RAFAEL: AN INTELLIGENT, MULTI-TARGET SIGNAL-FLOW COMPILER

by

Gábor Paller

July 31, 1995
Third final version

Technical University of Budapest
Department of Electromagnetic Theory, DSP group

This page is intentionally left blank

Acknowledgements

First I owe my thesis responsable, Ernő Simonyi thanks for showing me the direction of this thesis and sharing his experience with me during my work. I am especially grateful Klára Cséfalvay for her incredible work in creating the DSP group at the Department of Electromagnetic Theory and helping me when I was lost. I say a big THANK YOU to all the members of the DSP group whose enthusiasm brought and brings forward the projects of this group.

From the French part my thanks go first to Christophe Wolinski who was professionally responsible for my final 9 month stay at IRISA, Rennes. Valuable comments, critiques, helps of Christophe greatly contributed that this thesis is at least readable ... I am very grateful to Paul Le Guernic, head of the EP-ATR project in IRISA for accepting me in his group during my stay in IRISA and for carefully following my work. I must hereby thank all members of the EP-ATR group who showed me hope when I got lost in the details of the SIGNAL language and its compiler. I am obliged the Government of France which provided the scholarship for the stay at IRISA.

I am also obliged Christophe Lavarenne and Yves Sorel from the SynDEx team of INRIA, Rocquencourt who provided a one month stay at INRIA and helped me to develop the routing aspects of the Springplay scheduler.

At last I recall all the friends, acquintances and hitchhikers who made the stay on the Land of Bretons unforgettable.

Gábor Paller

.....

This page is intentionally left blank

Contents

Chapter 1

Introduction

1.1 The aim of the Rafael project

Writing programs for the modern Digital Signal Processors (DSPs) introduces difficult tasks for the software engineers because a painful tradeoff exists between the computing power and the productivity/task complexity. Unfortunately the existing and well-known higher level programming environments (for example the "C" language) performs very poorly on the DSP platforms because being general languages they cannot exploit the special capabilites of the DSPs (circular buffers, parallel instructions and so on) or avoiding pipeline effects. This can cause extremely high performance loss (can be as much as 1000% compared to the assembly realization). Several developments were made to improve C compilers on DSP platforms [Lear90] but generally they use system or DSP dependent language extensions and their performance is still not really convincing. So the developers have to choose - writing the DSP code in assembly for achieving higher performance thus lower hardware cost or using a high-level environment which will speed up the development but decrease the efficiency of the DSP so that more expensive DSP-s must be chosen. It can even happen that the problem cannot be solved on high level.

The other problem is the embarassing abundance of DSP architectures and languages. One often faces the problem of porting existing results onto other DSP platforms. If the code is written in assembly , this will be a long and tiresome process. Some "common language" is needed but not having efficiently realizable high level platform this solution does not seem to be promising. Nowadays the solution is sought toward optimized software

libraries (like the SPOX math libraries) which try to combine the power of assembly routines with the efficiency of C. SPOX [Spox88] does accelerate the developing process but it is a fixed set of routines and if we extend it (for example we need an arithmetic routine or new algorithm that the SPOX cannot offer) we still have to write it in assembly loosing the portability.

Nowadays the parallel DSP is in the focus of attention, first of all because real-world DSP problems often require immense computing power. A number of existing DSPs can be used for parallel realizations, some of them has been designed especially for parallel computing for example Texas Instrument's TMS320C40, TMS320C80 and Analog Devices ADSP21060. The task scheduling is an important part of the multiprocessor implementation of DSP algorithms. This equally means partitioning the tasks among multiple DSPs and scheduling the tasks on each DSP. Generally parallel programs are scheduled "by hand" in the existing parallel development systems which is a difficult task and in the case of more complex tasks it cannot be done effectively. The other approach used frequently in the existing DSP operating systems uses the well proven real-time operating system scheme (sometimes timesliced scheduling is added) [Spox88, Virt93]. This scheme is based on separate tasks and a task scheduler program which changes the tasks when it is necessary. This task scheduler requires processing time.

Speciality of the DSP algorithm is that they don't require much run-time decisions. Very handy description form of these algorithm is the *signal-flow graph (SFG)*. We will give later a more thorough definition, here we only mention that the signal-flow graph is a graphical description of an algorithm in which computations are represented by graph nodes and dependencies among the computations by graph branches. If we can cluster enough nodes together that their dependency graph and execution time do not depend on the input values, we can schedule in *compile time* thus eliminating the processor load of the dynamic scheduler.

An important, emerging feature of the DSP code generators whether or not they support heterogeneous target systems. This requirement arises because of the need of cost-effective design (using more expensive DSPs only if we need and replacing less-loaded ones by cheaper processors) and the new research area, the hardware-software co-design. Co-design means that the software and hardware partitioning decisions are not fixed at an early stage of development but the software and hardware design proceed in parallel interacting with each other [Kala93]. A design software supporting this technique should provide design model(s) which allow the separation of software and hardware as late as possible. This technique has very important

practical aspects now with the DSP-core libraries when the designer can use an industry-standard DSP-core (for example the TMS320C20) on his or her custom circuit.

As it is shown in the literature, more complex SFGs can be as difficult to overview or debug as a program coded in a traditional way. Formal languages are able to prove certain properties of their input programs so it is a lucrative idea to use them for checking if there are semantic errors in the SFG. The problem can be described in a formal language and the SFG can be generated by the compiler of the formal language.

Thus, the DSP code generation problem is the following: we need a system which is flexible enough to be adapted to several existing DSP platforms, avoids the power loss of the high-level languages, solves the partitioning and scheduling problems and in addition it is easy-to-use for the DSP algorithm developer who is generally not a programmer. A proposition for this problem will be presented in this document describing Rafael, an intelligent code generator based on signal flow graphs.

Rafael was designed as a small, flexible system which can run even on very small computers (it is implemented under Microsoft Windows on IBM PC compatible computers). It is a SFG compiler integrated into a simple framework which allows DSP algorithms to be described in SFG form and the compiler translates this description into program for a heterogeneous multiprocessor hardware. The compiler distributes the SFG on the multiprocessor system, schedules the operations on each processor, creates the communication scheme among the processors and generates executable assembly source program for each processor. Rafael features a programmable DSP database and code generator library so it can be adapted easily to any processor. Small resources of the host computer do not allow us to compete with the comprehensive features of existing SFG compilers hosted on workstations but we hope to prove that Rafael can compete successfully on several domains with those systems.

This document will be structured as follows :

- In the following part of Introduction we present the dataflow approach, we deal shortly with its problems in the case of graphs containing run-time decisions and we present the synchronous dataflow language concept which allows us to generate consistent dataflow graphs with simpler structure.

- In chapter 2 we will deal with the static scheduling problem, we present the schedulers used in Rafael and the Rafael schedulers will be com-

pared to other existing scheduler algorithms. As no existing scheduler could produce acceptable performance on heterogeneous architectures, a new scheduler class called Springplay has been devised. The chapter deals with this algorithm in details.

- In chapter 3 the internal structure of Rafael will be described and it will be compared to other existing SFG compilers.

1.2 The dataflow approach

1.2.1 Dataflow paradigm

The dataflow concept was proposed for its visuality which matches well to certain problems (Digital Signal Processing, for example) and for its capability to reveal the available parallelism. In dataflow, program is represented as directed graph where vertices represent computations (we will call them *operations* sometimes they are called *actors* [Lee87] in the literature) and branches represent FIFO channels that queue data values. These branches show the signal paths where a signal is simply an infinite stream of data and each data token is called a *sample*. An operation is activated by a given number of tokens on its inputs and it is *fired* (the computation assigned to the operation is executed) then it produces tokens on its outputs. These tokens may remain in the system for some time in the branch FIFOs before they are consumed by other operation.

The earliest reference to the dataflow paradigm appears to be the computation graphs of Karp and Miller [Karp66]. Each node has an associated function for computing outputs from inputs and each branch has four associated integer values :

- A_p, the number of data words initially in the queue associated with the branch,
- U_p, the number of data words written into the queue when the node connected to the input of the branch is executed,
- W_p, the number of data words removed from the queue when the node connected to the output of the branch is executed,
- T_p, a threshold giving the minimum queue length necessary for the output node to execute. We require $T_p \geq W_p$.

Karp and Miller prove that computation graphs with these properties are determinate (the sequence of data values produced by each node does not depend on the order of execution of the actors provided that the order of execution is valid). They also dealt with problems of determining the size of branch queues and the conditions that cause computations to terminate. (Later it became more important to avoid the deadlocks in the dataflow graphs so that computations can continue indefinitely). It is shown that Karp and Miller computation graph model can be analyzed in the terms of Petri nets [Buck93b].

The first papers about the usage of dataflow principles for the developement of computer architectures and programming languages were presented by Dennis [Denn75]. Dennis applied the concepts of dataflow to computer architectures thus creating the *dataflow computer architecture*. The first machine using this concept was built by Davis [Dav78]. There are two basic types of dataflow computers: *static* and *dynamic* or *tagged token* dataflow computers. The main difference between the two types is that in the static version at most one token is allowed on every branch and the storage of the edges is allocated at compile-time while in the dynamic version there is no such limitation and the storage for the branch queue is allocated dynamically. The biggest problem of the pure dataflow model is the excessive token matching and communication overhead between the operations. Hybrid models have been developed which group the operations into threads if possible and execute them sequentially [Bic91].

A data flow graph can be large or fine grain one [Lee87]. Fine grain data flow means that the operations are atomic computations like adders, multipliers while large grain data flow (also called block diagrams) are composed of more complex components like FFTs, filter blocks, etc. The granularity of the graph determines the amount of parallelism that can be exploited. We consider the basic operations undivisible and no effort is made to exploit the parallelism inside an operation.

An operation can be also a graph of operations, in this case the graph is *hierarchical*. By the most common approach hierarchical graphs are flattened (the hierarchy is destroyed) [Buck94] so that maximal parallelism be present. There are efforts, however, that the design system find the best granularity by itself [Hoan93].

1.2.2 Synchronous, boolean and integer-controlled dataflow

We call operations of the graph *synchronous* if they consume and produce constant amount of tokens when fired and the number of the tokens is known at compile time. (fig. 1.1) This restriction results in a big advantage that the control flow is completely deterministic, thus the scheduling can be accomplished in compile time. This approach can be used to describe a big number of DSP applications as they require no or little run-time decision making. As it was pointed out in [Buck93a], however, "little" is not the same as "none". Therefore, synchronous data flow (SDF) is not capable of describing important classes of problems. The traditional SDF approach was extended by conditional operations like SWITCH and SELECT (fig. 1.2). A dataflow graph containing these binary-controlled conditional operations is called *boolean dataflow graph* (BDF).

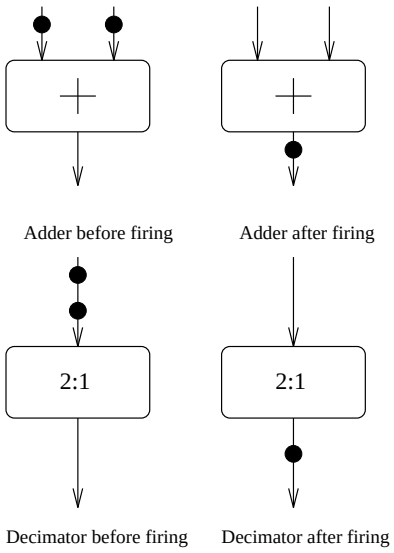
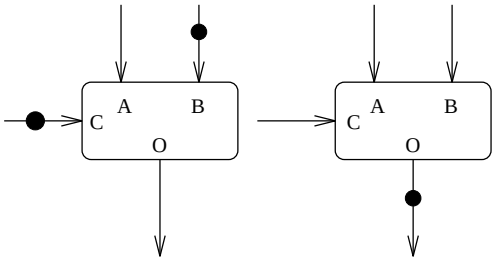


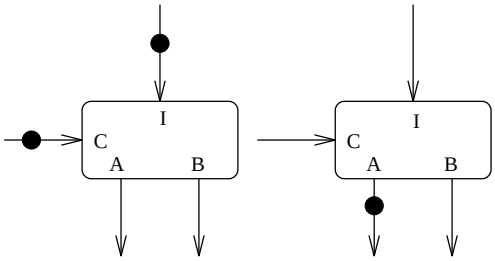
Figure 1.1: Synchronous Dataflow operations

In [Lee87] the foundations of compiling SDFs into sequential programs have been developed. The following questions can be asked about any dataflow graph :

1. Do cyclic schedules exist ? A cyclic schedule is a sequences of actor executions that return the graph to its original state.



SELECT operator before and after firing
The value of the token on C input selects whether the tokens on A or B input will be emitted



SWITCH operator before and after firing
The value of the token on C input selects whether the next token on I will be emitted on A or B outputs

Figure 1.2: Boolean Dataflow operations

2. Does the graph have bounded cyclic schedule ? The schedule length bound is important when generating code for a real-time environment.

3. Does the graph deadlock ? In deadlock situation there is no operation that can be executed.

4. Can the graph be scheduled to use bounded memory ?

A graph complying with these conditions is called *consistent*.

In [Lee87] algorithms are presented to answer all four questions for any graphs. The problem is more complicated in the BDF case.

[Buck93b] introduces an enhanced version of BDF, the Integer-controlled dataflow (IDF). BDF is extended by two new operations. The first is the CASE-ENDCASE structure which is similar to the SWITCH-SELECT pair with the distinction that the control branch consumes integer-valued tokens and CASE-ENDCASE have more outputs and inputs. The second is the REPEAT operation which repeatedly emits the input token and the number of repetitions is determined by the integer-valued token on the control branch. As the BDF model is already Turing-equivalent, these new operations do not extend the number of the algorithms that can be modelled but in several cases they simplify greatly the dataflow graph.

1.2.3 From dataflow graphs to directed acyclic precedence graphs

Dataflow graphs are not directly suitable for scheduling. The scheduler algorithm accepts a *directed acyclic precedence graph (APEG)* in which the firing precedence constraints of the operations has already been determined. As described in the previous section, the dataflow-precedence graph transformation is the central question of the dataflow graph theory.

In figure 1.3 a dataflow graph and the corresponding APEG are depicted.

In the SDF case the problem is solved in [Lee87] by means of the balance equations. By solving these equations one can determine the number of operation firings so that the tokens consumed and produced on each branch be equal. It is convenient to express these equations in matrix form. First we define the *topology matrix* which has one row for each branch and one column for each operation of the graph. A $\gamma_{i,j}$ element of this matrix Γ determines how many tokens are added by operation i to branch j. It $\gamma_{i,j}$ is negative if the operation consumes from that branch. This matrix is the following for the graph in figure 1.3.

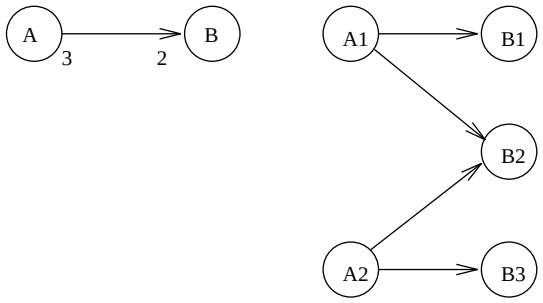


Figure 1.3: Dataflow graph and the corresponding APEG

$$\Gamma = \begin{bmatrix} 3 & -2 \end{bmatrix}$$

Now we look for the *repetition vector* $\vec{r}$ which solves the following equation:

$$\Gamma \vec{r} = \vec{0} \tag{1.1}$$

We find that all the non-trivial solutions have the form :

$$\vec{r} = k \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

which means that operation A should be executed twice while operation B three times to bring back the graph to its original state. The graph is inconsistent if equation 1.1 has only trivial solution.

If we have the repetition counts for each operations, the simplest algorithm to construct the APEG corresponding to the dataflow graph is a kind of list scheduler [Buck93b]. We maintain a "schedulable operation list" and we initialize this list at the beginning to the operations with no inputs or having sufficient initial tokens at their inputs for firing. We then add them one by one to the APEG with their data dependencies. Adding an operation to the graph may enable others and they are inserted into the schedulable operation list.

For example in the case of the graph in figure 1.3 operation A can be placed twice at once. After this step instances of operation B are allowed to be placed. As B1 consumes two only tokens of A1 so dependency branch must be added between A1 and B2 as well. B2 cannot be satisfied with the remaining one token of A1 so it will depend on A2 too. Then B3 can be

placed which consumes only from A2. More systematic algorithm can be found in [Lee87].

