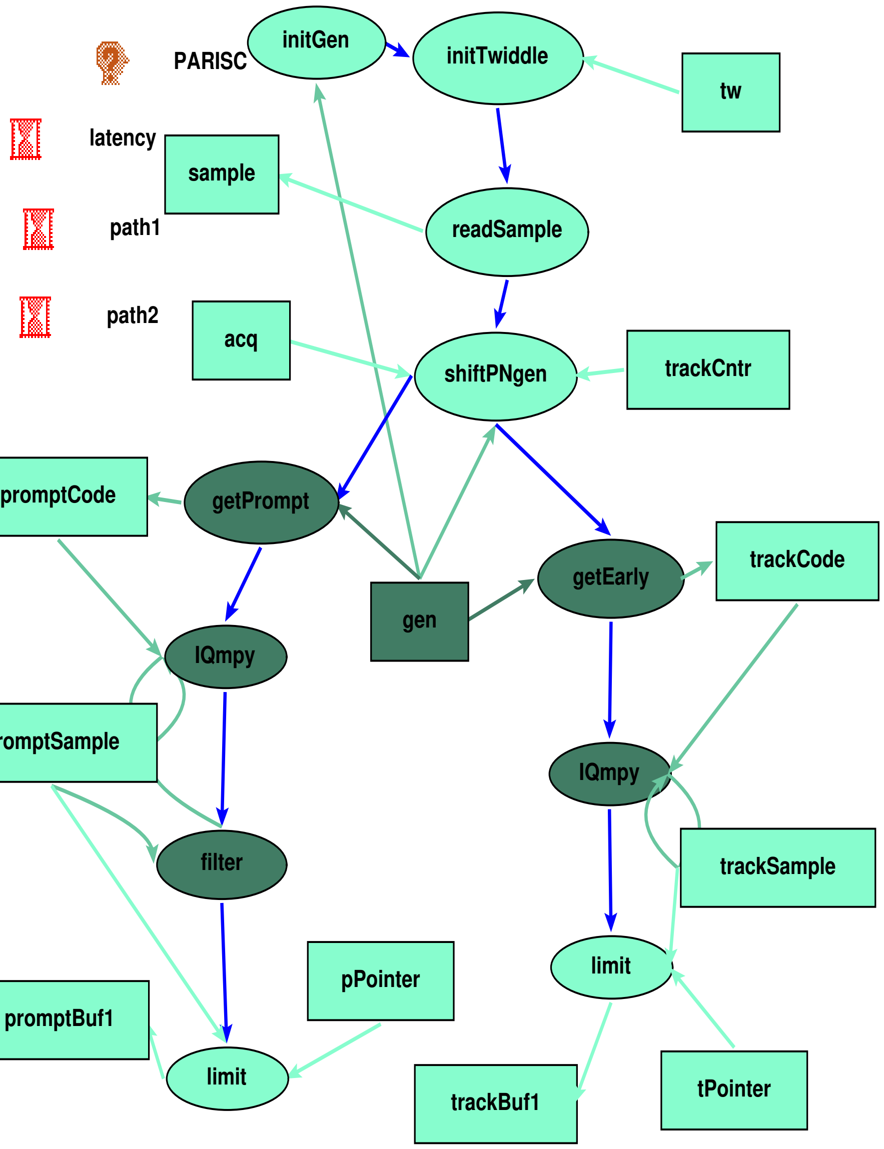


Figure 6.23. Partitioning of the WISSCE receiver for $L \leq 600us$, (implementation of **gen** locked to hardware)

Chapter 7

CONCLUSIONS

7.1 Performance driven synthesis of digital systems

In this thesis we have proposed a new approach to the performance driven synthesis of digital systems. To put our results in a proper perspective we first gave an overview of existing synthesis systems. After that, the time delay models were introduced, followed by a *design trade-offs* case study. We concentrated also on a, from our point of view, very important issue: the imprecision of the delay models. Important, since it is one of the motivations for our methodology. Most of the methods described in this thesis were based on *integer programming* and possibilistic programming. Therefore the **IP** and possibilistic methodology have been reviewed.

Two different synthesis approaches were proposed. The first one called the *time-budgets* method belongs to the group of so called "knowledge database based" approaches (group 3 in preface). The method removes the need for redesigns. The second, wider applicable and more practical, method has been shown to be effective in significantly decreasing the risk that a redesign is necessary. This goal was achieved using *possibilistic programming* [Zad78], a branch of fuzzy mathematical programming. Possibilistic programming has been shown to provide good means to model performance imprecision. The possibilistic methods were proven to be flexible, making the process of adopting the existing synthesis algorithms easy, as demonstrated in three successful applications of the possibilistic approach. They belong to different stages in the automated synthesis: the problem of *simultaneous scheduling, selection and allocation of functional units*, the problem of *hardware-software partitioning of embedded systems* and the problem of *retiming* of synchronous circuitry with imprecise delays. Another reason for selecting these problems was that they belong to different categories of problems from the point of view of applying possibilistic programming. The *scheduling* problem was an example of a problem with a solution known from literature. It could be efficiently solved with a general **IP** solver. Possibilistic programming could therefore be directly applied. For the hardware/software partitioning problem we presented a totally new solution. As far as we know, the problem has not been addressed in this way before (neither for the precise nor the imprecise case). We have shown that also that problem could be solved using a combination of a general **IP** solver and possibilistic methodology. The *retiming* problem belongs to the class of problems for which a dedicated efficient algorithm exists. We have shown that also for this class the possibilistic methods can be successfully applied. The implementation details and experimental results for all proposed algorithms were presented.

We certainly hope that this thesis will stimulate other researchers to incorporate the possibilistic methodology in their design automation tools. As has been argued, there are several good reasons for doing that. Even more, it is difficult to come out with counterarguments. It will be always possible to use the extended software with precise data (if it is available), providing the same functionality and efficiency as the classical (non-possibilistic) versions. If however at least a part of the design information is imprecise (with possibility distribution given) the extended software will be, with the same efficiency, able to take advantage of it.

7.2 Contributions

In this final section we briefly present the list of the contributions, restricted to the domain of Electronic Design Automation. We can divide them into two groups:

- Programs being part of the OCEAN sea-of-gates design system: the detailed placement part of the MADONNA placement program (working with arbitrary images and being able to take advantage of mirror axes), the SCHERZO schematic entry editor (featuring automatic interconnection (re)drawing), the KISSIS and blif2sls programs interfacing OCEAN with SIS, and the CRUISE user interface program. The OCEAN system can be retrieved via FTP from host `morra.et.tudelft.nl`, directory pub/ocean.
- Optimization programs: the general **PIP/PLP** solver called FSOLVE, the DYNAMITE timing optimization program (featuring time-budgets method of chapter 4 and timing optimization with simulated annealing), the FOAS architectural synthesis system (featuring scheduling algorithm of section 5.2), the FRTM retiming program (implementing FORTM algorithm of section 5.4) and the HSPART hardware/software partitioner (as described in section 5.3 and 6.6). All these programs are available via WWW. To retrieve them open URL `http://olt.et.tudelft.nl/~irek` and follow instructions provided there.

Figure 7.1 shows location of all above mentioned programs in the overall design flow. Please direct all questions about this thesis or the programs to `irek∂muresh.et.tudelft.nl`.