Whole thesises have been written on the questions of dataflow-APEG transformations in more complex cases [Buck93b, Bhat94a]. In the BDF case two important class of constructs, nested if-then-else and loops were shown to be "well-behaved" in the sense that a BDF using only these macro-constructs has always bounded memory requirement [Gao92]. In [Buck93a, Buck93b] more indulgent conditions are presented for the BDF graph to be consistent based on probabilistic analysis and clustering techniques. Probabilistic analysis is shown to produce unreliable results and the essence of clustering is to find basic constructs in the graph. As "well-behaved" graphs cover the most important algorithm constructs, accepting the restrictions in [Gao92] does not seem to cause many problems. [Buck93b] extended the probabilistic techniques to IDFs as well.

1.2.4 Partitioning and scheduling

If the dataflow graph is static (it contains no runtime decision making) operations can be assigned to processors and arranged on them statically so runtime scheduler is not needed. It is such a big advantage that hybrid solutions were presented that allows partial static scheduling even if there are runtime decisions [Buck91]. The scheduling problem is shown to be NP-complete [Sark89], heuristics are used to provide good-quality solutions in acceptable time.

Two main approaches exist for the partitioning and scheduling task as the problem is slightly different in the VLSI (called *data path synthesis*) and the programmable DSP case. In the VLSI case we are allocating specialized, low-level resources which can accomplish only one task type (adders, multipliers). In the synchronous case communication takes no time and the restriction that the duration of every operation is one control step is often acceptable. In the programmable DSP case we have much fewer resources that can accomplish many types of tasks (up to very complex operations like FFTs) and the communication must be accounted for. In this thesis we will mainly concentrate on the programmable DSP case (as the Rafael design system supports actually only these kinds of devices) but we will refer all the time to the very similar data path synthesis problems.

Major part of this thesis will be devoted to the static partitioning and scheduling problem. We already have the APEG and we have a description of resources in the target hardware. The goal is to find an assignment of op-

erations in the APEG to resources (*partitioning*) and an order of operations assigned to a resource (*scheduling*) which is optimal in some sense. In many cases we will simply use the inexact term *scheduling problem* which includes both tasks.

The target system can consists only of the same type of processing units, in this case we call our task *homogeneous scheduling problem.* If the execution time of an operations may depend on the processing unit which executes it, we face the much more complicated *heterogeneous scheduling problem.*

The heterogeneous scheduling problem has many practical aspects. First, it is a common practice to use less powerful (thus less expensive) processors and prescribe cheaper DSPs for subtasks which cannot exploit the capabilities of a bigger DSPs. Second, a very important (and fashionable) research area is the development of *co-design* methods. Co-design means that the output of a design process is a mixed hardware-software realization. An intelligent design software may decide, which part is to be realized in hardware and which in software and it has to generate circuit descriptions/programs. This design method gained considerable importance with the introduction of DSP cores - cell libraries containing entire DSPs that can be embedded into one's circuit design. Co-design results in integration of heterogeneous systems so it involves heterogeneous schedululing. Chapter 2 will deal with the scheduling problem in details presenting solutions to the less-researched heterogeneous case.

1.3 Synchronous dataflow languages

There is a big problem with the dataflow approach, the properties of the BDF graphs cannot be proven in every case. As we could see in the previous sections, static dataflow graphs can be proven quite easily but it holds true only for some classes of BDFs. Proving one's input algorithm is very important in the case of complex, time critical programs. This need inspired the creation of *synchronous dataflow languages* which are based on simple but deep mathematical principles so certain properties of the programs can be proven. The class of the systems to which these languages were developed is the *reactive system class.* We call systems reactive if they maintain permanent interaction with their environment. In addition, we use the word *real-time* if there are timing constraints defined.

The basic idea behind the synchronous approach is simple: we suppose ideal reactive systems in which internal processing takes no time. If out-

put signals are emitted at a certain time instant, they will be produced at the same time, *synchronously* with the input signals. We call a system *reactive* if it interacts continuously with its environment, it receives input events and produces output events. Note that this approach uses the "synchronous" word differently to the usage of this word by the dataflow terminology. Big advantage of this approach is that ideal synchronous systems can be decomposed into components without affecting the behaviour of the system. The first introduction of synchronous concept for software appeared in [Berr83, LeG86]. One of the best surveys of the topic is [Benv91].

There exists a set of actual machines for which the ideal synchronous model can be applied immediately. Fast processors in slow environments are handy examples for this. For example a VLSI realization with some 10 ns cycle time can give almost instantenous response. [Benv91] shows how the synchronous approach can be used to describe asynchronous systems like dataflow graphs.

There are two families of synchronous languages.

- *State based languages* like ESTEREL [Bous91]. ESTEREL modelizes state machines by instantaneous broadcast events which bring the machines into new states.

- *Multiple Clocked Recurrent Systems* (MCRSs) like SIGNAL [LeG91] and LUSTRE [Halb91]. These languages describe the state transition diagram to be modelized by means of recurrent equations. This approach is obvious for discrete time systems.

Synchronous languages provide an efficient tool for generating data flow graphs whose properties are proven.

We will introduce three dataflow languages in short here.

1.3.1 ESTEREL

The basic notion of ESTEREL (similarly to other dataflow languages) is the "signal". Signal is a stream of time instants at which a signal can be present or missing. If a signal is present, it can also have a value. All the synchronous dataflow languages share the common notion that the input signals provoke instantaneous reaction.

The reaction mechanism of ESTEREL is based on *broadcasts*. Broadcast is a signal which can be seen by every receptors in the program. Broadcasts can emit signals that can be tested for presence, value, etc. The emitted

signal is restricted to a given time instant and is received as present by all the receptors during this instant. Example :

```
   present T then emit U end
||
   emit S
||
   present S then emit T end
```

The || delimiter creates parallel tasks. Signal S is broadcasted by the emit statement, it is received by the present S which in turn emits T. Signal T is then emitted and is received by present T which emits U. At the end all the signals S,T and U are emitted at the same instant.

The most basic ESTEREL structure is the *generalized watchdog.* Its form is the following :

do
loop body
watching *signal*

This structure executes the loop body until

1. The loop body terminates
2. The signal argument to the watching arrives

Example :

```
do
  await INPUT;
  emit OUTPUT
watching TERM
```

The await statement waits for the INPUT signal. As soon as this signal arrives, the emit statement emits the OUTPUT signal. The watching structure guarantees that the body be interrupted as the TERM signal is raised. The structure can be extended by the timeout part which is executed if the loop body is terminated by the watched signal. Example :

```
do
  await INPUT;
  emit OUTPUT
watching TERM
timeout emit TIMEOUT
```

The `TIMEOUT` signal will be emitted only if `TERM` arrives before `INPUT`. Other basic structure is the `trap`. We may have :

```
trap SIG in
  await INPUT;
  emit OUTPUT;
  exit SIG
end
```

This structure waits for `INPUT`, emits `OUTPUT` then it is preempted by the `exit SIG` statement.

The ESTEREL compiler [Est88] uses effective compilation algorithms [Gonth88] to eliminate the programs with certain semantical problems, for example deadlocks, short circuits. This is done by means of a *potential function* that administrates which signals can or cannot be emitted at a certain instant. This is used for ordering the signals so that a sequential execution scheme can be generated. If this order cannot be established, casuality problem is signaled. This approach is called *execution semantics.*

Other, more exact proofing method of ESTEREL is based on the *behavioral semantics* [Plot81]. This semantic works with transitions which are described in the following general form :

$$Program \xrightarrow[Terminated]{Input/Output} NewProgram$$

This notation means the following: if the input event is *Input*, *Program* reacts by producing *Output* and goes to the new state so that at the next instant *NewProgram* will be executed. Inference rules are used to construct compound statements. The example rules for `do ... watching` taken from [Bous91] :

$$\frac{p \xrightarrow[\textbf{true}]{I/O} q}{\texttt{do}\ p\ \texttt{watching}\ S \xrightarrow[\textbf{true}]{I/O} \texttt{nothing}}$$

The rule above means that if we suppose that the body p terminates, the `watching` terminates too leaving `nothing` as residual program.

$$\frac{p \overset{I/O}{\underset{\textbf{false}}{\longrightarrow}} q}{\texttt{do } p \texttt{ watching } S \overset{I/O}{\underset{\textbf{false}}{\longrightarrow}} \texttt{present } S \texttt{ else do } q \texttt{ watching } S \texttt{ end}}$$

If the body p does not terminate, S is tested for presence at the next instant and loop body q is executed if S is not present.

The inference rules presented above are used for building *proof trees*. Inference rules are used systematically by the reasoning system for finding out properties of the program.

The ESTEREL compiler translates the program into finite state machines which can be implemented very efficiently. In the resulting program parallelism and local communications are transformed into sequential code. ESTEREL compiler is also able to produce conditioned dataflow graph that in common format with other dataflow languages. We will deal with this format in chapter 2.

Realizing the unability of the synchronous languages when handling asynhronous distributed algorithms in [Berr93] a new extension of ESTEREL is presented in which they integrate the synchronous ESTEREL with CSP [Hoar85]. CSP is an asynchronous language using classical interprocess communications like rendezvous. New CSP-like statements are :

```
channel C;
```

which declares a communication channel called C and

```
rendezvous L : C;
```

which accomplishes a rendezvous on channel C. The L parameter is an optional label. Three signals are automatically created by the `rendezvous` statement : `sL`, `L` and `kL`, `sL` is used to request the rendezvous on C, `L` is used for signaling the completion of the rendezvous while `kL` signals abandoning the rendezvous request. A given channel can accomplish only one rendezvous at a time. The referenced paper includes the new operations into the behaviorial semantics.

1.3.2 LUSTRE

As LUSTRE and SIGNAL (which will be described in section 1.3.3) are very similar, we will describe LUSTRE [Halb91] very shortly. As in the case of

every dataflow language, LUSTRE works on signals. Signals in LUSTRE are valued data streams, at a certain instant a signal can be *absent* so that it does not have value or *present* when it holds a value. We can approach it from an other direction: a signal is composed of two data streams: a *clock* which determines the instants of the signal's presence and a *data stream* which carries the values. Data in that stream can be absent. The most important restriction of LUSTRE to SIGNAL is that all the signals have one common clock.

LUSTRE is a definitional language. It means that a

```
O = I;
```

should be considered in the equational sense, not as a declarative assignment. It is the so called *substitution principle*, `O` can be substituted with `I` anywhere in the program and conversely. As a consequence, equations can be written in any order, it will not change the meaning of the program.

Types are usually imported from the "host language", the language to which the LUSTRE compiler translates. Usual operations are avilable on these types. A bit less usual operation is the `if ... then ... else` structure. An example:

```
if E < 0 then X+1 else 0
```

which means that `E` and `X` has the same clock and this will be the clock of the result as well. At the nth instant the result will have the value of $X_n + 1$ if $E_n < 0$ else 0.

The operation set is extended by "temporal" operators.

- **Delay**

  `pre` which realizes a delay. If the data stream of `I` is $(i_1, i_2, i_3, ..., i_n, ...)$ then `pre(I)` will be the sequence $(nil, i_i, i_2, ..., i_{n-1}, ...)$. The first nil denotes the value of an uninitialized memory.

- **Initialization**

  `->` (followed by). Let `I` and `F` be two signals with the same clock then `I->F` will be always equal to `F` except for the first instant when it is `I`.

- **Downsampling**

  `when`, downsampling. If `I` is a signal and `F` is a boolean signal with the same clock, `I when F` will have the clock of `F` but it will have absent value whenever `F` is false. If `F` is true, the result will be the value of `I` in that instant. See figure 1.3.2.

- **Interpolation**

  `current` interpolates the signal. `current I` has the value of `I` if it is present else it holds the last value of `I` when it was present. The statement is illustrated in figure 1.3.2.

F	true	true	false	false	true
I	x_1	x_2	x_3	x_4	x_5
I when F	x_1	x_2			x_5

Figure 1.4: The operation of when in LUSTRE

I	x_1	x_2			x_5
current I	x_1	x_2	x_2	x_2	x_5

Figure 1.5: The operation of current in LUSTRE

Assertion is a speciality of LUSTRE. One can define some known properties of the environment and it may influence program optimization and verification. We can say for example that a certain signal is never negative.

```
assert (x >= 0);
```

LUSTRE groups equations into *nodes*. A node is a part of the dataflow operations collected into one unit with parameter header and local variables. For example a FIR filter node would look like the following :

```
const a1,a2,a3:real.

node FIR(x: real) returns(y: real)
var r1,r2:real;
let
  r1 = (0. -> pre(x));
  r2 = (0. -> pre(r1));
  y = a1*x + a2*r1 + a3*r2;
tel.
```

LUSTRE compiler uses several program proofing methods, beside the common semantical analysis (number of variable definitions, absence of recursive calls, loop-free definitions) LUSTRE introduces clock calculus. This

calculus is used to prove that operator arguments have the same clock and the clock of any operands of the `current` operator is not the basic clock of the node. LUSTRE uses a simple scheme for proving clock equality, it substitutes the signal definitions when a derived signal is used in an equation then compares the resulting expressions. For example :

```
y = b+c;
a1 = 0.-> b+c;
a2 = 0.-> y;
```

After the substitution of `b+c` in the place of `y` we will have two identical equations.

The actual LUSTRE compiler produces a graph format common to ESTEREL. From this format code generators can be used to translate toward C, Lisp and ADA.

1.3.3 SIGNAL

SIGNAL language was presented in [LeG86, LeG91]. SIGNAL is based on very similar principles to LUSTRE. The main difference between SIGNAL and LUSTRE is the introduction of *missing signal state* denoted as $\perp$. This allows SIGNAL systems to be *multi-clocked* in the sense that the clock of signals can be different. Each signal is associated a clock which can be present or $\perp$ at a certain instant. Each time the clock is present, the signal has value. SIGNAL is based on only 5 kernel constructions.

- **Instantaneous function calls**

  ```
  Y := f(X1, ... , Xn)
  ```

  The basic functions defined by the host language are extended to signals. This is a *monochronous operator*, $Y, X_1, X_2, ..., X_n$ have the same clock. The function call is *instantaneous* so an Y value calculated from X values of a time index n will have the same n time index.

- **Delay**

  The delay operator is used to create index shift between its input and output signal. This operator is also monochronus, the clock of its output is the same as the clock of its input.

```
Y := X $ 2
```

will result in

$$\forall n > 1 \qquad y_n = x_{n-2}$$

The initial value of the delay can be prescribed when declaring the variable.

```
integer Y init 0
```

The behaviour of the delay operator above is shown in figure 1.3.3 (supposing 0 initial value)

X	x_1	x_2	x_3	x_4	x_5
Y	0	0	x_1	x_2	x_3

Figure 1.6: The operation of delay in SIGNAL

- **Extraction**

  Extraction operator is used very similarly to that of LUSTRE to down-sample a certain signal driven by a boolean gate signal. If `X` is any signal, `B` is a boolean-valued signal which has *true* or *false* values, the output will have the value of `X` at time instants when both `X` and `B` are present and `B` is *true*. This operator is called *polychronous* as it allows operands with different clocks.

  ```
  Y := X when B
  ```

  Time diagram 1.3.3 illustrates the operation of `when`.

X	x_1	$\perp$	x_3	x_4	x_5
B	*true*	*true*	*false*	$\perp$	*true*
Y	x_1	$\perp$	$\perp$	$\perp$	x_5

Figure 1.7: The operation of when in SIGNAL

- **Deterministic merge**

  This operator allows merging its two input signal with priority. If `A` and `B` are two signals,

  ```
  Y := A default B;
  ```

  will produce an `Y` signal which has the values of `A` whenever `A` is present and if `A` is not present but `B` is, `Y` has the value of `B`. The instant set of the resulting signal is the union of the instant set of the input signals and priority is given to `A` if both inputs are present. Figure 1.3.3 illustrates the operation of `default`.

A	a_1	$\perp$	a_3	$\perp$	a_5
B	b_1	b_2	$\perp$	$\perp$	b_5
Y	a_1	b_2	a_3	$\perp$	a_5

Figure 1.8: The operation of default in SIGNAL

  The `default` operator has an important theoretical consequence: faster clocks can be generated by means of this structure than the clock of any input.

- **Process composition**

  SIGNAL equations shown above are considered *elementary processes*. The structure

  ```
  (P | Q | R)
  ```

  allows for creating a new process from system of equations. In this new process common signal names denote common signals.

SIGNAL compiler uses a very elegant method called *clock calculus* for checking certain properties of the program and for generating the execution scheme. SIGNAL programs are transformed into boolean equations to express the presence of signals and boolean operations and dependency graphs to describe data dependencies of non-boolean functions. Integer values are assigned to signal states as the following :

$true \longrightarrow +1$

$false \longrightarrow -1$

$absent \longrightarrow 0$

$present \longrightarrow \pm 1$

Operations on these values are made modulo 3, the field consisting of the values above and the operations will be denoted $\mathcal{F}_3$. A signal is represented as x and x^2 represents its clock. With these notations `y := a+b` can be coded as the following :

$y^2 = a^2 = b^2, \quad a \xrightarrow{y^2} y \;,\; b \xrightarrow{y^2} y$

The above equations mean that y, a, b all have the same clock and $a \longrightarrow y$ and $b \longrightarrow y$ dependencies hold when the common clock of y, a, b signals y^2 is present. The description above is extended to all the SIGNAL operators [LeG91]. For example in the case of `default` :

In the case of boolean signals :

$$y = u + v(1 - u^2)$$

In the case of non-boolean signals :

$y^2 = u^2 + v^2(1 - u^2), \quad u \xrightarrow{u^2} y \;,\; v \xrightarrow{(1-u^2)v^2} y$

It means that the clock of y will be the clock of u if present, else the clock of v. y will depend on u if present else on v if present.

Systematically using the method above means will result in a conditional dependency graph and attached clock equations. The following simple equation set

```
(|  y := a + b
 |  z := y default c |)
```

has the dependency graph depicted in fig 1.9.

When SIGNAL compiler translates a SIGNAL program into sequential executable it builds a *clock tree* [Amag94]. This clock tree is constructed like the following

- We call *free* boolean signals the inputs, values resulted in the evaluation of boolean expressions and values from boolean memories. Clocks defined by the *true* value of a free boolean signal (`when C`) and clocks defined by the *false* value of that signal (`when not C`) are called *down-samplings* of that signal and inserted into the subtree of the boolean signal.

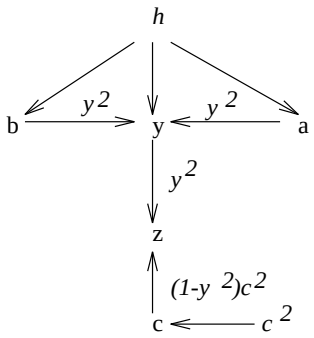


Figure 1.9: Conditional dependency graph of the example program

- If `K` lies under `H` then all instants of `K` form a subset of instants of `H`

We get this way a set of interconnected trees that we call *forest*. The forest may or may not have one common root, in the latter case the system does not have a single master clock.

Pieces of the conditional graphs are attached to this clock tree thus yielding the *conditional hierarchical graph*. The signals available at a given clock are connected to this clock in the tree and also the expressions which define these signals. The resulting graph is the base of code generation. Figure 1.10 depicts the scheme.

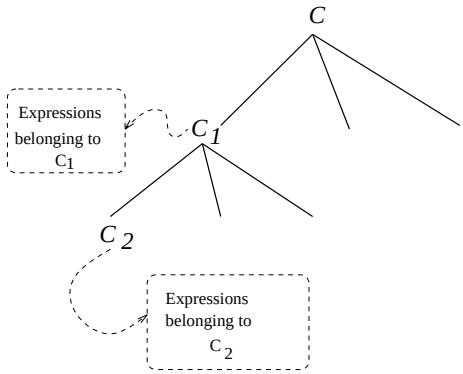


Figure 1.10: A conditional hierarchical graph

The code generation is based on nested `if ... then` constructs. Clocks are represented as boolean variables and the following structure is generated from the graph in fig. 1.10

```
if C then
  if C1 then
    ... operations belonging to C1 come here in
    dependency order ...
    if C2 then
      ... operations belonging to C2 come here
      in dependency order ...
    endif
  endif
endif
```

The SIGNAL compiler is capable of generating C and Fortran sequential code or an intermediate format for the SynDEx parallel code generator [Sor94]. The last extension of the language (SIGNALGTI, [March95]) introduces time intervals and preemptive tasks to the language.

1.3.4 SFG generation from synchronous description

We have presented three synchronous languages and we went into details how sequential program is generated from the concurrent description in the case of SIGNAL language. As we have already mentioned in section 1.2.3, the consistence of BDF graphs in general case can be difficult to prove. On the other hand, synchronous languages readily generate conditional dataflow graphs. We can exploit therefore the sophisticated proofing mechanisms of these languages and suppose much simpler dataflow graph constructs than the general BDF case. These important simplifications are the following:

- SIGNAL compiler groups the operations having the same condition together and resolves their dependency orders. We have therefore blocks of dataflow graph pieces assigned to their conditions as result of compilation.
- Biggest weakness the SIGNAL language is blamed for in [Buck93b] is that it allows only one token on every arc, token queuing is not supported. Correct SIGNAL program thus cannot describe dataflow system where an arc needs unbounded memory that must be allocated dynamically, in run-time. [Buck93b] mentions this fact as a disadvantage (context-free grammar parser that needs probably unlimited stack cannot be implemented in SIGNAL ...) but this restriction greatly simplifies the memory allocation scheme of the multiprocessor code generator attached to SIGNAL compiler.

- The condition scheme we get from SIGNAL compiler is consistent. An SFG compiler processing output of SIGNAL compiler must not deal with the problem whether the graph it transforms into multiprocessor code is consistent or not.

The approach of generating multiprocessor code from SIGNAL's output has already been used in the SynDEx environment [Sor94] and we will exploit the simplifications above when presenting the software models of the Rafael system.

Chapter 2

The static scheduling problem

2.1 Problem formalization

First we modelize both the algorithm to be scheduled and the target hardware. We use the following model for the algorithm.

- We suppose a problem which can be described by *static data flow*. The real restriction that we impose is that we know the precise dependency graph. As it was described in the introduction the creation of acyclic precedence graph from the general dataflow graph is not an evident question in the case of BDFs. We suppose this problem solved.
- The precedence relations among the tasks can be represented as a directed acyclic graph. Nodes in this graph represent operations, branches represent precedence constraints.
- We *decorate* this graph, we call the decorated version Decorated Acyclic Precedence Graph (DAPEG). Execution time vectors are attached to each node. Execution time vector contains the time necessary to accomplish a task on a certain processor.
- Branches of the precedence graph are decorated by the amount of data units transferred through them when the operation node at the input of the branch is fired.

An example DAPEG can be seen in figure 2.1.

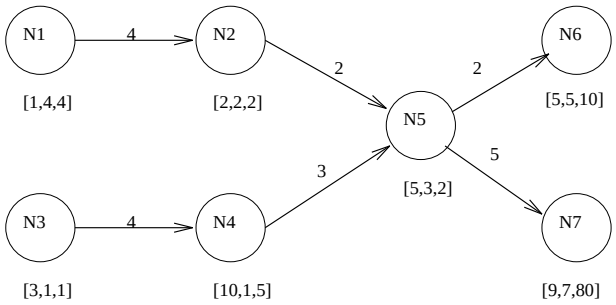


Figure 2.1: A decorated acyclic precedence graph

We will use the following notations: we have a $G(V, E)$ acyclic precedence graph where V is the set of nodes, E is the set of edges. The graph contains N nodes ($N = card(V)$) and we will denote one node as $n_i \in V$. We say that n_i is *immediate predecessor* of n_j if there exists a directed edge from n_i to n_j. In this case we will say that $n_i = prev(n_j)$ and $n_j = succ(n_i)$. We attach the computation time vector to each node :

$$\overline{t}^e_n \equiv [t^e_{n,1}, t^e_{n,2}, \ldots t^e_{n,P}], 1 \leq n \leq N$$

where P is the number of the processors in the target hardware and $t^e_{n,p}$ is the execution time t^e of node n on processor p.

The modelization of the target hardware is a harder task. The scheduling problem is just enough complex so the hardware model is often simplified in many approaches. First, most frequent simplification is that the target hardware is homogeneous so it contains the same type of processors. This restriction must be relaxed because of the importance of mixed-type realizations. Other accepted simplification area is the communication model. Realistic modelling of the wide variety of communication hardware is not an easy task. In the common DSP hardwares the following main types of communication hardwares exist :

- There is no need for communication hardware in VLSI realizations where the internal operation connections are made via wires and it takes no time to communicate thorugh them.

- Programmed communication without hardware support. Several custom-built low speed communication hardware of cheaper DSPs fall into this cathegory.

- Programmed but interrupt-supported communication hardware. Almost every cheaper DSPs (Texas TMS320C25, Motorola DSP56000) has some means of communication (serial lines, parallel interfaces) which are supported by interrupt-driven software.
- DMA-supported communication hardware in the more expensive (Texas TMS320C40, Motorola DSP96000) DSPs and in the Transputers (T800, T9000).
- Shared memory in custom DSP realizations and in the Texas TMS320C80 multi-DSP chip.
- Dedicated communication hardware (communication coprocessors) in massively parallel computers.

As we advance on the list, each form of communication is more sophisticated and require less processor overhead. Other important question is the *communication topology*. The simplest one is the totally interconnected structure which is suitable only for smaller processor numbers. Fig. 2.2 shows some of the more frequently used topologies.

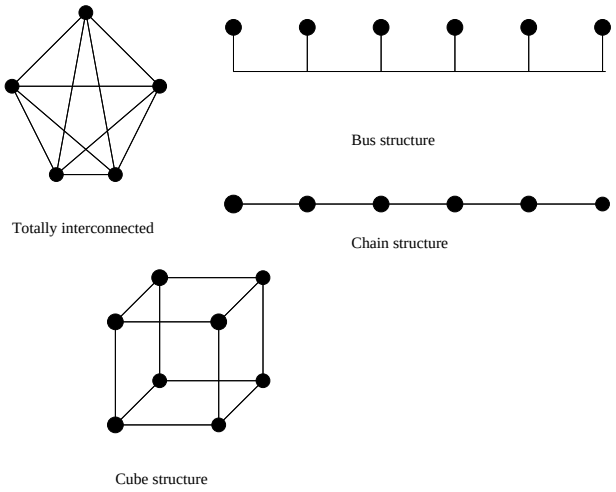


Figure 2.2: More frequently used communication topologies

The communication activities can be dynamically or statically scheduled as the operations. Dynamically scheduled communication needs a *communication layer* which works in the background. This layer processes autonomously the communication tasks, sends the messages in the background,

passes the received messages up to the computation layer, forwards messages whose destination is an other processor. This solution is easier to realize than the totally static communication where all the communication activities are scheduled together with the computations. Big drawbacks of the dynamically scheduled communication are that computation durations are more difficult to estimate (because of background tasks' processor load) and the dynamic scheduler, interrupt handler, etc. waste processor time.

A scheduling algorithm which has to consider the effect of the communication must calculate a correct estimate of the communication time necessary to pass a data block between two processors. In the case of more complex communication scheme this estimation is not an obvious task. If the communication activities use shared communication channels, the interference of the messages passing on the same channel must be considered. If the communication activities are scheduled statically as well (so there is no underlying communication layer) and the communication graph is not entirely connex (messages must travel through intermediary processors to reach their target) the schedules on the intermediary processors must also be considered.

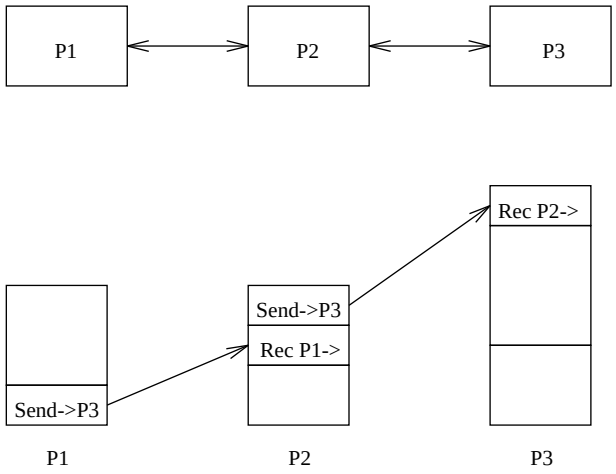


Figure 2.3: The effect of the schedule on the communication

Because of the difficulties of the exact communication modelling, simplifications are frequent. Following approaches are common :

1. Communication costs are neglected. This approach is acceptable only in the case of very coarse-grain DAPEGs or hardware realizations. In

fact, VLSI design algorithms never consider internal communication costs.

2. Uniform communication cost is supposed for every communication link. This model is acceptable in the case of totally interconnected topology with the same type of communication links.

3. Different communication cost for each communication links. This approach tries to offer an acceptable model for the effects of the message forwarding and bus occupation. Higher communication costs are given to "farther" processors (if the message must be routed through an other processor, for example).

4. Full modelling of the communication hardware including correct communication channel scheduling and operation scheduling interference.

Modern scheduler algorithms use only the 3rd and 4th models. The problem with the 3rd model is that it does not consider communication channels as resources and it may seriously underestimate the necessary communication time if there are many requesters for one channel. The 4th point is problematic because - as we will see - the scheduling problem is just enough complicated without communication channel scheduling. This results in the fact that the 4th model is employed only by simpler schedulers which produce much worse quality results than the more complicated ones while more efficient schedulers stick at the 3rd model acknowledging its inexactness. Our schedulers use the 3rd model as well but in section 2.5 we will present a version of our scheduler using the 4th model.

2.2 Static scheduling methods

In the following we give a quick overview on the existing scheduler algorithms before presenting the schedulers used in Rafael.

2.2.1 ASAP and ALAP schedules

These most basic scheduling methods are the As Soon As Possible and As Late as Possible algorithms. If we have unlimited resources, these algorithms produce minimal-length schedules but the resource utilization can be suboptimal. The algorithm was presented first in Hu's classical publication

[Hu61]. ASAP scheduler starts operations as soon as all the predecessor nodes terminate the computation that is

$$E(n_i) = max(E(pred(n_i))) + t_i \tag{2.1}$$

where t_i is the execution time of node i and $E(n_i)$ is the earliest time when n_i can be executed. Nodes with no predecessors has $E = 0$. This simple version is only for homogeneous architectures. The original version supposes unlimited resources and schedules nodes just at their E.

ALAP schedule is based on very similar principles. Nodes are scheduled as late as possible without increasing the length of the schedule.

$$L(n_i) = min(L(succ(n_i))) - t_i \tag{2.2}$$

$L(n_i)$ is the latest time when n_i can be executed in the case of minimal length schedule. L values of nodes with no successors are initialized to the maximal E value over the entire graph. Figure 2.4 depicts the ASAP and ALAP schedules of an example graph.

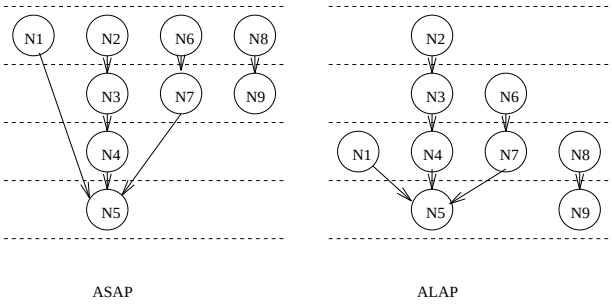


Figure 2.4: ASAP and ALAP schedules

2.2.2 Integer Linear Programming (ILP)

[Lee89a] presents an algorithm for VLSI data path synthesis which produces optimal solution. This algorithm leads back the scheduling problem to a linear optimization task. The basic version of this algorithm supposes the common restrictions in data path synthesis :

- Each operation has one cycle propagation delay
- Communication is costless

The algorithm is capable of finding minimal-length schedules but unlike ASAP and ALAP this method allocates minimal number of computation resources. The optimization task is formulated as a set of linear equations. The formulation uses the following variables :

1. M_{t_i} integer variables which denote the number of computation units of type t_i needed. (t_i can be multiplier type, comparator type, etc.)

2. $x_{i,j}$ are 0-1 integer variables. $x_{i,j} = 1$, if node i is scheduled at time step j, 0 otherwise.

We will denote the cost of a computation unit of type t_i (FU_{t_i}) as c_{t_i}, there are m types of computation units and let S be the length of the ASAP and ALAP schedules (number of control steps required).