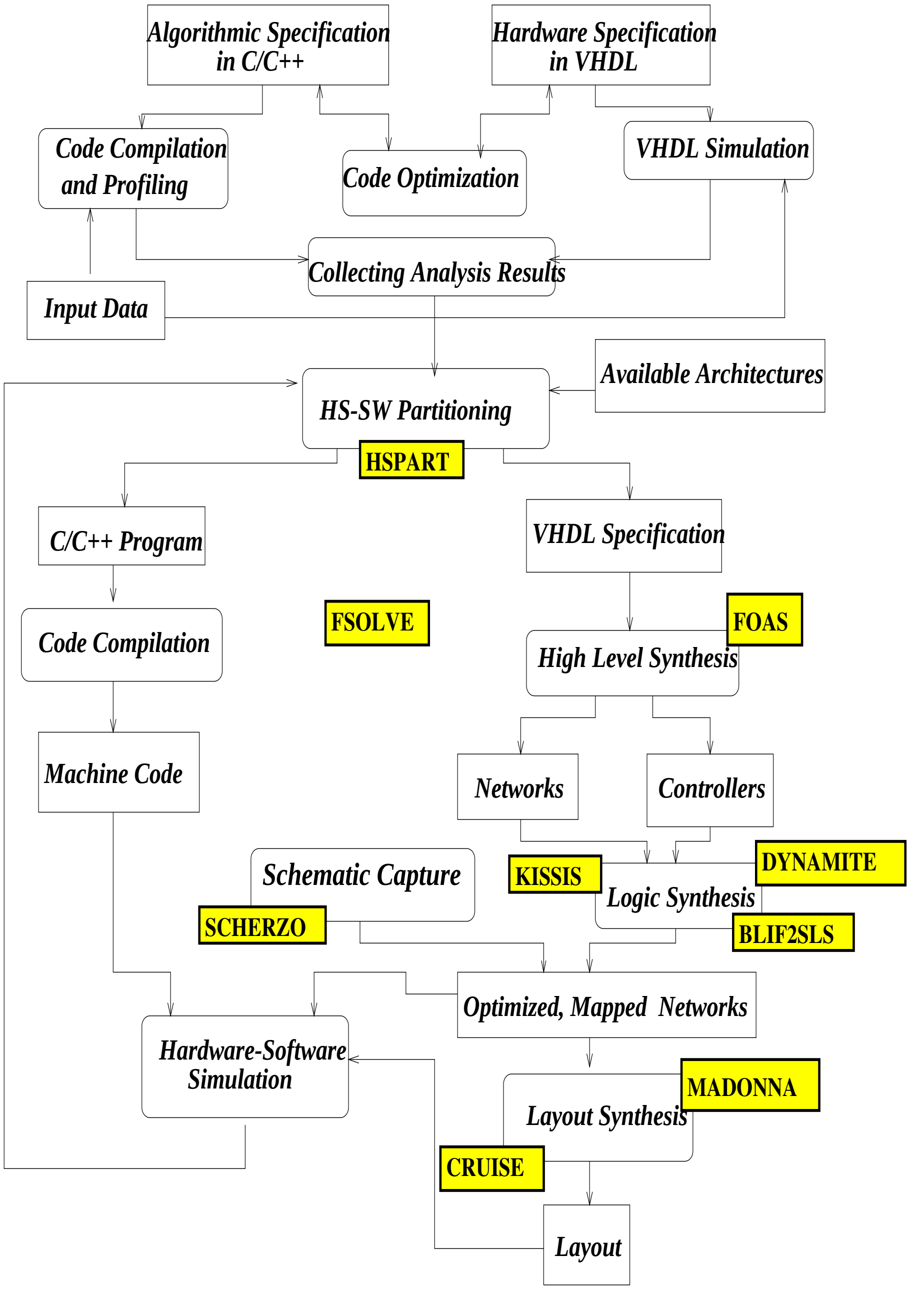


Figure 7.1. Location of the contributed programs in the design flow

Appendix A

FUZZY SETS AND POSSIBILITY THEORY

Contents

As the title says in this appendix we will review some fundamentals of fuzzy sets and possibility theory. We do not intent to give here a complete course on these topics. Only notions, which are important in understanding the methods used in this thesis have been selected. We will start with the definition of fuzzy sets, their properties and operations on them. After that possibility distributions will be formally defined. Finally we will compare different interpretations of fuzzy sets.

A.1 Fuzzy sets versus ”crisp” sets

Let X be the universe of discourse. The classical sets (also referred as “crisp” sets) have the property that any element $x \in X$ is a member of a set A or not. Formally this can be stated using the notion of *membership*. The *membership function* of the classical set is defined as:

$$\mu_A(x) = \begin{cases} 1 & \iff & x \in A, \\ 0 & \iff & x \notin A. \end{cases} \tag{A.1}$$

In other words an element $x \in X$ is a member of a “crisp” set A if $\mu_A(x) = 1$ and is not if $\mu_A(x) = 0$. For classical sets we also have that for any $x \in X$, $x \in A \vee x \notin A$.

Example 14 *X be a set of all natural numbers, and A a set of all $x \in X$ which are multiples of 3. The crisp set A can then be represented as:*

$$A = \{(1,0),(2,0),(3,1),(4,0),(5,0),(6,1),\ldots\}$$

where the second number in every pair is the measure of membership for a given x (first number in pair).

In practice there are many complex systems which cannot be modelled by use of precise mathematics. In many situations it is not really clear whether x belongs to A or not. In other words some sets may not have sharp boundaries. A solution to the problem is to use membership grades in the interval $[0,1]$.

Example 15 *Consider the following statement:* the car is fast. *Imagine that we have five cars with the following maximum speeds:*

Car model	*maximum speed [km/h]*
Trabant GTI	*80*
Ford Escort	*155*
Opel Astra	*165*
VW Golf	*150*
Porsche S	*250*

We could define our membership function to be:

$$\mu_A = \begin{cases} 1 & \iff speed(x) \geq 160, \\ 0 & \iff speed(x) < 160. \end{cases}$$

This would result in the following set:

$$A = \{(Trabant\_GTI, 0), (Ford\_Escort, 0), (Opel\_Astra, 1),$$

$$(VW\_Golf, 0), (Porsche\_S, 1)\}$$

There is no doubt that $Trabant\_GTI$ is slow, and the $Porsche\_S$ fast. The other 3 cars have however quite comparable maximum speed. A model in which $Opel\_Astra$ is considered fast and cars $Ford\_Escort, VW\_Golf$ not, does not really reflect the real meaning of the statement the car is fast. *Especially because it would mean that $Ford\_Escort$ and VW_Golf are slow, what is obviously not true. Clearly the set of fast cars does not have a sharp boundary. This can be modelled by another membership function, where the elements on the fuzzy boundary have membership grades in the real interval $(0,1)$. In figure A.1 a mapping from speed to membership grades in fuzzy set "fast" is presented. Using this membership function we obtain the following fuzzy set:*

$$A = \{(Trabant\_GTI, 0), (Ford\_Escort, 0.55), (Opel\_Astra, 0.8),$$

$$(VW\_Golf, 0.5), (Porsche\_S, 1)\}$$

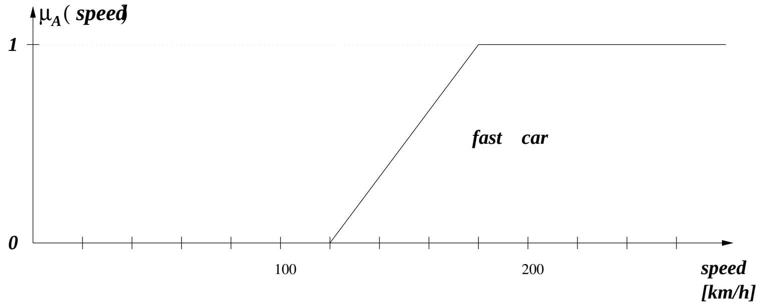


Figure A.1. Fuzzy set "fast"

A.2 Terminology and definitions

A fuzzy set is a set with the evaluation set allowed to be real interval $[0, 1]$. The higher the value of $\mu_A(x)$, the more x belongs to A. Zadeh [Zad65] introduced the following notation for fuzzy sets:

$$A = \sum_{i=1}^{n} \mu_A(x_i)/x_i = \mu_A(x_1)/x_1 + \ldots + \mu_A(x_n)/x_n \tag{A.2}$$

where $x \in X$, X is the universe of discourse and $\mu_A(x_i)$ is the membership function. In case that X is infinite another notation is used:

$$A = \int_X \mu_A(x)/x \tag{A.3}$$

Every element $x \in X$ has two attributes: a *label* and a *value* (or a property). The label is just a string, while the value a real number. The following example illustrates it:

Example 16 *Consider again the fuzzy set of fast cars from example 15. Every $x \in X$ represents a car, with a label (the name of the car) and a property (its speed). Using just introduced notation we obtain:*

$$A = 0/Trabant\_GTI + 0.55/Ford\_Escort + 0.8/Opel\_Astra+$$

$$+0.5/VW\_Golf + 1/Porsche\_S$$

if we use labels of the elements, and:

$$A = 0/80 + 0.55/155 + 0.8/165 + 0.5/150 + 1/250$$

if we use the property of the cars (in this case their speed). On the other hand, the fuzzy set "fast" from figure A.1 has an infinite universe. The following notation can be used:

$$F = \int_{speed} \mu_F(speed)/speed,$$

where the membership function of that fuzzy set is:

$$\mu_F(speed) = \begin{cases} 0 & if\ speed \leq 120, \\ (speed - 120)/60 & if\ 120 < speed < 180, \\ 1 & if\ speed \geq 180. \end{cases}$$

A fuzzy set is *normal* if the following is true:

$$\sup_{x \in X} \mu_A(x) = 1. \quad \text{(A.4)}$$

A fuzzy set A is *convex* if and only if for every pair of elements of X with values a, b, the membership function of A satisfies the following inequality:

$$\mu_A(\delta a + (1-\delta)b) \geq \min(\mu_A(a), \mu_A(b)), \quad \text{(A.5)}$$

where $\delta \in [0, 1]$ (see figure A.2).

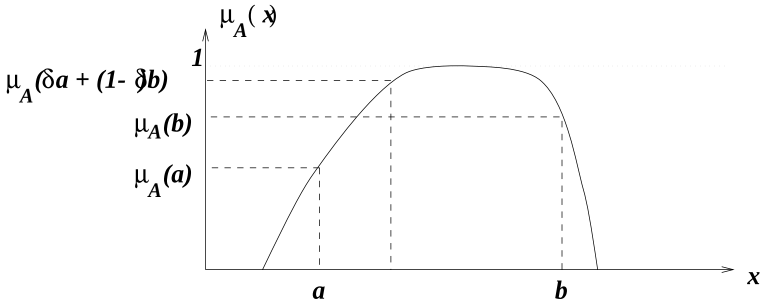


Figure A.2. A convex fuzzy set

The *support* of a fuzzy set A is the crisp subset of X, which has the membership degree larger than zero:

$$supp(A) = \{x | x \in X\ and\ \mu_A(x) > 0\}. \quad \text{(A.6)}$$

The *core* of a fuzzy set A is a crisp subset of X, for which the membership functions is equal 1:

$$core(A) = \{x | x \in X\ and\ \mu_A(x) = 1\}. \quad \text{(A.7)}$$

The α-$level\ set$ (or α cut) is a crisp subset of X defined by:

$$A_\alpha = \{x | x \in X\ and\ \mu_A(x) \geq \alpha\}. \quad \text{(A.8)}$$

The definitions of $core(A)$, $supp(x)$ and A_α are illustrated in figure A.3.

A *fuzzy number* is a special kind of fuzzy set for which the following conditions are satisfied:

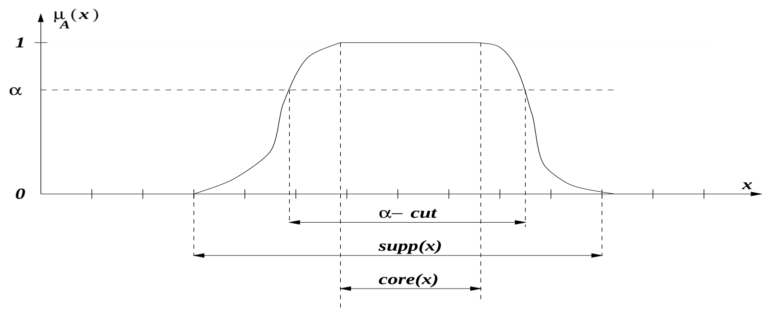


Figure A.3. An α-$level\ set$, $core\ set$ and $support$ of a fuzzy set

1. the fuzzy set is normal,
2. the fuzzy set is convex,
3. the membership function of the fuzzy set is piecewise linear,
4. the core of the fuzzy set contains only one point.

If the core is not restricted to one point we obtain a *fuzzy interval*. Examples of fuzzy number and fuzzy interval are presented in figure A.4.

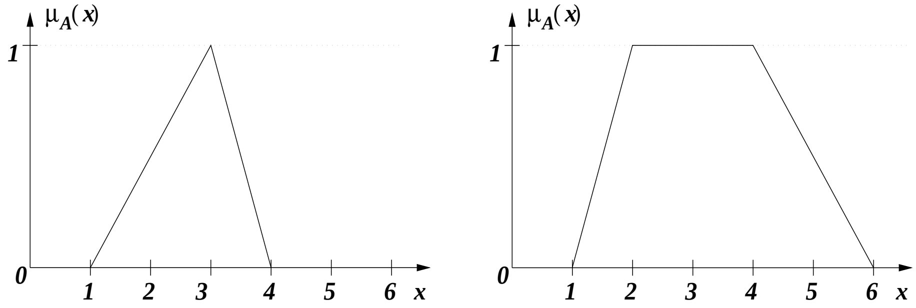


Figure A.4. A fuzzy number and fuzzy interval

The *Cartesian product* of fuzzy sets $A_1, \ldots, A_n$ is defined as:

$$A_1 \times \ldots \times A_n = \int_{X_1 \times \ldots \times X_n} \min[\mu_{A_1}(x_1), \ldots, \mu_{A_n}(x_n)]/(x_1, \ldots, x_2). \quad \text{(A.9)}$$

One of the most important elements of the fuzzy sets theory is *the extension principle*. Its importance comes from the fact that it provides a mechanism to deal with fuzzy quantities by extending classical (non-fuzzy) mathematics.

The *extension principle* allows us to induce a fuzzy subset B on Y through f on n fuzzy sets $A_1, \ldots, A_n$:

$$\mu_B(y) = \begin{cases} \sup_{y=f(x_1,\ldots,x_n)} & \min(\mu_{A_1}(x_1), \ldots, \mu_{A2}(x_n)) \\ 0 & if \;\; f^{-1}(y) = \emptyset \end{cases} \quad \text{(A.10)}$$

where $f^{-1}(y)$ is the inverse image of y. Let us illustrate the principle by means of a small example.

Example 17 *Consider the following fuzzy set* $A = 0.4/-2+0.5/-1+0.8/0+0.7/1$ *and the function* $f(x) = x^2$*. We will use the extension principle to calculate a square of the fuzzy set* A*. The equation A.10 can be simplified to:*

$$\mu_B(y) = \begin{cases} \sup_{x=f^{-1}(y)} \mu_A(x) & \\ 0 & if \;\; f^{-1}(y) = \emptyset \end{cases} \quad \text{(A.11)}$$

We obtain:

$$B = A^2 = (0.4/{-2} + 0.5/{-1} + 0.8/0 + 0.7/1)^2 = (0.8/0 + 0.5/1 + 0.4/4),$$

as illustrated in figure A.5.