First we make the ASAP and ALAP schedules thus determining S, ASAP and ALAP times. Now the minimization problem is the following: we want to minimize

$$\sum_{i=1}^{m} c_{t_i} \cdot M_{t_i} \tag{2.3}$$

with the following restrictions :

$$\sum_{i=1, n_i \in FU_{t_k}}^{N} x_{i,j} - M_{t_k} \leq 0, \qquad \text{for} \qquad 1 \leq j \leq S, 1 \leq k \leq m \tag{2.4}$$

Eq. 2.4 expresses that in each control step at most M_{t_k} computation unit of type t_k can be used.

$$\sum_{j=E_{n_i}}^{L_{n_i}} x_{i,j} = 1, \qquad \text{for} \qquad 1 \leq i \leq N \tag{2.5}$$

which means that only one $x_{i,j}$ variable can be 1 in the ASAP-ALAP range for each operation.

$$\sum_{j=E_{n_i}}^{L_{n_i}} j \cdot x_{i,j} - \sum_{j=E_{n_i}}^{L_{n_i}} j \cdot x_{k,j} \leq -1 \qquad \text{for all} \qquad n_i = pred(n_k) \tag{2.6}$$

Eq. 2.6 forces the precedence constraints so that each predecessor operation be scheduled before all its successors. The equation set yielded is then

solved by a linear programming package. [Lee89a] extends the method to multicycle operations, pipelined data paths and mutually exclusive operations.

Big disadvantage of the ILP method is that its computational requirement quickly becomes intractable [Gajs92]. For example if we increase the control step by 1, it means N new $x_{i,j}$ variables. There were efforts to split the problem into smaller parts and use ILP for the subproblems only [Hwang93] but ILP is still unpractical for larger problems.

2.2.3 Branch & Bound algorithms

Other well-known approach used mainly in the artificial intelligence fields is the decision tree search. If we evaluate all the possible solutions, we must find the best. See fig. 2.5 for a decision tree of a simple scheduling problem.

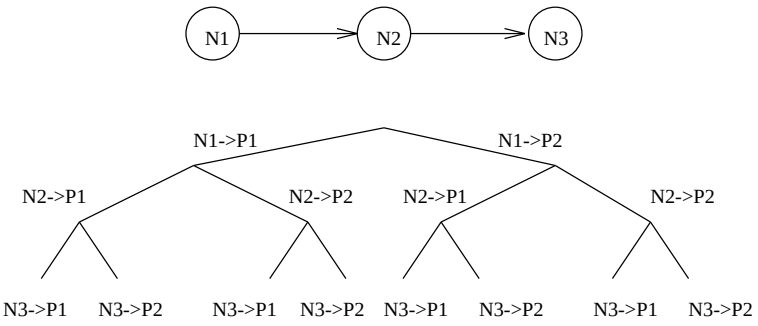


Figure 2.5: Decision tree of a simple APEG on a two-processor system

As we could easily see, the number of leaves on the tree - thus the combinations to be evaluated - is P^N. An algorithm which traverses the whole tree has exponentional complexity and it produces inacceptable execution time even in the case of very small problems. Heuristic methods are used to "purge" the decision tree. "Purging" the tree means that some combinations are eliminated based on heuristic rules. The resulting algorithm is called *Branch & Bound* class which includes broad variety of solutions. First publication of B&B for the scheduling problem appears to be [Green87].

In [Green87] two heuristic functions are presented to predict the length of the final schedule from the partial schedule. The algorithm maintains a "length of the best schedule found so far" variable. At the beginning this variable is not valid. First we start at the root of the decision tree and we make a tentative decision, for example we suppose N1 to be scheduled on P1.

We evaluate the heuristic estimation function and compare the result with the best schedule found so far (if this variable is not valid yet, we consider the result "less"). If the comparation yields "greater" result, we ignore that branch of the decision tree and make a new tentative decision. If the result of comparation is "less", we advance on that branch of the decision tree and we schedule a new node. If we got to a final schedule and its length is less than the best schedule, we record the best schedule and its length and step back to the previous level of the decision tree. Fig. 2.6 illustrates the method.

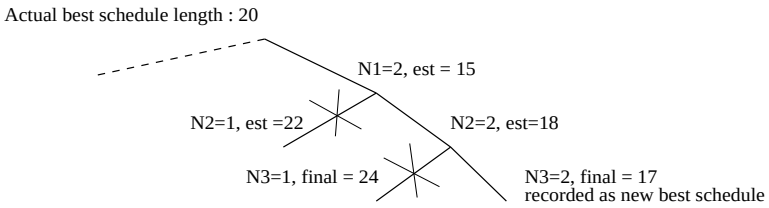


Figure 2.6: Purging the decision tree in the Branch & Bound algorithm

The heuristic function has critical role in the utility of the B&B algorithm. If it underestimates too much the schedule length, too few branches will be purged and the execution time will be inacceptable. If it overestimates, however, branches leading to the best solutions may be cut.

[Green87] introduces three heuristic functions called OPT,H1 and H2. OPT produces smaller estimates than H1 and H2. The OPT function generates the largest number of nodes. Let f_i be the time when processor i becomes free (finish time of the last operation scheduled on it) and

$$W = \sum min(\vec{t}_n^{\,e}) \qquad \text{for all the operations not scheduled so far} \qquad (2.7)$$

With these notations :

$$OPT = max(f_i) + max(0, \sum_{j=1}^{P} \frac{W - f_j}{P}) \qquad (2.8)$$

[Green87] claims that OPT never overestimates the final completion time. It is not true, see fig. 2.7 for an example. Other simplified function which generates less nodes is called H2. H2 assumes that tasks not in the partial schedule will have the same average execution time as the tasks in the partial schedule.

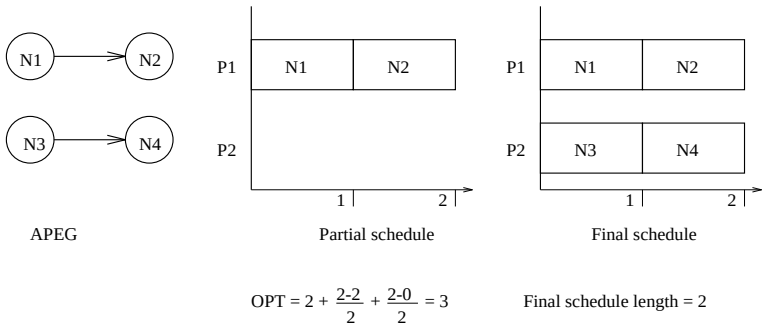


Figure 2.7: A case when OPT overestimates the final schedule length

$$H2 = N \cdot \frac{max(f_i)}{\text{tasks in the partial schedule}} \tag{2.9}$$

The referenced article uses H2 only after 20% of the tasks has been assigned because the value of H2 varies widely at the beginning. Communication costs are not taken into account in this approach.

There were many efforts concerning the usage of the B&B style algorithms for solving the static scheduling task. [Chow91] presents the well-known A^* algorithm in static scheduling application and [Konst90] describes a B&B-style algorithm which is able to consider communication cost as well. Advantage of the B&B methods is that it is capable of supporting heterogeneous architectures without difficulty. The biggest disadvantage is that in spite of the decision tree cuts, B&B still has exponential complexity in average so its evaluation can be very costly. We will see that algorithms in polynomial complexity can produce as good results as B&B methods in exponential time.

2.2.4 General List Schedulers

By far the simplest and most popular method is the list scheduler algorithm. Huge number of applications has been published based on list schedulers. The basic idea is very simple. We have a *ready node list* which contains all the nodes that can be executed at the moment. First it is initialized to nodes with no predecessors. Heuristic functions are used to pick a *candidate node* among the nodes in the ready list and choose a processor for it. This node is scheduled on the processor chosen, removed from the ready list and the successor nodes whose all predecessors have been assigned are inserted

into the ready list. The algorithm terminates when the ready list becomes empty.

The interesting part here is the heuristic which selects the candidate node and/or the appropriate processor for it. The simplest approach is based on ASAP level: nodes with smaller ASAP levels are scheduled first [Tseng86]. More advantageous approach is to delay the nodes as late as it does not block the execution of other nodes, this is the ALAP-based scheduling, nodes with the lowest ALAP levels are scheduled first [Kung85]. Combination of the two approaches is the mobility or freedom-based scheduling. The mobility is the difference between the ALAP and ASAP time, operations on the critical path has 0 mobility. An operation belongs to the critical path if it cannot be delayed without increasing the length of the schedule. In the mobility-based schedulers [Pang87, Goos87, Mirch88] nodes with lower mobility will be scheduled first.

We will examine here a little bit more in detail an ALAP-based scheduler which is used in the SynDEx system [Sor94] as we will use both the system and the scheduler as reference.

SynDEx scheduler is an ALAP-based scheduler whose cost function is :

$$J_{n_i,p} = t_p^{start} - t_{alap} \tag{2.10}$$

where t_p^{start} is the earliest time when the execution of node i can be started on processor p taking into account the finish time of the operations on that processor and the time necessary to pass each variable needed by the operation from other processors.

The node-processor selection part is rather complicated.

1. First the best processor is found for every ready node. The best processor is the one on which the the node achieves the minimal start time.

2. We look for the node with the best (earliest) start time and we will denote it *earliestcandidate*. We define the limit date variable as the following: *limit date = start time of earliestcandidate + duration of earliestcandidate*.

3. We evaluate the cost function for each ready node whose start time is smaller than the limit date. We pick the node with the highest cost (most urgent to execute) and we schedule it on its best processor.

We will evaluate this method in section 2.3.

In their very influential and frequently referenced article Paulin and Knight [Paul89] introduced a sophisticated heuristic function using "forces" to describe how necessary it would be to assign an operation to a functional unit. The force calculation considers the effect of the preceeding assignments and the effect of the assignment on successor nodes as well. Advantage of the method is that it paralelly considers the effect of scheduling of all the nodes and also their successors.

One of the most advanced list scheduler (practically the state of the art) was introduced in [Sih93a], the DLS scheduler. DLS uses a very complicated heuristic function in which

1. the communication time
2. heterogeneous target system
3. descendant consideration
4. resource scarcity (how important it is that a certain node obtain a certain processor)

are taken into account. We will use DLS as a reference algorithm for performance evaluations of our algorithms.

2.2.5 Graph partitioning algorithms

The rather fuzzy class name in the section title denotes algorithms which cut the graph into pieces first using heuristic methods then schedule these partitions on processors. These algorithms are the most complicated heuristic schedulers and on homogeneous architectures these methods produce the best results.

The linear clustering technique groups the most expensive paths into linear clusters [Kim88]. Linear cluster is a degenerate tree in which every node has exactly one predecessor and successor. In each iteration the most expensive path (both in communication and in computation) is grouped into a linear cluster and nodes belonging to this path are removed from the graph. The resulting clusters are then mapped to the processor architecture using graph-theoretic techniques.

The internalization technique [Sark89] clusters nodes together in order to minimize the schedule length on an unbounded number of processors. First each node is put into different clusters then the algorithm tries to unify clusters. Two clusters can be merged if there are communication between

them, the algorith tries to "internalize" the communication between clusters. A clustering step is accepted if it does not increase the length of the schedule else the clusters remain unmerged and the next arc is considered. The schedule lenght estimation algorithm is similar to critical path methods. First the earliest and latest start time of the nodes are calculated similarly to the ASAP-ALAP method taking into account the communication times. The main difference is that now we enforce that nodes in the same cluster be scheduled on the same processor. We suppose unbounded number of processors in this pass. When the clustering phase finishes, a modified list scheduler is used to assign clusters to processors. This list scheduler tries to assign each unassigned cluster on each processor and maps the cluster to the processor which yields the minimum execution time.

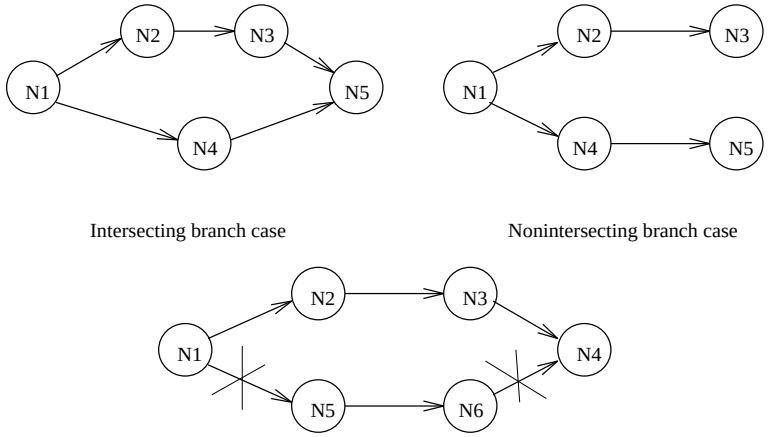


Figure 2.8: NBranch and IBranch instances and arc cuts

In [Sih93b] a complex method is presented which produces even better clusters than the previous algorithms. We describe the method only briefly here, interested reader should consult the referenced article.

- First the *reachability set* is determined for each node. If node j is in the reachability set of node i it means that precedence constraint (direct or indirect) exists between the two nodes, node i must be executed before node j. Then branches of the graph are categorized to intersecting and nonintersecting branches. A branch is intersecting if $RS(B) \cap RS(C)$ is nonempty (where $RS(B), RS(C)$ are the reachability sets of the branch nodes), nonintersecting else.

- The branch instances are inspected whether it is worth cutting down one branch. We verify whether we gain by parallelism or loose by communication cost by scheduling the parallelism instance on an other processor. If we gain, the branch is sliced down (are sliced down in the Ibranch case) and is registered as a new cluster.
- The clusters are grouped hierarchically. At each step the clusters which communicate the most are selected and merged.
- Declustering. We take the two top clusters in the cluster hierarchy and shift some of them to candidate processors selected according to communication costs and we verify, if we yield better schedule than the actual one and save this schedule if it is better. Then we move to the next cluster pair in the hierarchy tree.

As we could see, all the schedulers in this section depend heavily on the fact that speedup can be yielded only by maximizing parallelism exploitation and minimizing communication load. Neither of them is able deal with heterogeneous architectures.

2.3 The Rafael heterogeneous list scheduler (RHLS)

The first Rafael scheduler was developed for really rapid prototyping, for this reason we chose a list scheduler. Experiences got with RHLS shown us what we can expect from a list scheduler. RHLS is an ALAP-based scheduler which was made suitable for heterogeneous environment.

In the first step we create ASAP and ALAP schedules in order to get the ALAP levels. (See section 2.2.1). We assume that we can always schedule the nodes on the fastest processor possible so minimum execution time is supposed when building the ASAP-ALAP schedules. Then we launch the list scheduler as it was described in section 2.2.4.

$$t_n^{e,asap} = min(\overline{t}_n^e)$$

Then we define *urgency* of the operation n like the following :

$$u_{n_i} = L_{n_i} - t_v \tag{2.11}$$

where t_v is the *virtual time* and it will be detailed later.

The base of the scheduling heuristic is to assign the nodes on the critical path to the fastest processor available. The more urgent it is to execute a

node (as its delaying would set back the execution of the whole graph) the faster processor it deserves. The most urgent nodes are those which have the lowest ALAP time.

We pick hence the node to be scheduled, we need the best processor to execute it. The best processor selection is very simple: we try the node on each processor considering the communication costs and we pick the one on which the node achieves the earliest completion time. Before trying a node on a processor, necessary communication activities are scheduled tentatively so that we know how much time must be calculated for fetching the input variables produced on other processors.

The heuristic algorithm works like the following :

```
Create the ready node list from nodes that has no
   predecessors
while the ready list is not empty do
  for all nodes do
      if u(i) < minimum so far
         Candidate = node i;
  end for
  Try the candidate on each processor
     considering communication cost;
  Choose the processor on which the task achieves
     the earliest ending time;
  Schedule candidate node and the necessary
     communication activities on candidate
     processor;
  Update u(i)s and tv;
  Add nodes that become ready to the ready list;
end while
```

As the real t_{n_i} node starting times will generally not be equal to the ideal ASAP or ALAP starting times the scheduler maintains *real processor times* and t_v *virtual time*. The virtual time is used to track the time in the ALAP schedule graph while the real time is the scheduling time on the processors. The t_v variable shows where we are in the ALAP schedule graph, it is set to the lowest ALAP time among the ready nodes. The last step is the updating of urgency and virtual time variables.

We evaluated RHLS with two reference algorithms: SynDEx and DLS (see section 2.2.4). The prototypes of the three algorithms were realized

alg/RHLS results(%)	worst	best	average
DLS	-21.95	6.96	-5.7
SYN	-10.10	7.24	-1.92

Tableau 2.1: Performance comparation of RHLS with the reference algorithms

in Lisp and a 21-member set of DAPEGs was generated randomly. The execution time of the nodes was in the 1-10 interval, the number of the nodes was between 25 and 50, the set was homogeneous as SynDEx could not support heterogeneous architectures. Communication times were not generated randomly, we used a chain-like model where farther processors had bigger communication cost. The graphs were scheduled by each algorithm and the following ratio :

$$\frac{t_{alg}^{sched}}{t_{RHLS}^{sched}} 100\% - 100\%$$

was calculated wher t_{alg}^{sched} is the length of the schedule produced by the reference algorithm and t_{RHLS}^{sched} is the length of the schedule produced by RHLS. The worst, best and average performance ratios can be seen in table 2.1 and the histograms of the performance ratio distribution is depicted in figures 2.9 and 2.10.

As we can see the more complicated the heuristic rule is more cases there are when it fits well the problem. The complexity of the heuristic rule , however, is not proportional with the performance gain. The very complex DLS heuristic rule gains only about 6% over RHLS and less than 4% over SynDEx in average and there are numerous cases when the simpler algorithms produce better schedules. This disappointing experience diverted us from the list schedulers and urged us to develop new scheduler methods.

2.4 Nonpipelined Springplay algorithm

The heterogeneous environment adds a new dimension of liberty to the scheduling problem so the complexity of the problem increases significantly. The - even partial - search of the decision tree or a formal ILP solution are proved to produce inacceptable execution times. Inspired by the succes

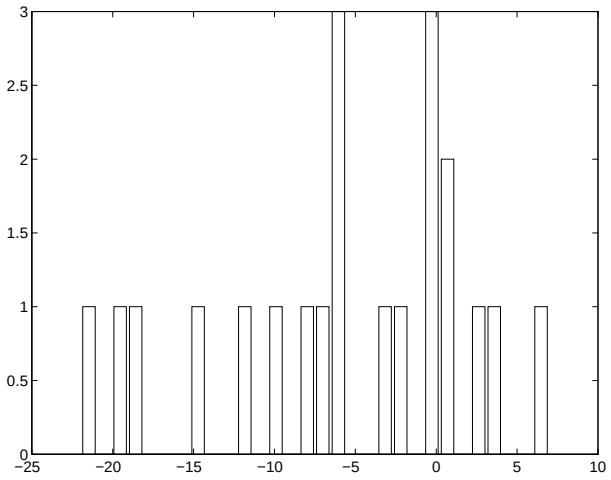


Figure 2.9: Distribution of DLS/RHLS results, 21 samples

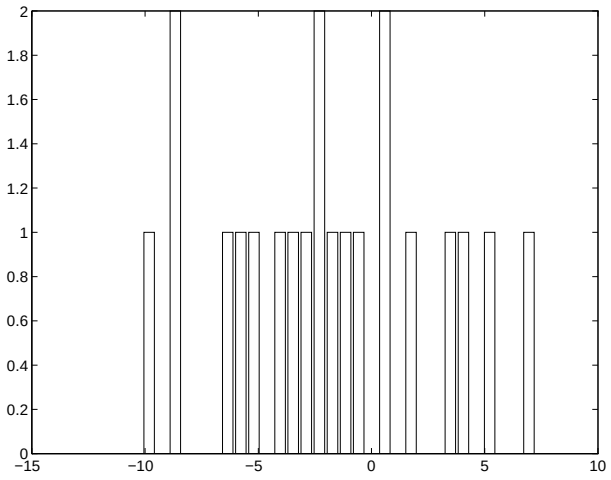


Figure 2.10: Distribution of SYN/RHLS results, 21 samples

of Hopfield class of neural networks in solving optimization tasks [Hopf85, Pars87, Mitt93, Rabe93] and by a very graphic representation of affectations in the Force-Directed Scheduling algorithm [Paul89] we devised a new class of heuristic scheduler that we call *global heuristic optimizer*. In this method no decision is made but the heuristic rule is used to affect the state of the resolver system. This system is constructed in such a way that it converges toward the optimal solution. Springplay algorithm is the first application of this global optimizer idea [Pall95].

2.4.1 Principles of the Springplay algorithm

In Springplay there is no decision making according to heuristic rules. Each node has a *state* which is an analog quantity. For clarity, we have chosen a geometric description form so node states are expressed as coordinates in a $P-1$ dimension coordinate system. The state of node i will be denoted by a $P-1$ dimensional vector $\vec{V}_n, 1 \leq n \leq N$. The processors are represented by similar points as well, we call points representing processors *fixpoints* denoted by $\vec{FP}_p$ where p is the processor number. The fixpoints are arranged in such a way that the distance between each point is 1, it is always possible to find P such points in a $P-1$ dimension coordinate system. The strongness of membership of a node to a certain processor is measured by the distance between the node state and the fixpoint of that processor. So the distance between node n and processor p is :

$$d_{n,p} = \mid \vec{V}_n - \vec{FP}_p \mid$$

We use the following distance definition :

$$\mid \vec{a} - \vec{b} \mid = \sqrt{\sum_{i=1}^{P-1} (a_i - b_i)^2}$$

where a_i, b_i are the ith coordinates of vectors a, b, respectively. We call the processor whose fixpoint is the closest to the state of the nth node ($d_{n,p} = min(d_{n,p}), 1 \leq p \leq P$) *principal processor of node* n and it is denoted as p_n. Node states are influenced by *forces* which are created in such a way that they push and pull the node states to places which represent minimal-length schedule. This force system will be detailed later. The resulting force affecting node n is denoted $\vec{F}_n$ its component pointing to processor p will be referenced as $\vec{F}_{n,p}$. Figure 2.11 shows the arrangement for 3 processors and one node.

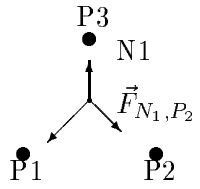


Figure 2.11: Springplay arrangement of fixpoints, node state and forces

At the beginning all node states are initialized to the "center point" so that $\forall d_{n,i} = d_{n,j}, 1 \leq i,j \leq P, i \neq j$ "naive schedule" is generated using a simple list scheduler. This schedule puts all the nodes to the first processor.

The algorithm maintains an actual schedule which is generated so that the nodes be put on their principal processor. Actual schedule is made by a simple list scheduler detailed later, at the beginning it is initialized to the naive schedule.

After this the node state modification phase starts. The algorithm takes the nodes one by one, calculates the actual value of the force exerted then the node state is updated according to

$$\Delta\vec{V}_n = \vec{F}_n \Delta t \tag{2.12}$$

$$\vec{V}_n = \vec{V}_n + \Delta\vec{V}_n \tag{2.13}$$

If the principal processor has changed because of the node state modification, the list scheduler is invoked. The modification of the principal processor changes the execution time of the node, the communication time and the parallelism.

The list scheduler used in Springplay uses the actual principal processors to determine where the nodes should be scheduled and higher priority is given to nodes which can be started earlier.

$$SL = max(\text{finish time of all the predecessor nodes}) \tag{2.14}$$

The list scheduler simply gets the node with the lowest SL value (one is chosen arbitrarily if there is more of them) and schedules it on its principal processor. No communication activities are scheduled this time.

When one pass of node state modification is finishes - all the node states has been modified - Δt is normalized according to Eq. 2.15.

$$\Delta t = \frac{0.2}{M} \tag{2.15}$$

$$M = fact_1 \cdot M + fact_2 \cdot \mid \vec{F}_n \mid \tag{2.16}$$

Eq. 2.16 expresses that Δt is adapted to the absolute values of the forces calculated during the convergence. Bigger the forces are, smaller Δt is to avoid instability. Exponential averaging is applied to the force absolute values. This solution has two reasons.

1. The absolute value of the force in the last iteration cannot be used as it introduces a too strong feedback in the system. This solutions showed to cause oscillations in Δt so in the convergence process as well.

2. At the beginning extremely big forces can be present mostly if the force coefficients are big. Later, approaching to a solution the forces become much smaller. Appropriate stepsize is required in the two cases.

During the experiments Springplay produced the best solutions with $fact_1 = 0.9, fact_2 = 0.1$ and $fact_1 = 0.99, fact_2 = 0.01$ values depending on the graph.

The node state modification continues until any of the stopping condition is satisfied. As no stability property has been proved for the algorithm, we use strict stopping criteria at the beginning which is changed to more indulgent ones later and after a certain number of iterations the modification is stopped. As we will see, the length of the schedule can *grow* during the convergence. If the convergence process is not stopped after a $1.5 \cdot N$ iterations we start to make *final schedules* and the best final schedule produced during N iterations that follows is picked. The final schedule is the same list schedule as above but it schedules the communication activities as well. Recently send communication activities are inserted just after the node which generates the output to be sent and receive operations are put just before the nodes which need the value. This time we do not consider the possible hardware support of parallel computation and communication. This simple scheme will be refined in the future. The definition of SL (Eq. 2.14) is modified so that it considers now the communication time.

$$FSL = \text{earliest time when all the inputs are available} \tag{2.17}$$

If the convergence is stopped before reaching $1.5 \cdot N$ iterations, final schedule is made at the end else the best schedule found during the second N iteration

is the output of the algorithm. The pseudocode of the entire algorithm can be found below.

```
procedure SpringPlay
begin
  Initialize all node principal processors
    to the 1st processor;
  Initialize all node states to the
    center point;
  ActualSchedule = List Schedule;
  BestSched = not valid;
  Limit1 := 1.5 * number of nodes;
  Limit2 := 2.5 * number of nodes;
  dt := 0.001;
  maxdF := 0;
  ActLimit := Limit1;
  IterationWithoutChanges := 0;
  IterationCount := 0;
  do
    ChangeList := empty list
    for all nodes do
      F:= Calculate force;
      dV := F * dt;
      Modify node state;
      if principal processor changed then
        ActualSchedule = List Schedule;
        add changing to ChangeList;
      endif
      M = M * 0.9 + F * 0.1;
    endfor
    Update dt according to M;
    Increment IterationCount;
    if ChangeList == empty list then
      Increment IterationWithoutChanges;
    else
      IterationWithoutChanges := 0;
    endif;
    if IterationCount <= Limit1
      ActLimit := 5;
```

```
      elseif IterationCount <= Limit2
        ActLimit := 1;
      else
        ActLimit := 0;
      endif;
      if IterationCount > Limit1
        ActFinalSchedule = FinalSchedule;
        if ActFinalSchedule is better than BestSchedule
          BestSchedule = ActFinalSchedule;
        endif
      endif
    until ( IterationWithoutChanges > ActLimit );
    if BestSchedule is not valid
      BestSchedule = FinalSchedule;
    endif
/* Output of the algorithm */
    return BestSchedule;
  end;
```

The force system consists of 4 different components, each represents a certain property of the schedule.

$$\vec{F}_n = \vec{F}_n^T + \vec{F}_n^C + \vec{F}_n^P + \vec{F}_n^A \tag{2.18}$$

In the following sections each of these force components will be detailed.

2.4.2 Execution time minimizing component

This component introduces a preference that the principal processor of a node be the one on which the node achieves the shortest execution time. For this purpose forces are created toward each fixpoint which mean "how strongly" the node would like to be on that processor. We define now a shorthand notation for the unit vector pointing from the node state to a processor fixpoint.

$$\vec{e}_{n,p} = \begin{cases} 0 & \text{if } \mid \vec{FP}_p - \vec{V}_n \mid = 0 \\ \dfrac{(\vec{FP}_p - \vec{V}_n)}{\mid \vec{FP}_p - \vec{V}_n \mid} & \text{else} \end{cases}$$

The force affecting node state n from processor p is :

$$\vec{F}^T_{n,p} = D^T_{n,p}\vec{e}_{n,p} \tag{2.19}$$

where $D^T_{n,p}$ is the coefficient and it is multiplied by the unit vector toward the processor.

$$D^T_{n,p} = \sum_{i=1, i\neq p}^{P} t^e_{n,i} \tag{2.20}$$

The $D^T_{n,p}$ coefficient is defined in a bit complicated way so that it have time dimension. It is the sum of all the components in the execution time vector but the execution time on the processor for which it is calculated which gives the result that the coefficient of the processor on which the node achieves the smallest execution time will be the biggest. This will try to pull the node state to that processor. The final force from this component is the sum of all forces with the same n index.

$$\vec{F}^T_n = \sum_{p=1}^{P} \vec{F}^T_{n,p} \tag{2.21}$$

2.4.3 Communication cost component

This term adds forces which try to influence the actual schedule so that the communication cost be minimal. Let C_n denote the number of nodes connected - both inputs and outputs - to node n and $N_{n,i}$ will stand for the node with connection number i connected to node n. We will use $C_{n,i}$ to refer to the ith connection of node n. We will use the following shorthand notation : $T_{n,p,i}$ is the time required for communication between processor p if node n is scheduled on it and the principal processor of node $N_{n,i}$. If $N_{n,i}$ consumes the result of node n $T_{n,i}$ is the time of the send activity on p_n, else it is equal to the time necessary for receiving the output of $N_{n,i}$ from its principal processor. An example for explaining the notations can be seen in figure 2.12.

The communication forces are defined similarly to F^T. For each connection of node n forces are added toward each processor which symbolize how necessary it would be to put the node onto that processor because of the communication cost with its neighbours. $F^C_{n,C_{n,i},p}$ is the force caused by connection $C_{n,i}$ toward processor p.

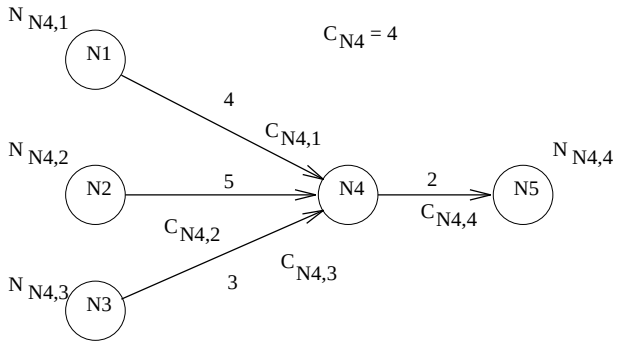


Figure 2.12: Communication cost notations

$$\vec{F}^{C}_{n,C_{n,i},p} = D^{C}_{n,C_{n,i},p}\vec{e}_{n,p} \tag{2.22}$$

$$D^{C}_{n,C_{n,i},p} = \sum_{j=1,j\neq p}^{P} T_{n,j,i} \tag{2.23}$$

$$\vec{F}^{C}_{n} = \sum_{i=1}^{C_n}\sum_{p=1}^{P} \vec{F}^{C}_{n,C_{n,i},p} \tag{2.24}$$

The meaning of $D^{C}_{n,C_{n,i},p}$ in Eq. 2.23 is similar to the constant defined in Eq. 2.20, its value is bigger if the communication cost to processor p is smaller.

2.4.4 Parallelism optimization

The two previous terms assure that nodes tend to be placed on processors on which they achieve the lowest execution time while minimizing communication costs. To get Springplay to strive toward solutions where the total execution time is minimized by exploiting parallelism in the algorithm to be scheduled we introduce a new term. First we define our notion of parallelism of node n. In our definition parallelism is the sum of the occupied time of all the other processors in the time frame of node n. This value is used to express if there is a possible amelioration of the actual schedule by exploiting the possible parallelism better. Figure 2.13 illustrates the definition. In the following we will denote this parallelism quantity calculated for node n in a schedule S as $t^{par}_{n,S}$. Moreover, we define the amount of parallelism of node

n with respect to processor p as the occupied time on processor p in the time frame of node n and we will denote this quantity as $t^{par}_{n,p,S}$. For example $t^{par}_{N1,P3,S} = 2$ in Figure 2.13.

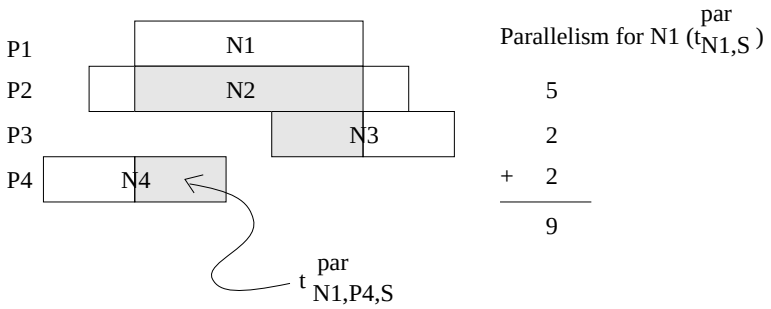


Figure 2.13: Measuring parallelism in the graph.

The force calculation is based on the "ideal parallelism" measured in ASAP schedule. The algorithm makes an ASAP schedule first using the $t^{asap}_n = min(\overline{t}^e_n)$ as node execution time then calculates the parallelism in this schedule. Ideal parallelism is defined as the following :

$$t^{ipar}_n = \begin{cases} t^{par}_{n,ASAP} & \text{if } t^{par}_{n,ASAP} \leq P \cdot t^{asap}_n \\ P \cdot t^{asap}_n & \text{otherwise} \end{cases} \tag{2.25}$$

Eq. 2.25 says that the ideal parallelism is measured in the ASAP schedule but limited to the amount of parallelism that can be exploited in the target hardware. This limitation is imposed by the number of processors (P) available.