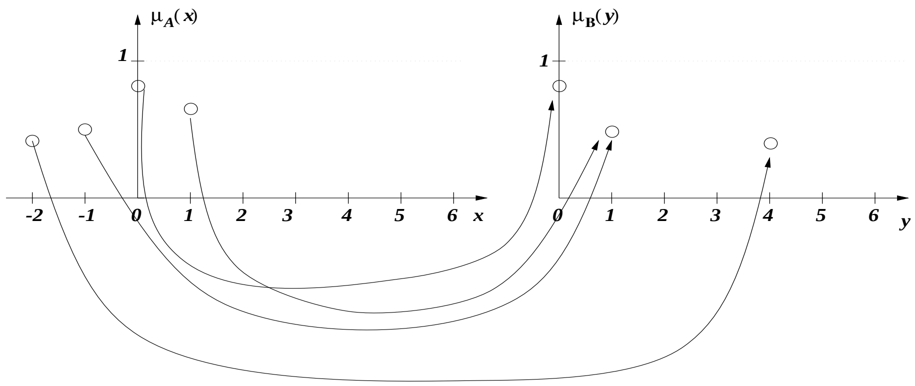


Figure A.5. The extension principle

A.3 Basic operations on fuzzy sets

In this section we will present an overview of basic set-theoretic operations on fuzzy sets. In all definitions below, A and B are two fuzzy subsets of X. Here follows the (incomplete) list:

- **Set inclusion:** The fuzzy set A is included in B ($A \subset B$), if:

$$\mu_A(x) \leq \mu_B(x), \;\; for \; all \; x \in X. \quad \text{(A.12)}$$

- **Set equality:** The fuzzy set A is equal to the set B (denoted $A = B$), if:

$$\mu_A(x) = \mu_B(x),\ for\ all\ x \in X. \tag{A.13}$$

- **Set complementation:** The A and B are complementary (denoted $B = \neg A$), if:

$$\mu_A(x) = 1 - \mu_B(x),\ for\ all\ x \in X. \tag{A.14}$$

- **Set intersection:** The intersection of A and B (denoted as $A \cap B$) is the largest fuzzy subset contained both in A and B:

$$\mu_{A\cap B} = \min(\mu_A(x), \mu_B(x)),\ for\ all\ x \in X. \tag{A.15}$$

- **Set union:** The union of A and B (denoted $A \cup B$) is the smallest fuzzy set containing both A and B:

$$\mu_{A\cup B} = \max(\mu_A(x), \mu_B(x)),\ for\ all\ x \in X. \tag{A.16}$$

A.4 Fuzzy arithmetics

In fuzzy mathematical programming programming we have to deal with equations containing fuzzy numbers. It is not possible to solve such an equation without defining operations of addition, subtraction, multiplication and division. Therefore in this section we will define these operations. We will concentrate only on triangular fuzzy numbers, since only they are used in this thesis. The triangular fuzzy number is denoted by $X = (x^m, x^l, x^u)$, where x^m is the central value (point where $\mu_X(x) = 1$), x^l and x^u are lower and upper bound values. In case of general fuzzy numbers the above mentioned operations can be accomplished by using $\alpha - level\ cut$ and max-min convolution [KG85]. The approach can be applied to triangular fuzzy numbers, and results in the following definitions.

Let $X = (x^m, x^l, x^u)$ and $Y = (y^m, y^l, y^u)$ be two triangular fuzzy numbers. Then, the following operations on X and Y are defined:

- **Inverse:**
$-X = (-x^m, -x^u, -x^l)$

- **Addition:**
$X(+)Y = (x^m + y^m, x^l + y^l, x^u + x^u)$

- **Subtraction:**
$X(-)Y = X(+) - Y = (x^m - y^m, x^l - y^u, x^u - x^l)$

- **Multiplication:**
$X > 0,\ Y > 0:\ X(\cdot)Y = (x^m y^m, x^l y^l, x^u, y^u)$
$X < 0,\ Y > 0:\ X(\cdot)Y = (x^m y^m, x^l y^u, x^l, y^u)$
$X < 0,\ Y < 0:\ X(\cdot)Y = (x^m y^m, x^u y^u, x^l, y^l)$

- **Scalar multiplication:**
$a > 0,\ a \in \mathbb{R}:\ a(\cdot)X = (ax^m, ax^l, ax^u)$
$a < 0,\ a \in \mathbb{R}:\ a(\cdot)X = (ax^m, ax^u, ax^l)$

- **Division:**
$X > 0,\ Y > 0:\ X(:)Y = (x^m/y^m, x^l/y^u, x^u/y^l)$
$X < 0,\ Y > 0:\ X(:)Y = (x^m/y^m, x^u/y^u, x^l/y^l)$
$X < 0,\ Y < 0:\ X(:)Y = (x^m/y^m, x^u/y^l, x^l/y^u)$

Example 18 *Consider the following two triangular fuzzy numbers:* $X = (3, 1, 4)$ *and* $Y = (2, 1, 3)$ *(figure A.6). Then applying formulas given above we obtain fuzzy numbers:*
$-X = (-3, -4, -1)$
$X(+)Y = (5, 2, 7)$
$X(-)Y = X(+) - Y = (1, -1, 3)$
$X(\cdot)Y = (6, 1, 12)$
$X(:)Y = (1.5, 0.33, 4).$
This has been illustrated in figure A.7.

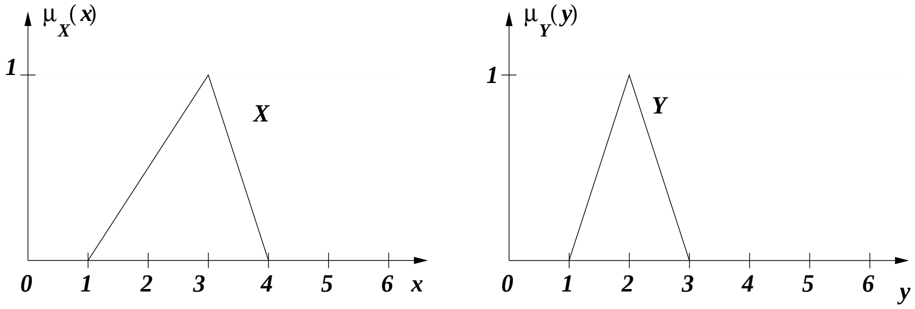


Figure A.6. Two fuzzy numbers

A.5 Possibility theory

Possibility theory is based on fuzzy sets theory and has been introduced by Zadeh [Zad78]. In this section we will present Zadeh's definition of *possibility distributions*, and his interpretation of them. Let us start with a so called *possibility assignment equation*:

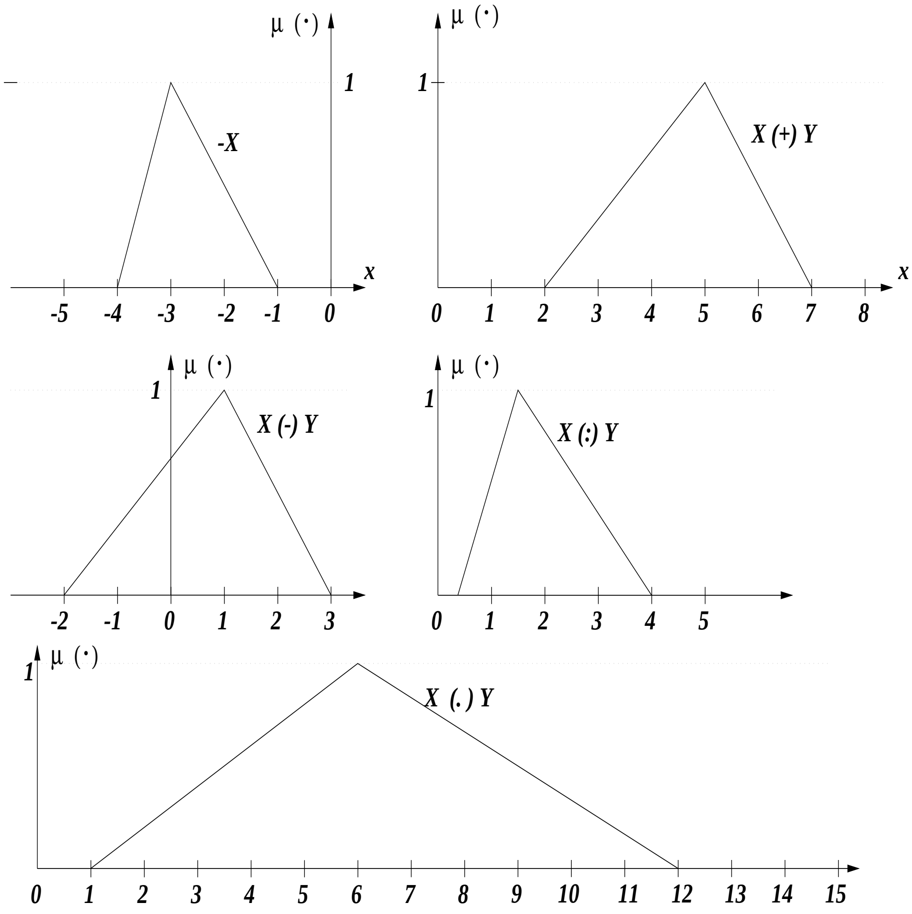


Figure A.7. Fuzzy arithmetics demonstration

Consider the following proposition:

$$B \ is \ A$$

where A is a fuzzy set. For instance: *the temperature is low*. In this case B is *the temperature*, while A denotes fuzzy set *low*.