When calculating F^P_n force the parallelism in the actual schedule is calculated and compared with t^{ipar}_n. If the actual parallelism is bigger than a givent percent (called *exploitation factor*, η_e) of the ideal one, we strengthen the node's membership to its principal processor, else we introduce forces which pull the node state to other processors. The value of the exploitation factor determines, how good parallelism exploitation will be accepted. The closer it is to 1, the better schedules the algorithm produce but its stability deteriorate. We found that $\eta_e = 0.9$ is a good compromise between stability

and schedule quality.

$$\vec{F}_n^p = \begin{cases} (t_{n,ACT}^{par} - t_n^{ipar})\vec{e}_{n,p_n} & \text{if } t_{n,ACT}^{par} > \eta_e t_n^{ipar} \\ \sum_{p=1}^{P}(t_n^{ipar} - t_{n,ACT}^{par})\vec{e}_{n,p}(\frac{t_{n,p}^e - t_{n,p,ACT}^{lpar}}{t_{n,p}^e}) & \text{otherwise} \end{cases} \tag{2.26}$$

$$t_{n,p,ACT}^{lpar} = \begin{cases} t_{n,p,ACT}^{par} & \text{if } t_{n,p,ACT}^{par} \leq t_{n,p}^e \\ t_{n,p}^e & \text{else} \end{cases} \tag{2.27}$$

Eq. 2.26 expresses that depending on the parallelism exploited the node state is pulled to p_n or pushed to other processors. This latter force depends on how much time is available in the given time window on other processors, it is 0 if the time window has already been filled by other tasks and bigger if there is unoccupied time on that processor. In Eq. 2.27 we limit the time we demand on other processors to the execution time of node n on that processor.

2.4.5 Anchoring nodes

Race situations can exist in some cases when the schedule alternatives are equally good. Let us imagine for example one task and 2 processors and equal task execution times on both processors, in this case we are free to schedule the task on any processor (Fig. 2.14). Race situations cause instability in Springplay. To prevent this, F^A component is added which introduces slight preference toward the principal processor. The amount of this preference is controlled by the *anchoring factor* (η_a).

$$\vec{F}_n^A = \vec{e}_{n,p}(\eta_a min(\overline{t}_n^e)) \tag{2.28}$$

During the experiments $\eta_a = 0.1$ gave good stabilizing effect without deteriorating the quality of the final schedule.

2.4.6 Fixpoint generation

Generating fixpoints needs solving a simple geometrical problem. Given a $P-1$ dimension space, we want P points so that the distance between each pair is 1. The following simple algorithm can generate the solution from the $P-1$ point case.

Let us denote the coordinates as elements of a $P-1$ dimension vector : $[c_1, c_2..., c_{P-1}]$ and $c_{i,j}$ will mean the ith coordinate of the jth point.

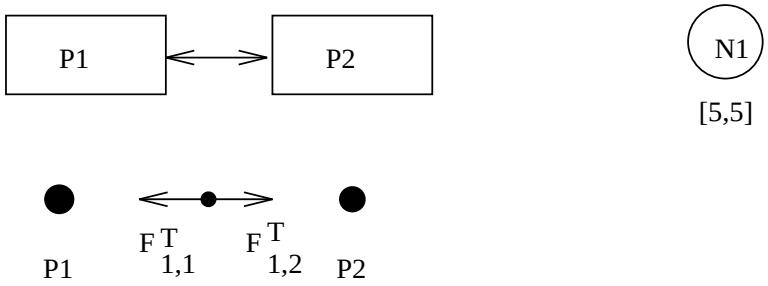


Figure 2.14: Race situation in a DAPEG

When we generate the new point we exploit the fact that the $P-1$ point problem could be solved in $P-2$ dimension so the $P-1$th coordinate of the points coming from the $P-1$ point solution is necessarily 0. Now we have to calculate a Pth point whose distance from all the old points is 1. This is done by generating a set of well-known distance equations for the new point.

$$(c_{1,P}-c_{1,p})^2+(c_{2,P}-c_{2,p})^2+...+(c_{P-1,P}-c_{P-1,p})^2 = 1, 1 \leq p \leq P-1 \quad (2.29)$$

Note that $c_{P-1,p} \equiv 0$. Systematically subtracting the $2 \leq p \leq P-1$ members of this equation set from the $p = 1$ element we get

$$\frac{\sum_{i=1}^{P-2}(c_{i,p}^2 - c_{i,1}^2)}{2} = \sum_{i=1}^{P-2}(c_{i,p} - c_{i,1})c_{i,P}, 2 \leq p \leq P-1 \quad (2.30)$$

which gives us $P-2$ linear equations for $c_{i,P}, 1 \leq i \leq P-2$. The missing $c_{P-1,P}$ coordinate can be yielded by substituting the known $c_{i,P}$ components into Eq. 2.29.

2.4.7 Complexity

The complexity of Springplay is $O(N^3 + N^2P)$ and it is derived as follows. Each force component requires $O(P)$ operations (weighted sums from 1 to P). Each iteration requires the evaluation of all the force components for each nodes so one iteration costs $O(NP)$. The maximum number of iterations in the current prototype is $2.5 \cdot N$ so the maximum number of force computations is $O(N^2P)$. After each node state modification there is a possibility that the list scheduler is invoked which is an $O(N)$ algorithm. In worst case the list scheduler is invoked after each node state modification.

As there are $2.5 \cdot N^2$ possible node state modifications this component can have $O(N^3)$ complexity. So the total complexity is $O(N^2P + N^3)$.

2.4.8 Performance evaluation

We made comparative tests between Springplay and two other algorithms: a Branch&Bound type algorithm described in [Green87] and the DLS algorithm [Sih93a] which is a Generalized List Scheduler class method. We chose these two ones because of their support of heterogeneous architectures which seems to be a less-investigated topic according to our bibliography research. Further problem was that the system model of these two algorithm was different to ours regarding the cost of communication : [Green87] considers no communication cost while [Sih93a] supposes dedicated communication hardware. We forced our - more realistic - system model to these algorithms which caused runtime problems to the B&B method ameliorating the quality of the solutions it produced at the same time. In the case of the DLS the differences of the system models have less importance. In the following we use BBOPT and BBH2 to refer to the Branch&Bound algorithm with two types of heuristic functions in [Green87] and DLS to denote the method presented in [Sih93a].

The prototypes of the algorithms were realized in LISP. To counterbalance the slow execution speed of Lisp we allowed a generous amount of runtime before terminating the computation. This caused problems only for the B&B type algorithms, the exponential growth of computation requirement resulted in inacceptable run times over 25 nodes. For this reason more graphs were generated with smaller node numbers and less processors.

The algorithms were tested with 80 randomly generated data-flow graphs where the number of the nodes was between 16 and 50 and the node execution times were in the [1,11] interval,the number of the processors was 4 and 5. Instead of randomly generating the communication costs we used four communication models: a totally interconnected model with small cost, the same with heavy cost and a chain-like model (each processor is connected only with two neighbours) with small and heavy communication costs.

As all the methods tested use some kind of heuristic, their results show a significant variance, "good cases" (where the heuristic ruler matches well to the problem) and "bad cases" can be found for any of the four algorithms. For this reason, we present our results in the form of histograms and express the performance of the algorithms by means of average. The histograms

alg/SP results(%)	worst	best	average
BBOPT	-44.12	38.46	3.39
BBH2	-37.75	69.44	9.7
DLS	-32	137.5	46.8

Tableau 2.2: Performance comparation of Springplay with the reference algorithms

2.22, 2.23,2.24 compare an algorithm pair by calculating the ratio

$$\frac{t_{sched,alg1}}{t_{sched,alg2}}100\% - 100\%$$

of schedule lengths produced for the same input graph and by representing the distribution of this expression for the ensemble. There are less samples in the case of BBOPT and BBH2 ensembles because they could not terminate the computation for bigger node numbers. It must be noted that in spite of claims in [Green87], BBOPT heuristic function can significantly overestimate the finish time of a partial schedule thus it cannot always find the optimal solution. This is the reason why Springplay could produce better results than BBOPT in numerous cases.

In table 2.2 we compare the worst, the best and the average performance ratio for each pair. The advantage of Springplay against BBOPT is 3.4%, against BBH2 it is 9.7%. The advantage is more significant against DLS : the method presented here is 46.8% better and there is an important number of cases where Springplay is 80-100% better.

The significant performance advantage of Springplay is demonstrated on an example signal-flow graph (Figure 2.15). The graph was chosen among the randomly generated test set. Figures 2.16, 2.17, 2.18 and 2.19 shows the schedules generated by BBOPT, BBH2, DLS and Springplay, respectively.

- Springplay does not schedule the longest path on one processor. The "critical path" techniques loose much of their efficiency on heterogeneous environments.

- We supposed asynchronous buffered communication model in this article. It means that send activities are guaranteed to be scheduled before or paralelly with corresponding receive operations but no other restrictions are imposed.

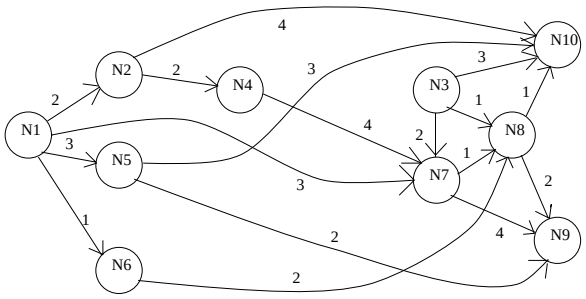


Figure 2.15: The example DAPEG

- Springplay not only generated the shortest schedule but used less processors than BBOPT and BBH2.

Figures 2.20 and 2.21 shows the convergence process in two cases. For generating these plots the original algorithm was slightly modified, after each iteration the final schedule routine was called and the length of this schedule is shown on the diagrams. This modification does not affect the parallelism optimization forces, they are calculated from the actual schedule generated by the internal list scheduler as before. Figure 2.20 shows the convergence in the case of the example DAPEG. The algorithm finds the solution in one step, the following steps are generated to trigger the stopping condition. In figure 2.21 the schedule length changes are shown in the case of a 20 node DAPEG chosen from the test set where a limit cycle can be observed at the end of the convergence process. This shows that the heuristic rule does not guarantee that the schedule length will not increase. The simple stopping conditions used during the tests does not consider this fact, we intend to develop more exact stopping criterias.

2.5 Springplay with enhanced communication model

So far we used a totally interconnected model and we supposed that the topology of the processor connections can be fully modelled by communication cost constants. This assumption is only approximatively correct. If

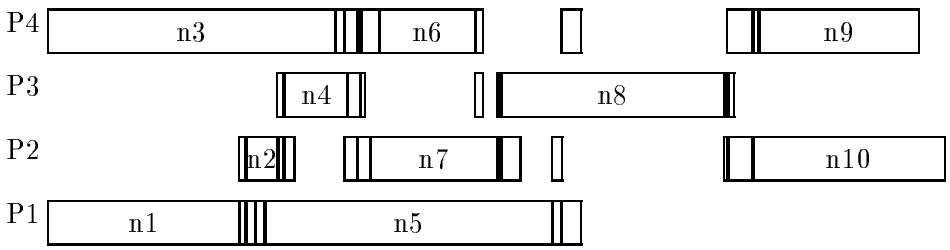


Figure 2.16: Example schedule by Branch&Bound OPT

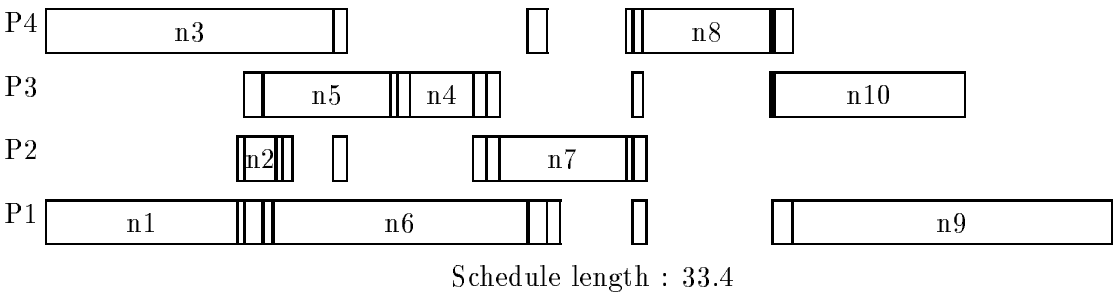


Figure 2.17: Example schedule by Branch&Bound H2

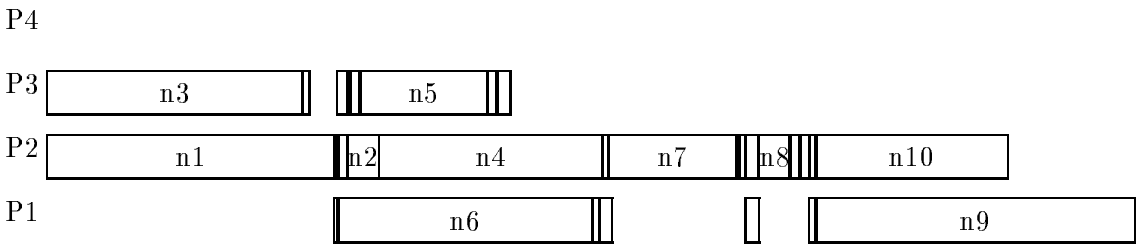


Figure 2.18: Example schedule by DLS

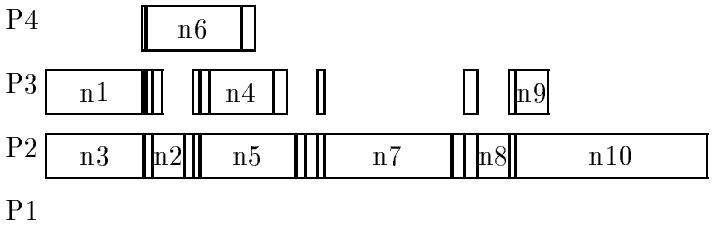


Figure 2.19: Example schedule by Springplay

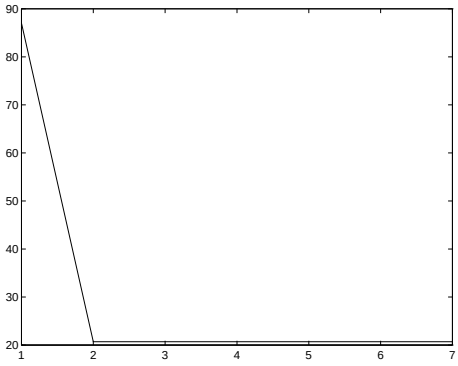


Figure 2.20: Convergence in the example case

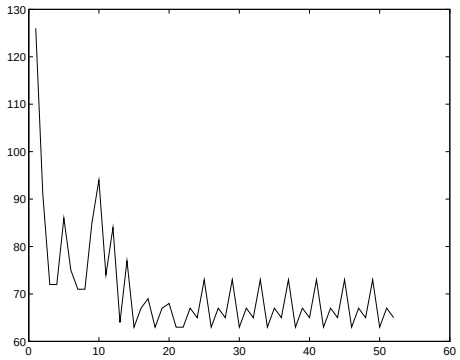


Figure 2.21: Convergence in a 20 node case

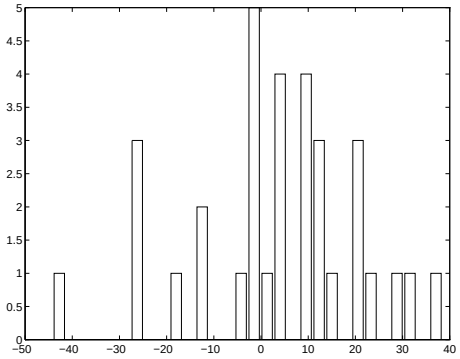


Figure 2.22: Distribution of BBOPT/SP results, 33 samples

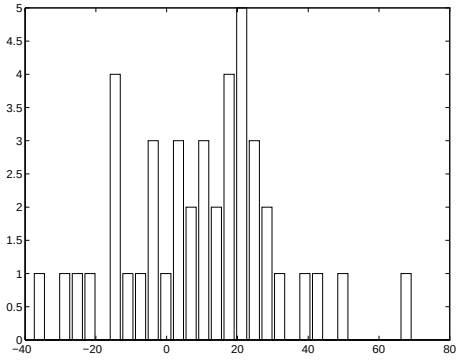


Figure 2.23: Distribution of BBH2/SP results, 43 samples

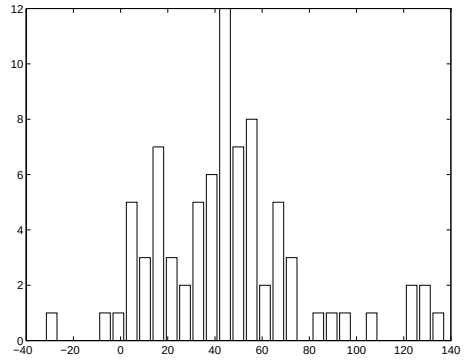


Figure 2.24: Distribution of DLS/SP results, 80 samples

more processor pairs share the same communication channel (for example bus-like structures), collisions can occur that increase the communication time significantly.

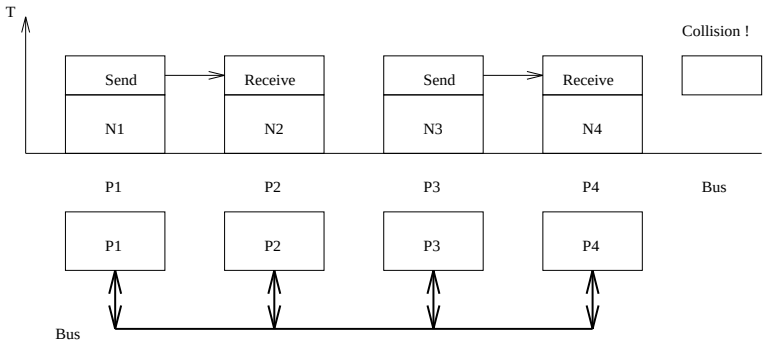


Figure 2.25: Communication activity collision on a bus

Fig. 2.25 depicts a case when two communication activity pairs collides on a bus shared among four processors. The bus arbitrer will delay one of the activity pair effectively doubling the communication time in that case. If the scheduler algorithm does not take into account this collision, it will underestimate the communication time that results in big differences between the calculated and real schedule lengths.

Other problem is the effect of *routing*. If the target architecture is not totally interconnected (processors are connected only with certain other processors and not with all the processors), it might be necessary to pass information between processors that have no direct link with each other. In this

case the data packet must be routed through other processors till we get to the final destination.

For correctly modellizing the effects above we have to consider communication links as resources (similarly to processors) and we have to schedule the activities on them as on processors. We will call a physical communication link between a certain processor pair *channel*. Each possible communicating processor pair is assigned a channel on which that pair accomplishes its data transfer. Channels have activity scheduling similarly to processors, a data transfer in progress occupies the channel for the duration of the activity and no other activities can be scheduled during that time.

We suppose *static routing* (fig. 2.26) as done in SynDEx. Static routing means that before the scheduler algorithm is launched, we make the target architecture totally interconnected by finding a route between any two processor pairs. This simple scheme cannot exploit multiple paths connecting the same processor pair but it is easy to implement and in many practical cases (dedicated signal-processing systems) there are no multiple paths in the architecture as it would introduce extra cost. A routing table is constructed that store for each processor pair the path through which the data packet can be passed.

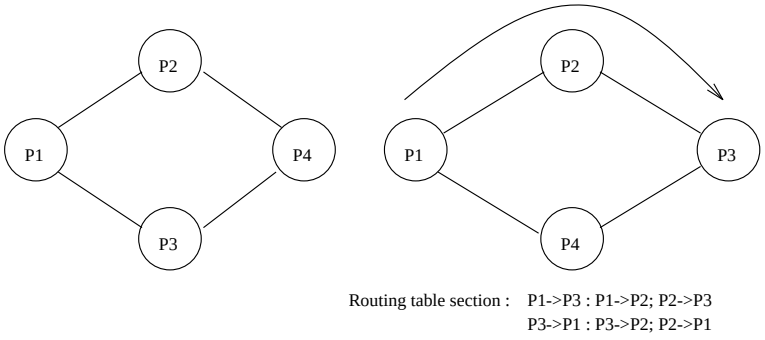


Figure 2.26: Static routing

Algorithms that can handle correctly the channel collision and routing effects are rare, they are mostly of list scheduler class. The technique used by these algorithm is called *tentative routing*. It means that the algorithm actually *measures* the necessary communication time in a partial schedule. It supposes that a certain node will be put to a certain processor, routes and schedules all the necessary communication activity on the channels and processors, measures the resulting communication time then removes these

communication activities. It will use the communication time it got for deciding, which processor will be allocated for the node and after the decision it will finally schedule the communicaction activities and the node itself on the chosen processor updating the partial schedule. This technique can be easily immplemented in a list scheduler as this algorithm advances always according to dependency constraints so we always have a valid partial schedule. The version of Springplay presented in section 2.4 takes the nodes one by one in an arbitrary order so partial schedule is not available all the time. For this reason the communication time factors in section 2.4.3 cannot be calculated in the enhanced communication model.

In this section we will present a version of Springplay that is adapted to this model. First let us define the model itself.

- Beside processor activities we will also define *channel activities*. Channels represent physical communication links, one channel belongs to each connected communicating processor pair. Channel activities are scheduled similarly to processor activities; a communication activity being accomplished on a channel occupies that channel for the duration of activity.

- We use static routing, the routing path between any processor pair is calculated before the execution of the scheduler.

- Communication activities reserve no time on the processor. In contrast, they reserve the appropriate duration on the channel.

- If a certain communication activity needs routing, the later channel activity in the routing list can start at the end of the previous channel activity and no sooner (fig. 2.27).

The main modification in the Springplay algorithm principles is that the node state modification phase is integrated into the final list scheduler. The list scheduler is modified so that after having chosen the candidate node, it calls the node state modification routine that calculates the forces, modifies the node state, calls the internal list scheduler if necessary. Then the node will be scheduled on the processor determined by the node state. Descendants are added to the ready node list if they are ready to execute then the algorithm proceeds with the next candidate. This turn of node state modification is finished if the ready node list is empty.

As we are inside a list scheduler loop, we always have a partial schedule, we can use tentative routing as in a list scheduler to calculate $T_{n,j,i}$ values

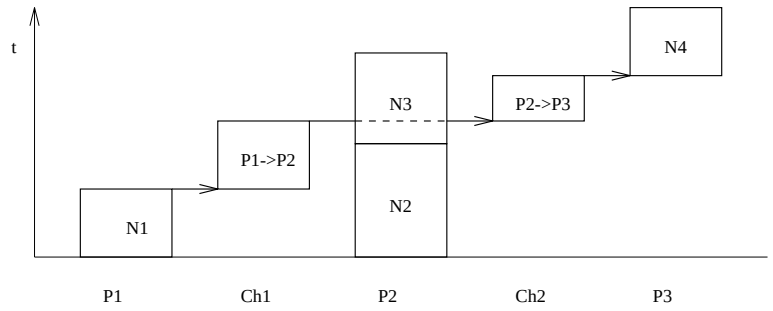


Figure 2.27: Channel activities while routing P1-¿P2; P2-¿P3

in eq. 2.23. The solution above allows us to combine the list scheduler approach (so a partial schedule is always present) with the "converging" Springplay approach. The modified Springplay algorithm adapted to the enhanced communication model (RSP) is the following in pseudocode.

```
procedure RSP
  Naive Schedule;
  BestSchedule = Actual schedule;
  loop 4*N times
    Initialize the ready node list to the input nodes;
    while ready list not empty
      Evaluate the earliest possible start time of
        ready nodes;
      Select candidate;
      Modify node state of the candidate;
      Recalculate M, dt;
      Call internal list schedule if changement of
        principal processor;
      Schedule candidate on its principal processor;
      Add ready nodes to the ready list;
    end while;
    if ActualSchedule is better than BestSchedule
            BestSchedule = ActualSchedule;
  end loop;
/* Output of the algorith is in BestSchedule */
end;
```

RSP was compared with the SynDEx [Sor94] and DLS [Sih93a] list sched-

alg/RSP results(%)	worst	best	average
RDLS (light)	-11.06	-1.88	8.88
RDLS (heavy)	-35.87	1.07	31.03
RSYN (light)	-16.90	-2.16	22.40
RSYN (heavy)	1.33	40.55	105.17

Tableau 2.3: Performance comparation of RDLS and RSYN with RSP on the homogeneous set

ulers whose original versions were prepared for routing and channel collisions. To make distinction between the SynDEx and DLS prototypes used with the simplified communication model and these ones with the routing extension, we will denote the routing versions as RSYN and RDLS.

First we tested RSP on a 10-member set of randomly generated homogeneous graphs. The hardware model we used was a 4-processor one where the processors were connected only with two neighbours in a chain-like structure. (fig. 2.28) The communication cost of the three channels were 0.1 (light cost) and 2 (heavy cost). Table 2.3 shows the best, worst and average performance ratios and figures 2.29, 2.30, 2.31, 2.32 show the distribution of results in the four cases. The performance of RSP is worse now with some percent than without routing and it puts RSP at the same quality level as RDLS. As RDLS needs much less calculation, RSP is not recommended in the homogeneous case. It is remarkable that RDLS and RSP produces significantly better results than RSYN in the "heavy" communication cost case. We think that this is the result of descendant consideration in RDLS and RSP.

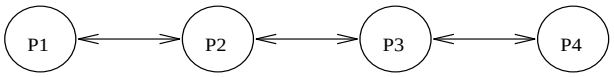


Figure 2.28: Hardware model used during the experiments

As RSYN does not support inhomogeneous architectures, RSP was compared only with RDLS in the inhomogeneous case where the same target hardware was used on a 10-member randomly generated graph set. The results are depicted in table 2.4 and figures 2.33, 2.34. We can see that despite the performance degradation of Springplay, RSP still produces considerable

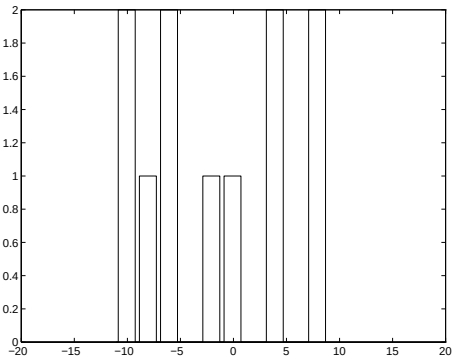


Figure 2.29: Histogram of the performance ratios, homogeneous case, RDLS light

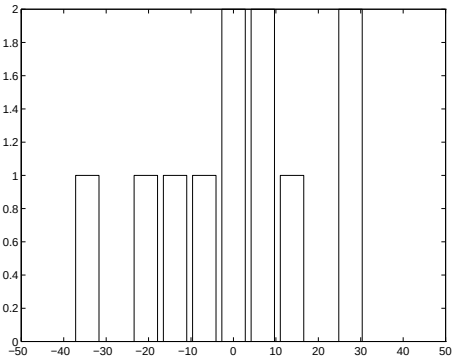


Figure 2.30: Histogram of the performance ratios, homogeneous case, RDLS heavy

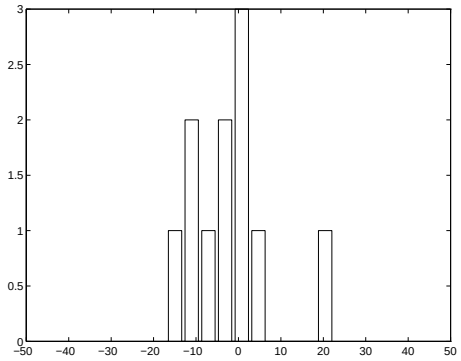


Figure 2.31: Histogram of the performance ratios, homogeneous case, RSYN light

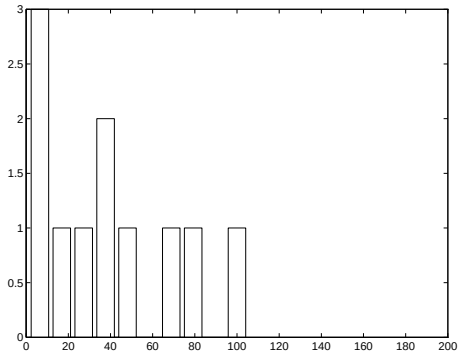


Figure 2.32: Histogram of the performance ratios, homogeneous case, RSYN heavy

alg/RSP results(%)	worst	best	average
RDLS (light)	8.7	28.87	47.28
RDLS (heavy)	-6	15.53	3

Tableau 2.4: Performance comparation of RDLS with RSP, inhomogeneous set

performance advantage (28 and 15%).

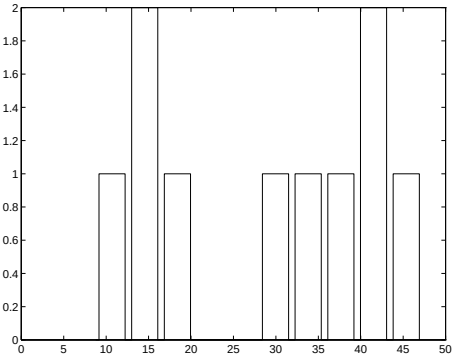


Figure 2.33: Histogram of the performance ratios, inhomogeneous case, RDLS light

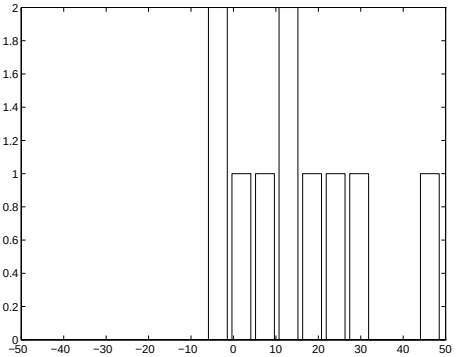


Figure 2.34: Histogram of the performance ratios, inhomogeneous case, RDLS heavy

In this section we presented a version of Springplay that has been adapted to a more exact communication model. As this extension introduces more nonlinearity, Springplay falls more easily into local minimas so its performance necessarily deteriorates. We showed, however, that despite this performance degradation RSP still produces better results than its competition.

2.6 Pipelined Springplay algorithm

The function to be minimized by the scheduling can be

1. The *latency* between the arrival of the input data and response.
2. The *input data period time*. In this case we are not interested in the latency but we want to feed new data into the system as fast as possible.

The first requirement generally arises in control system when the stimulis can be quite random but the response must be quick. The second approach can be found frequently in DSP systems where data arrive at fixed sampling frequency and arbitrary delay can be introduced by the processing system as long as this delay is constant.

When scheduling onto a multiprocessor system one exploites *spatial* and *temporal* concurrency [Hoan93] as shown in Fig. 2.35.

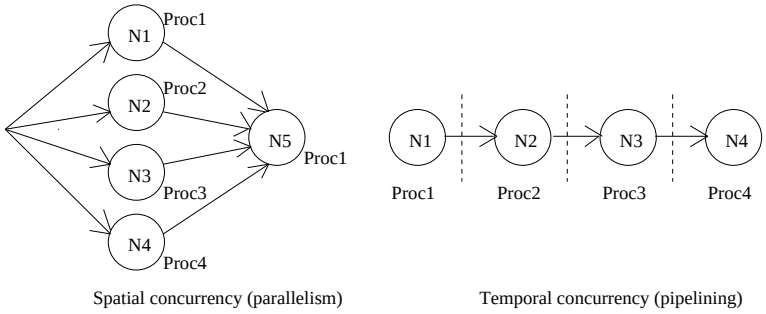


Figure 2.35: Temporal and spatial concurrency in a signal-flow graph

Spatial concurrency (parallelism) means that tasks can be executed on different processors without breaking the dependence constraints. In the case of temporal concurrency (pipelining) the graph is cut into chains of stages with every stage processing the results of the previous stage. The task of a pipeline scheduler is to exploit both spatial and temporal concurrencies to achieve minimal pipeline stage time.

The success with the Nonpipelined Springplay scheduler (SP) gave us the idea to extend Springplay for supporting heterogeneous pipelined realizations. In the following we present the Pipeline Springplay scheduler (PLSP) and we demonstrate its performance on a few examples. As many details are common in SP and PLSP, we will focus only on the differences.