The proposition induces a possibility distribution π_B, which is by definition equal to the fuzzy set A:

$$\pi_B = A \tag{A.17}$$

If the fuzzy set A has a membership function $\mu_A(x)$ then the possibility that $B = x$ is given by:

$$\pi_B(x) \stackrel{\text{def}}{=} \mu_A(x), x \in X. \tag{A.18}$$

Let us illustrate it with a small example:

Example 19 *Consider again the proposition:* the temperature is low. *Then* the temperature *is our imprecise variable* x*, and the following can be derived:*

$$\pi_{temperature}(x) = \mu_{low}(x)$$

where $\pi_{temperature}(x)$ *is the possibility distribution function and* $\mu_{low}(x)$ *is the membership function of the fuzzy set* low.

As can be seen in possibility theory the interpretation of fuzzy sets is different than what we described in previous sections. Since it usually causes much misunderstanding, let us try to elaborate on this difference. Depending on the context, the fuzzy sets can be used in the following two ways:

- *As a model of the gradual nature of properties*. This way we used them in previous sections. Consider the following proposition *the price is high*. The fuzzy set can be used to model linguistic uncertainly (or ambiguity) of the term "high". Having the membership function $\mu_{high}(x)$ and the price we can determine if the proposition is true.

- *As a model of incomplete knowledge*. There are many situations where precise value of a variable is not known. We may only know that x is A, and therefore the x is not precisely known. In such case the fuzzy set A can be considered a possibility distribution. The higher value of $\mu_A(y)$, the higher possibility that $x = y$. Consider again the proposition *the price is high*. We only know that the price is high, while the price itself is not exactly known. The fuzzy set *high* is a possibility distribution, because the higher value of $\mu_{high}(y)$, the higher possibility that the price is indeed y.

Whether we use fuzzy sets in the first or the second way depends on our current context. In this thesis, we will use them only as possibility distributions. We would only like to mention that there exist other interpretations of the possibility assignment equation (eq. A.17) [DP91].

Bibliography

[Ake78] S.B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–616, June 1978.

[AU77] A.V. Aho and J.D. Ullman. *Principles of Compiler Design*. Addison-Wesley, Reading, 1977.

[Bak74] K.R. Baker. *Introduction to Sequencing and Scheduling*. John Wiley & Sons, 1974.

[Bak90] H.B. Bakoglu. *Circuits, interconnections, and packaging for VLSI*. Addison-Wesley Publishing Company, 1990.

[BBC+85] R.K. Brayton, N.L. Brenner, C.L. Chen, G. De Micheli, C.T. McMullen, and R.H.J.M. Otten. The yorktown silicon compiler. In *ISCAS'85, Kyoto, Japan*, 1985.

[BDM94] L. Benini and G. De Micheli. State assignment for low power dissipation. Technical report, Stanford University, 1994.

[Bel57] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[BHLG94] J. T. Buck, S. Ha, E. A. Lee, and Messerschmitt D. G. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation*, 4:155–182, April 1994. special issue on Simulation Software Development.

[BHMSV84] R.K. Brayton, G.D. Hachtel, C.T. McMullen, and A.L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[BJ90] M.R.C.M. Berkelaar and J.A.G. Jess. Gate sizing in mos digital circuits with linear programming. In *Proceedings of the European Design Automation Conference*, pages 217–221, 1990.

[BJ91] T.N. Blalock and R.C. Jaeger. A high-speed clamped bit-line current-mode sense amplifier. *IEEE Journal of Solid-State Circuits*, 26(4), 1991.

[Bok81] S.H. Bokhari. A shortest tree algorithm for optimal assigments across space and time in a distributed processor system. *IEEE Transactions on Software Engineering*, SE-7(6):583–589, November 1981.

[BRSVW87] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A.R. Wang. Mis: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):1062–1081, November 1987.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Cap95] P. Cappelletti. *The STONE user's manual*. Delft University of Technology, 1995.

[Cha79] L.G. Chacijan. Polinomialnij algoritm v linejnom programmirovanii. *Dokl. AN SSSR*, 244:1093–1096, 1979.

[CJP83] H. Crowder, E. Johnson, and M. Padberg. Solving large-scale zero-one linear programming problems. *Opns. Res.*, 7:56–66, 1983.

[CLR90] T.H. Corman, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. McGraw-Hill, MIT-Press, 1990.

[CPL93] CPLEX Optimization Inc. *The CPLEX Linear Optimizer and Mixed Integer Optimizer with Barrier. User manual*, second edition, 1993.

[Dep88] E. F. Deprettere. *SVD and signal processing; applications and architectures*. North-Holland, Amsterdam, 1988.

[Don79] W.E. Donath. Placement and average interconnection lengths of computer logic. *IEEE Trans. on Circuits and Systems*, CAS-26(4):272–277, 1979.

[DP91] D. Dubois and H. Prade. Fuzzy sets in approximate reasoning, part 1: Inference with possibility distributions. *Fuzzy Sets and Systems*, 40:143–202, 1991.

[Dye80] M.E. Dyer. Calculating surrogate constraints. *Math. Prog.*, 16:255–278, 1980.

[Dye82] M.E. Dyer. An $o(n)$ algorithm for the multiple-choice knapsack linear problem. *Math. Prog.*, 29:57–63, 1982.

[EA92] E.Aardoom and A.K.Nieuwland. Gollum: A vlsi multi-system navigation receiver. In *Proceedings of ION GPS-92*, pages 1111–1116. The Institute of Navigation, September 1992.

[EHB93] R. Ernst, J. Henkel, and T. Benner. Hardware-software cosynthesis for microcontrolers. In *IEEE Design & Test of Computers*, December 1993.

[FD85] J.P. Fishburn and A.E. Dunlop. Tilos: A posynomial programming aproach to transistor sizing. In *Proc. of the IEEE Int. Conference on Computer Aided Design*, pages 326–328, 1985.

[Fou81] L.R. Foulds. *Optimization Techniques: An Introduction*. Springer-Verlag, 1981.

[GCM92] R.K. Gupta, C.N. Coelho, and G. De Micheli. Synthesis and simulation of digital systems containing interacting hardware and software components. In *Proc. of DAC'92*, pages 225–230, 1992.

[GE90] C.H. Gebotys and M.I. Elmasry. A global optimization approach to architectural synthesis. In *IEEE International Conference on Computer Aided Design*, pages 258–261, 1990.

[Geo69] A.M. Geoffrion. An improved implicit enumeration approach for integer programming. *Opns. Res.*, 17:437–454, 1969.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of the NP-Completeness.* Freeman, San Francisco,CA, 1979.

[Gla94] J.P.F. Glas. On multiple access interference in a ds/ffh spread spectrum communication system. In *Proceedings of the Third IEEE International Symposium on Spread Spectrum Techniques and Applications*, Oulu, Finland, July 1994.

[Gla95] Jack P.F. Glas. An embedded cdma-receiver: A design example. Technical report, Delft University of Technology, 1995.

[Gom58] R.E. Gomory. Outline for an algorithm for integer solutions to linear programs. *Bull. Amer. Soc. Math.*, 64:275–278, 1958.

[GPB92] P. Gutwin, McGeer P., and K. Brayton. Delay prediction for technology-independent logic equations. In *Proc. of the International Conference on Computer Design*, pages 468–471, 1992.

[Gro91] P. Groeneveld. *Context-Driven Channel Routing*. PhD thesis, Delft University of Technology, 1991.

[GS91] P. Groeneveld and P. Stravers. All you ever wanted to know about fish, fishbone and ghoti. Technical report, Delft University of Technology, January 1991.

[GS93] P. Groeneveld and P. Stravers. *OCEAN: The Sea-of-Gates Design System. On-line users manual. (http://olt.et.tudelft.nl/~patrick/docs/wwman/wwman.html)*, 1993.

[GS94] P. Groeneveld and P. Stravers. A robust design environment for vlsi training with a very short learning curve. In *Proceedings of the fifth Eurochip workshop on VLSI training*, October 1994.

[GT87] A.V. Goldberg and R.E. Tarjan. Finding minimum-cost circulations by successive approximation. Technical memorandum mit/lcs/tm-33, MIT, 1987.

[HCO74] S. Hong, R. Cain, and D. Ostapko. Mini: A heuristic approach for logic minimization. *IBM Journal of Research and Development*, 18:443–458, September 1974.

[HRSJ91] G.D. Hachtel, J.K. Rhio, F. Somenzi, and R. Jacoby. Exact and heuristic algorithms for the minimization of incompletely specified state machines. In *Proc. of the European Conference on Design Automation*, pages 184–191, Amsterdam, 1991.

[Ins78] Institute of Electrical and Electronics Engineers, New York. *IEEE Standard Digital Interface for Programmable Instrumentation*, 1978.

[Ins85] Institute of Electrical and Electronics Engineers, New York. *Token-passing Bus Access Method and Physical Layer Specification*, 1985.

[Ins89] Institute of Electrical and Electronics Engineers, New York. *Information Processing Systems; Local Area Networks. Pt. 3. Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*, 1989.

[JCV+86] R. Jain, F. Catthoor, J. Vanhoof, B. Deloore, G. Goossens, N. Goncalves, L. Claesen, J. Van Ginderdeurer, J. Vandelwalle, and H. De Man. Custom design of a vlsi plm-fdm transmultiplexer from system specifications to layout using a cad system. *IEEE Journal of Solid State Circuits*, SC21(1):73–85, January 1986.

[JE89] M.A.B. Jackson and Kuh E.S. Performance-driven placement of cell based ic's. In *Proc. of the 26th ACM/IEEE Design Automation Conference*, pages 370–374, 1989.

[Kao85] W.H. Kao. Algorithms for automatic transistor sizing in cmos digital circuits. In *Proc. of the IEEE Design Automation Conference*, pages 781–784, 1985.

[Kar91] I. Karkowski. Image independent program for detailed placement. Master's thesis, Technical University of Gdansk, Gdansk, Poland, August 1991.

[Kar93] I. Karkowski. Circuit delay optimization as a multiple choice linear knapsack program. In *Proceedings of the European Design Automation Conference*, pages 419–423, 1993.

[Kar94] I. Karkowski. *Scherzo Schematic Editor Tutorial*. Delft University of Technology, The Faculty of Electrical Engineering, Delft, June 1994.

[Kar95] I. Karkowski. Architectural synthesis with possibilistic programming. In *Proceedings of the HICSS-28*, January 1995.

[KCH82] R.H. Krambeck, Lee C.M., and Law H.S. High-speed compact circuits with cmos. *IEEE Journal of Solid-State Circuits*, SC-17:614–619, June 1982.

[KDM90] D. Ku and G. De Micheli. Relative scheduling under timing constraints. In *Proc. of the 27th Design Automation Conference*, pages 59–64, June 1990.

[KDM92] D.C. Ku and G. De Micheli. *High level synthesis of ASICs under timing and synchronization constraints*. Kluwer Academic, Boston, 1992.

[KG85] A. Kaufmann and M.M. Gupta. *Introduction to Fuzzy Arithmetics: Theory and Applications*. Van Nostrand Reinhold, New York, 1985.

[KM72] V. Klee and G.J. Minty. How good is the simplex algorithm? *in Shisha(ed.)*, pages 159–116, 1972.

[KO95] I. Karkowski and R.H.J.M. Otten. Retiming synchronous circuitry with imprecise delays. In *Proceedings of 32nd DAC*, June 1995.

[KO96] I. Karkowski and R.H.J.M. Otten. An Automatic Hardware-Software Partitioner Based on the Possibilistic Programming. In *Proceedings of the ED&TC Conference*, Paris, March 1996.

[KR88] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall, 1988.

[KSF94] L.N. Kannan, P.R. Suaris, and H.G. Fang. A methodology and algorithms for post-placement delay optimization. In *Proc. of the 31th ACM/IEEE Design Automation Conference*, pages 327–332, 1994.

[Lew84] E.T. Lewis. Optimization of device area and overall delay for cmos vlsi designs. *Proceedings of IEEE*, 72(6):670–689, June 1984.

[LH92] Y.J. Lai and C.L. Hwang. A new approach to some possibilistic linear programming problem. *Fuzzy Sets and Systems*, 49:121–133, 1992.

[Lis89] R. Lisanke. Logic synthesis benchmark circuits for the international workshop on logic synthesis. Technical report, Department of Electrical Engineering and Computer Science, University of California, Berkeley, May 1989.

[LRS83] C. Leiserson, F. Rose, and J. Saxe. Optimizing synchronous circuitry by retiming. In *Proc. of the 3th Caltech Conference on VLSI*, 1983.

[LS91] C. Leiserson and J. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–25, 1991.

[Mar89] D. Marple. Transistor size optimization in the tailor layout system. In *Proc. of the IEEE Design Automation Conference*, pages 43–48, 1989.

[Maz92] S. Mazor. *A guide to VHDL.* Kluwer Academic, Boston, 1992.

[MC80] C Mead and L. Conway. *Introductions to VLSI systems*. Addison-Wesley, 1980.

[McC56] E. McCluskey. Minimization of boolean functions. *The Bell System Technical Journal*, 35:1417–1444, November 1956.

[Min80] G.J. Minty. On maximal independent sets of vertices in a claw-free graph. *Journal of Combinatorial Theory*, B28:284–304, 1980.

[MR79] Karwan M.H. and Radin R.L. Some relations between lagrangian and surrogate duality in integer programming. *Math. Prog.*, 17:320–334, 1979.

[MSBSV91] S. Malik, E. Sentovich, R. Brayton, and A. Sangiovanni-Vincentelli. Retiming and resynthesis: Optimizing sequential networks with combinatorial techniques. *IEEE Transactions on CAD/ICAS*, CAD-10(1):74–84, January 1991.

[Ott83] R.H.J.M. Otten. Efficient floorplan optimization. In *Proceedings of ICCD*, pages 499–502, 1983.

[OvG89] R.H.J.M. Otten and L.P.P.P. van Ginneken. *The annealing algorithm.* Kluwer, Boston, 1989.

[Phi84] Philips Reseach Laboratories, Eindhoven. *D^2B Specification*, 1984.

[Phi88] Philips Reseach Laboratories Eindhoven. *C3TU Blue Book*, May 1988.

[PP92] S. Prakash and A.C. Parker. Sos: Synthesis of application-specific heterogeneous multiprocessor systems. *IEEE Journal on Parallel and Distributed Computing*, page 338, 1992.

[Qui52] W. Quine. The problem of simplifying truth functions. *American Mathematical Monthly*, 59:521–531, 1952.

[RPG76] A.U. Ruehli, Wolff P.K., and Goerzel G. Power and timing optimization of large digital systems. In *Proc. of the IEEE Int. Symposium on Circuits And Systems*, pages 402–405, 1976.

[RR85] J. Ramin and J. Rimanek. Inequality relation between fuzzy numbers and its use in fuzzy optimization. *Fuzzy Sets and Systems*, 16:123–138, 1985.

[RS91] Tsay Ren-Song. Exact zero skew. In *Proc. of ICCD*, 1991.

[Sak83] T. Sakurai. Approximation of wiring delay in mosfet lsi. *IEEE Journal of Solid-State Circuit*, SC-18:418–426, Aug 1983.

[SDM91] P.J. Song and G. De Micheli. Circuit and architecture trade-offs for high-speed multiplication. *IEEE Journal of Solid-State Circuits*, 26(9), September 1991.

[Sec87] Carl Sechen. Average interconnection length estimation for random and optimized placement. In *Proceedings of the International Conference on Computer-Aided Design*, pages 190–193, 1987.

[Sen72] J.K. Sengupta. *Stochastic Programming*. North Holland Publishing Company, 1972.

[SFHS91] Y. Sekiyama, Y. Fujihara, T. Hayashi, and M. Seki. Timing-oriented routers for pcb layout design of high-performance computers. In *Proc. of the International Conference on Computer Design*, page 332335, 1991.

[Sin92] K.J. Singh. *Performance Optimization of Digital Circuits*. PhD thesis, University of California at Berkeley, 1992.

[SM84] I.M. Stancu-Minasian. *Stochastic Programming with Multiple Objective Functions*. D. Reidel, Dordrecht, 1984.

[SSL+92] E. M. Santovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. Sis: A system for sequential circuit synthesis. Technical report, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1992.

[SSVFD88] J Shyu, A Sangiovanni-Vincentelli, J.P. Fishburn, and A.E. Dunlop. Optimization based transistor sizing. *IEEE Journal of Solid-State Circuits*, 23:400–409, April 1988.

[Sto77] H.S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, SE-3(1):85–93, 1977.

[Str91a] P. Stravers. Partitioning a network into n pieces with a time efficient netcost function. In *Proc. of the European Conference on Design Automation*, pages 177–182, Los Alamitos, California, February 1991. IEEE Computer Society Press.

[Str91b] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1991.

[SWBSV88] K.J. Singh, A.R. Wang, R.K. Brayton, and A. Sangiovanni-Vincentelli. Timing optimization of combinational logic. In *Proc. of the International Conference on Computer-Aided Design*, pages 282–285, November 1988.

[Szy92] T.G. Szymanski. Computing optimal clock schedules. In *Proc. of the 29th ACM/IEEE Design Automation Conference*, pages 399–404, 1992.

[TAM93] V. Tiwari, P. Ashar, and S. Malik. Technology mapping for low power. In *Proc. of the 30th ACM/IEEE Design Automation Conference*, pages 74–79, 1993.

[TM91] D.E Thomas and P.H. Moorby. *The Verilog hardware description language*. Kluwer, Boston, 1991.

[Tou90] H.J. Touati. *Performance-Oriented Technology Mapping*. PhD thesis, University of California Berkeley, 1990.

[Tow86] D.F. Towsley. Allocating programs containing branches and loops within a multiple processor system. *IEEE Transactions on Software Engineering*, SE-12:1018–1024, October 1986.

[TPD93] C.Y. Tsui, M. Pedram, and A.M. Despain. Technology decomposition and mapping targeting low power dissipation. In *Proc. of the 30th ACM/IEEE Design Automation Conference*, pages 68–73, 1993.

[TSB91] H. Touati, H. Savoy, and R.K. Brayton. Delay optimization of combinational logic circuits by clustering and partial collapsing. In *Proc. of the International Conference on Computer-Aided Design*, November 1991.

[TSV94] M. Theißinger, P. Stravers, and H. Veit. Castle: An interactive environment for hw-sw co-design. In *Proc. of the HW-SW Co-Design Conf.*, 1994.

[vGdG87] A. van Genderen and A.C de Graaf. *SLS: Switch-Level Simulator User's Manual*. Delft University of Technology, 1987.

[vGvdMB95] A.J. van Genderen, N.P. van der Meijs, and F. Beeftink. Space: An accurate layout-to-circuit extractor for high-speed mos and bipolar circuits. In *Proc. of the ProRISC/IEEE Benelux Workshop on Circuits, Systems and Signal Processing*, March 1995.

[VZN+81] A. Vladimirescu, K. Zhang, A.R. Newton, D.O. Pederson, and A. Sangiovanni-Vincentelli. *SPICE Version 2G Users' Guide*, August 1981.

[WE92] N.H.E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design*. Addison-Wesley, 1992.

[Zad65] L.A. Zadeh. Fuzzy sets. *Information and Control*, 8:94–102, 1965.

[Zad78] L.A. Zadeh. Fuzzy sets as a basis for a theory of possibility. *Fuzzy Sets and Systems*, 1:3–28, 1978.

SUMMARY

Performance Driven Synthesis of Digital Systems

Due to the tremendous progress in the microelectronics of last years a shift of interest in the optimization methods of the digital systems has been observed: from minimization of chip area to the maximization of performance. In the same time the increase in complexity of the systems resulted in situation, where the designer's role is being constantly moved to the higher design levels. This causes a need to create new design automation tools and to rewrite most of the existing ones. One of the urgent problems addressed in this thesis is how to reduce a risk of redesigns in the automatic design process. We propose two solutions. One, called here *the time-budgets* method is aimed at removing the need for redesigns, but it cannot be applied at all the stages of the synthesis process. We propose therefore also another solution, which is wider applicable and more practical. It decreases significantly the risk that a redesign is necessary. The approach is based on possibilistic programming, a branch of fuzzy mathematical programming.

First, in chapter 1 we give an overview of past and present design automation systems. Four self contained **CAD** systems are described (we select them on the basis of local usage). They cover together a whole design trajectory starting from a description in high-level language to a working layout.

Chapter 2 is devoted to the issue of *design trade-offs*. It is illustrated by the so called "high C" circuits (a transmitter-receiver pair sending a signal over high capacitance wire). Several "high C" circuits are presented with different performance, power dissipation and area characteristics. After defining a delay model the circuits are compared based on simulation results. The so called design *trade-off curves* are generated. At the end we list additional sources of the imprecision of circuit delay models. We argue that there are enough reasons to treat the problem seriously. This conclusion forms one of the motivations for the algorithms presented in this thesis.

Since most of the methods described in this thesis are based on *integer programming* (shortly **IP**), chapter 3 surveys some **IP** methods. After that, in the same chapter, the concepts of possibilistic programming are described.

In chapter 4 our first solution to the problem of performance driven synthesis (called the *time-budgets method*) is introduced. The next chapter is devoted to three successful applications of the possibilistic programming. We selected three subtasks belonging to different stages in the automated synthesis trajectory: the problem of *retiming*, the problem of *simultaneous scheduling, selection and allocation of functional units*, and the problem of *hardware-software partitioning of embedded systems*.

In chapter 6 we present the implementation details and experimental results for all proposed algorithms. In addition, we describe our general **PIP/PLP** solver called FSOLVE.

Ireneusz Karkowski

Keywords: design automation, delay modeling, high-level synthesis, logic-synthesis, layout synthesis, embedded systems, application-specific hardware, hardware-software codesign, OCEAN, design trade-offs, sea-of-gates, fishbone, integer programming, fuzzy sets and systems, possibilistic programming.

SAMENVATTING

Performance-gedreven synthese van digitale systemen

De snelle ontwikkeling in de microelectronica-wereld van de afgelopen jaren heeft geleid tot een duidelijke verschuiving in de optimalisatie van digitale systemen: van minimalisatie van de chip-oppervlakte tot maximalisatie van de snelheid. Tegelijkertijd is de complexiteit van de systemen sterk toegenomen met als resultaat dat de werkwijze van de ontwerper is opgeschoven in de richting van hoog-niveau synthese. Nieuwe ontwerpautomatiseringshulpmiddelen zijn hiervoor noodzakelijk en veel oude programma's moeten worden aangepast. Dit proefschrift behandelt één van de belangrijke problemen: hoe kan het risico van een *redesign* in het automatisch ontwerptraject gereduceerd worden. Wij stellen twee oplossingen voor. In een zogenaamde *time-budgets*-methode wordt het risico van een *redesign* helemaal weggenomen. Deze methode kan echter niet in alle fasen van het ontwerpproces worden toegepast. Daarom stellen wij ook een andere oplossing voor, die veel veelzijdiger en praktischer is. Deze methode verkleint in sterke mate het risico dat een *redesign* noodzakelijk is. De benadering is gebaseerd op *possibilistic programming*, een soort *fuzzy mathematical programming*.

In hoofdstuk 1 geven wij een overzicht van de conventionele en huidige ontwerpautomatisering systemen. Vier complete **CAD** systemen worden beschreven (gekozen op basis van wat aan de TU Delft gebruikt wordt). Samen omvatten ze het gehele ontwerptraject, van een ontwerp-specificatie in een hoog-niveau taal naar een werkende layout.

In hoofdstuk 2 is het onderwerp *ontwerpafwegingen* behandeld. Dit wordt geïllustreerd door de "hoge-C" schakelingen (een transmitter-ontvanger paar die een digital signaal over een lijn met hoge capaciteit stuurt). Enkele "hoge-C" circuits, met verschillende performance, vermogen-verbruik en oppervlakte karakteristieken, worden gepresenteerd. Na de definitie van het vertragingsmodel, vergelijken we de schakelingen op basis van simulatieresultaten. De zogenoemde *curves van de ontwerpafwegingen* worden gegenereerd. Tenslotte presenteren wij een lijst met extra bronnen van onnauwkeurigheden in de vertragingsmodellen. Er worden argumenten aangevoerd waarom deze problemen serieus genomen moeten worden. Deze conclusie vormt één van de motiveringen voor de in dit proefschrift gepresenteerde algoritmen.

Omdat de meeste in deze dissertatie beschreven methoden gebaseerd zijn op *geheeltallig programmeren* (afkorting **IP**), geeft hoofdstuk 3 een overzicht van enkele **IP**-methoden. Daarna worden, in ditzelfde hoofdstuk, de beginselen van *possibilistisch programmeren* beschreven.

In hoofdstuk 4 wordt de eerste oplossing voor het probleem van performance-gedreven synthese van digitale systemen, de *time-budgets-methode*, geïntroduceerd. In het daarop volgende hoofdstuk worden drie succesvolle toepassingen van *possibilistisch programmeren* gepresenteerd. Er zijn drie onderdelen geselecteerd, die deel uitmaken van de verschillende fasen in het automatisch ontwerptraject: *retiming*, *simultaneous scheduling, selection and allocation of functional units* en *hardware-software partitioning of embedded systems*. De implementatie-details en de resultaten van experimenten met alle optimalisatie programma's worden gepresenteerd in hoofdstuk 6. Als appendix presenteren wij daar onze *possibilistisch* solver FSOLVE.

Ireneusz Karkowski

Keywords: ontwerpautomatisering, vertragingsmodellen, hoog-niveau synthese, ontwerpafwegingen, logische synthese, layout synthese, ingebedde systemen, toepassingspecifieke elektronika, hardware–software co-ontwerp, OCEAN, sea-of-gates, fishbone, geheeltallig programmeren, vage verzamelingen, possibilistisch programmeren.

STRESZCZENIE

Synteza układów cyfrowych zoriętowana na optymalizację szybkości

Szybki postęp w mikroelektronice ostatnich lat spowodował wyraźną zmianę wśród metod optymalizacji układów cyfrowych: z minimalizacji powierzchni układu scalonego w kierunku maksymalizacji jego szybkości. Jednocześnie na skutek wzrostu stopnia złołoności tych układów, rola projektanta uległa dalszemu przesunięciu w kierunku wyŗszych poziomów projektowych. Spowodowało to konieczność napisania nowych narzędzi projektowych, wiele starych programów musiało zostać zmodyfikowanych. W niniejszej rozprawie zajmujemy się jednym z podstawowych problemów: zmniejszeniu ryzyka iŕ na skutek błędnych decyzji systemowych otrzymany układ nie satysfakcjonuje ułytkownika. Pomyłka tego typu zwykle kończy się koniecznościa powtórzenia części lub całości wykonanej pracy. Proponujemy dwa rozwiązania tego problemu. Metoda "budŗetów czasowych" usuwa całkowicie ryzyko powtórek projektowych. Nie moŕe być ona jednak stosowana na wszystkich etapach procesu projektowego. Dlatego teŕ przedstawiamy również inne, bardziej uniwersalne podejście. Wywodzi się ono z dziedziny programowania rozmytego (*possibilistic programming*) i znacznie zmniejsza ryzyko powtórek w procesie projektowym.

Rozdział 1 zawiera przegląd istniejących systemów do automatycznej syntezy układów cyfrowych. Cztery kompletne systemy CAD, razem pokrywające wszystkie fazy projektowe, są szczegółowo opisane.

Rozdział 2 jest poświęcony zagadnieniu *alternatyw projektowych*. Jako ilustracje prezentujemu tzw. układy "high-C" (para składająca się z nadajnika oraz odbiornika przesyłająca sygnał cyfrowy przez połączenie o wysokiej pojemności). Zalety oraz wady poszczególnych rozwiązań sa dyskutowane. Po zdefiniowaniu uŕytego modelu opóżnień układowych, charakterystyki uzyskane z symulacji wszystkich układów są zestawione w formie *krzywych alternatyw projektowych*. W końcowej części rozdziału, omawiamy dodatkowe zródła nieprecyzyjności modeli opóżnień. Przedstawione są argumenty świadczące o tym, ŗe problem ten powinien być traktowany poważnie. Ta konkluzja leŕy u podstaw metod prezentowanych w tej rozprawie.

Jako ŕe większość opisanych w tej pracy algorytmów jest oparta na programowaniu całkowitoliczbowym, rozdział 3 zawiera przegląd stosowanych metod programowa-

nia całkowitoliczbowego. W drugiej części opisujemy wybrane metody programowania rozmytego.

W rozdziale 4 prezentujemy pierwszy z proponowanych algorytmów, tzw. metodę "budŗetów czasowych". Druga z metod, oparta na *possibilistic programming*, opisana jest w następnym rozdziale. Jako demontracja trzy wybrane problemy naleŗące do róŗnych faz procesu projektowego sa omawiane: problem *retiming'u* układów sekwencyjnych, problem szeregowania zadań w tzw. *data paths* oraz problem podziału zadań pomiędzy hardware i software w tzw. *embedded systems*.

Rozdział 6 zawiera szczegóły implementacji oraz wyniki eksperymentów z wszystkimi proponowanymi w tej pracy algorytmami. Dodatkowo opisujemy tam uniwersalny program do optymalizacji rozmytej o nazwie FSOLVE.

Ireneusz Karkowski

Słowa kluczowe: automatyczne projektowanie, modelowanie opóżnień, synteza systemowa, synteza układów logicznych, synteza layoutu, współbiełna synteza hardware'u i software'u, embedded systems, OCEAN, styl sea-of-gates, krzywe alternatyw projektowych, układy scalone sea-of-gates, fishbone, programowanie dyskretne, zbiory rozmyte, possibilistic programming.

ACKNOWLEDGEMENTS

I would like to thank all people from the electronic systems implementation group for creating a very stimulating research environment in which this work could be completed. First of all to Prof. Ralph Otten for his guideness and very helpful discussions, Dr. Mirosław Kwiesielewicz from Technical University of Gdańsk for the introduction to the fuzzy sets theory and Martin Huisman for designing the electronic circuits. I am also grateful to Prof. Giovanni De Micheli of Stanford University, Dr. Patrick Groeneveld, Jacek Serafin, Jack Glas and Cezar Bruma for their valuable remarks, Marion de Vlieger for correcting my version of "Samenvating". Dr. Tamas Terlaky was very helpful in choosing the right linear optimizer package.

Index

ABOUT THE AUTHOR

Ireneusz Karkowski was born in Gdynia, Poland on April 15, 1966. In May 1985 he received his final diploma from the "Liceum Og"olnokszta"lc"ace nr. 1" in Rumia. In October of that year he entered the Electronics Department of the Technical University of Gda"nsk, in Gda"nsk (Poland), as undergraduate student. The last year of his studies he spent at the Delft University of Technology, the Netherlands, where he worked on his master thesis - *"Image Independent Program For Detailed Placement on Sea-of-gates"*. On October 2nd, 1991 he received his Magister In"rynier Elektroniki degree (equivalent of an M.Sc). In November 1991 he joined the electronic systems implementation group of prof. R. Otten at Delft University of Technology, to work on the "Simultaneous delay and area allocation" for sea-of-gates (IOP project). His research interests evolved from layout design automation tools, through logic and high-level synthesis algorithms, to algorithms for hardware-software co-synthesis of embedded systems, as reported in this thesis.