We introduce only one new item to the system model: the scheduler can create pipeline stages anywhere in the graph by inserting delays into a graph branch breaking the dependence constraints this way. It must be guaranteed, however, that the delay inserted be balanced for each input node of every operation. (Fig. 2.36)

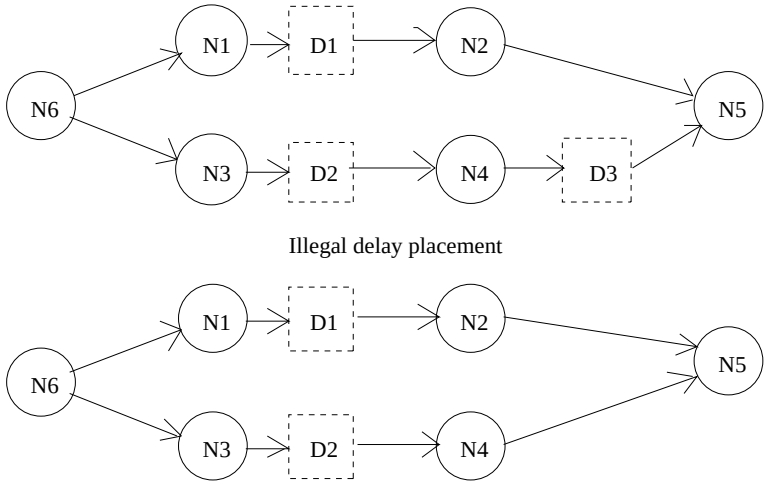


Figure 2.36: Delay insertion constraints

2.6.1 Changes in the algorithm principles

One of the biggest difference between the non-pipelined SP and the pipelined PLSP is the way the nodes are scheduled. In SP precedence constraints are always respected so the start time of any node must be bigger or equal than the maximum finish time of its every predecessor. In the case of PLSP a node is scheduled as soon as possible on its prescribed processor and a dependence constraints can be broken. Each time a node is scheduled earlier than its predecessor an *implicit delay* is created because the input of the node must

come from previous iteration of the predecessor (Fig. 2.37).

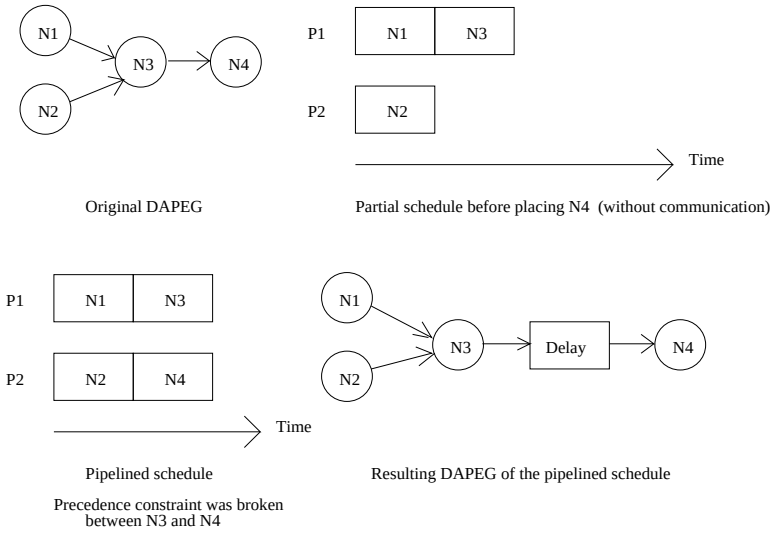


Figure 2.37: Implicit delays in a pipelined schedule

Before scheduling a node the communication activities and explicit delays are placed. The receiving communication activities are put just before the node to be scheduled, the send activity pairs are placed as near as possible to their corresponding receive activities. Figure 2.38 demonstrates the scheme.

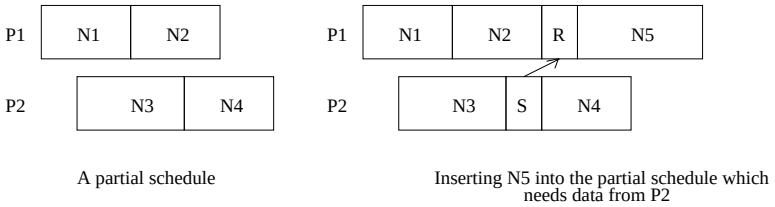


Figure 2.38: Communication scheme in PLSP

As it was mentioned before delays inserted by the algorithm must be balanced (Fig. 2.36). It can happen that implicit delays resulting from pipelining are not balanced, in this case explicit delay operations must be placed. The list scheduler in PLSP tracks the delay levels by means of the pipeline stage number (PSN). All input nodes are assigned a 0 PSN level. Each time a dependence constraint is broken the PSN on that branch is increased by one. The successor node gets the maximum PSN of all its

inputs. (See fig. 2.39)

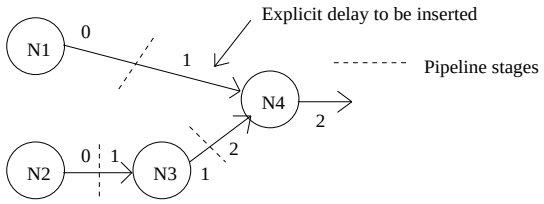


Figure 2.39: Calculating pipeline stage numbers

An explicit delay is inserted into the branch if the PSN on that branch is smaller than the PSN of the operation it feeds. The length of the delay is the difference of the PSN of the operation and the PSN of the input branch. PLSP has a built-in mechanism which considers the cost of the explicit delays and tries to eliminate them.

Each time the list scheduler is invoked, the length of the actual schedule is compared against the best schedule produced so far. The best schedule is updated to the actual schedule if the actual schedule is shorter. The pseudocode representation of PLSP can be seen below.

```
procedure PLSP
begin
  Initialize all node principal
    processors to the 1st processor;
  Initialize all node states to
    the center point;
  ActualSchedule = List Schedule;
  BestSchedule = ActualSchedule;
  Limit := 4.0 * number of nodes;
  dt := 0.001;
  maxdF := 0;
  ActLimit := 10;
  IterationWithoutChanges := 0;
  IterationCount := 0;
  do
    ChangeList := empty list
    for all nodes do
      F:= Calculate force;
      dV := F * dt;
```

```
        Modify node state;
        if principal processor changed then
          List Schedule;
          add changing to ChangeList;
          if ActualSchedule is shorter than
             BestSchedule
             BestSchedule = ActualSchedule;
          endif
        endif
        M = 0.9 * M + 0.1 * F;
      endfor
      Update dt according to maxdF;
      Increment IterationCount;
      if ChangeList == empty list then
        Increment IterationWithoutChanges;
      else
        IterationWithoutChanges := 0;
      endif;
      if IterationCount > Limit
        ActLimit := 0;
      endif;
    until
      (IterationWithoutChanges > ActLimit );
    return BestSchedule;
  end;
```

The force system consists of 4 different components, each represents a certain property of the schedule.

$$\vec{F}_n = \vec{F}_n^T + \vec{F}_n^C + \vec{F}_n^B + \vec{F}_n^D \tag{2.31}$$

1. $\vec{F}_n^T$ execution time minimizing component introduces preference toward processors on which execution time is shorter. This component is calculated in exactly the same way as in SP.

2. $\vec{F}_n^C$ communication minimizing component optimizes the communication costs. It is same with the corresponding SP force as well.

3. $\vec{F}_n^B$ processor balance component which tries to balance the processor loads.

4. $\vec{F}_n^D$ explicit delay elimination component which moves out the resolver system from states where the cost of explicit delays is excessive.

In the following sections we detail the two new force components.

2.6.2 Processor balance component

This component assures that the algorithm strives for schedules in which the length of the schedule is minimal by moving nodes from more heavily loaded processors to less used ones. This is achieved by creating forces toward each processor which is proportional with the difference between the maximal occupied time and the actual occupied time of the processor. Let us define t_p^{occ} occupied time on processor p as the sum of the execution time of all operations except for the communication activities (i.e. operations and explicit delays). We will define the maximal occupied time as the following :

$$t_{max}^{occ} = max(t_p^{occ}), 1 \leq p \leq P$$

The processor balance component can be imagined as a "pression" which pushes the nodes to less loaded processors. The force exerted to node n toward processor p is :

$$\vec{F}_{n,p}^B = (t_{max}^{occ} - t_p^{occ})\vec{e}_{n,p} \tag{2.32}$$

It means that the node is placed more possibly to less loaded processors. The resulting force is the sum of this component for every processor :

$$\vec{F}_n^B = \sum_{p=1}^{P} \vec{F}_{n,p}^B \tag{2.33}$$

Originally we were experimenting with a force function which considered that if a node is moved to other processor the t_p^{occ} of that processor will be increased by the execution time of the node on that processor. This force function, however, resulted in a too stable system which was prone to stick in local minimas. The force function used in the final version introduces deliberate instability so that the algorithm can move out from these traps.

2.6.3 Explicit delay elimination component

As it was discussed in section 2. explicit delays must be placed if the implicit delays resulting from pipelining are not balanced. Depending on the system

model delays can be considered costless or having a certain cost. By our opinion explicit delays do have cost, for example in a software realization the routines storing and retrieving data to and from the delay data field need certain excution time and in a hardware realization the delay occupy die area. PLSP has an option so that it strive for explicit delay elimination.

The delays are eliminated by moving nodes to other processor which are most possibly the causes of the explicit delays. This is accomplished by creating forces based on a "tag" variable denoted as t_n^{tag}. The list scheduler also calculates the tag variables. At the beginning each node's tag variable is initialized to 0. Each time an explicit delay is inserted the cost of this explicit delay is propagated backward to each predecessor of the input branch into which the delay is placed. Fig. 2.40 illustrates how the tag variables are propagated.

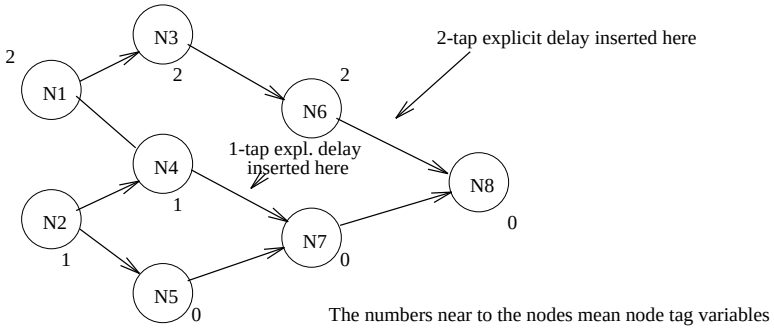


Figure 2.40: Illustration for the tag variable propagation

The routine is the following :

```
procedure TagInput(node,tag)
begin
 if node's actual tag variable > tag
           return;
 else
 for each input of the node
   node tag variable := tag;
   TagInput(input node,tag);
 end for;
end;
```

This routine is called with the node whose output the delay is placed to

and the cost of the delay. The cost of the delay is the execution time in the case of the software realization. The following notation will be used: $C(n)$ cost function means the cost of a n tap delay. It can be the execution time of the delay in software realization (can be 0 or a certain cost depending on the code generator). With this notation the delay elimination component is :

$$\vec{F}_n^D = \sum_{p=1, p \neq p_n}^{P} -C(t_n^{tag})\vec{e}_{n,p} \tag{2.34}$$

The solution presented here tries to mark the nodes which are considered responsible for the creation of the explicit delays and moves them to other processors.

2.6.4 Results of PLSP's testing

We have realized PLSP in a prototype version in Lisp language to test and tune the ideas developed. PLSP was tested on a variety of example DAPEGs. In figures 2.41,2.42 a 2-stage FIR lattice filter and the schedule made by PLSP can be seen. In this case the target system is homogeneous. Fig. 2.43 shows the DAPEG of the resulting schedule.

The length of the actual schedule during the execution is depicted in figure 2.44. We changed our original concept in which we supposed gradual improvement of the schedule length, now we expect no convergence but rather we pick the good quality solutions from the solutions generated. It must be noted that the schedule length can *grow* so the algorithm can escape from local minimas.

The pipeline stages created by the algorithm are the following :

1. N1,N2,N3,N6
2. N4,N7,N9
3. N5,N10
4. N8,N11

The prototype scheduler could not consider that N2 and N7 are actually delays so they were scheduled as simple nodes.

Figure 2.45 shows the resulting schedule if the target is heterogeneous. The input DAPEG is the same as in figure 2.41 but the execution time of nodes N3, N4, N8, N9 (multipliers) is 1 only on processor 1, on processors

2,3,4 it is 4. The length of the actual schedule is shown in figure 2.46 and the pipeline stages are :

1. N1,N2
2. N3,N4,N6,N7
3. N5,N9
4. N8,N10,N11

Finally we present the schedule generated by PLSP in the case of the example DAPEG in [Hoan93]. This example is used in [Hoan93] to demonstrate that the algorithm described there can consider parallelism and pipelining at the same time. Figure 2.47 shows the DAPEG and in figure 2.48 the schedule generated by PLSP can be seen. PLSP generated the same stage-time bound as the algorithm in [Hoan93]. This shows that PLSP has the same capability of exploiting parallelism and pipelining concurrently. As the cited article considers explicit delays costless, we switched off explicit delay scheduling for this example.

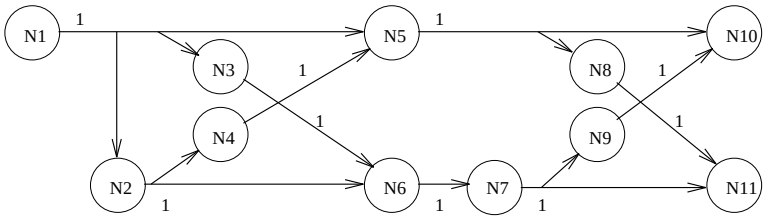


Figure 2.41: DAPEG of the FIR Lattice

2.7 Conclusion on the Springplay algorithms

We have presented a new scheduling method which try to overcome the problem imposed by the local decision making in heuristic algorithms. An application of this idea was investigated and the results are promising. During the experiments Springplay produced slightly better solutions in $O(N^3)$ computational complexity as B&B in exponentional complexity. The tests

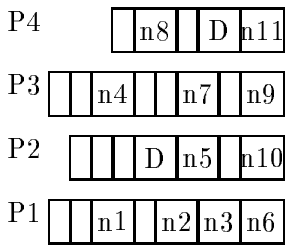


Figure 2.42: FIR Lattice schedule by PLSP

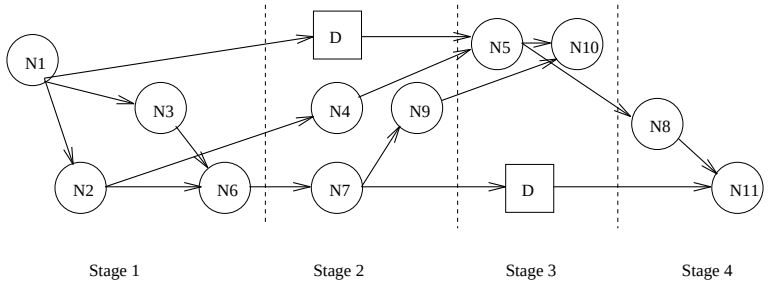


Figure 2.43: DAPEG of the pipelined schedule

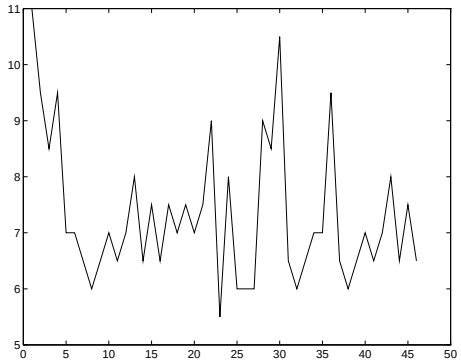


Figure 2.44: Length of the actual schedule during the execution

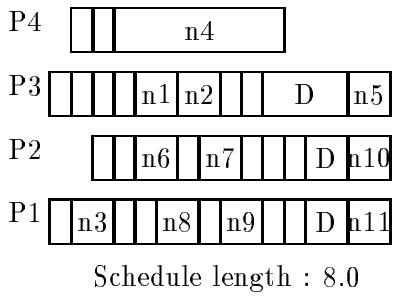


Figure 2.45: FIR Lattice schedule by PLSP in the inhomogeneous case

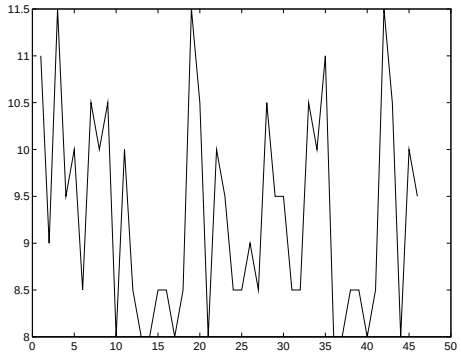


Figure 2.46: Length of the actual schedule during the execution in the inhomogeneous case

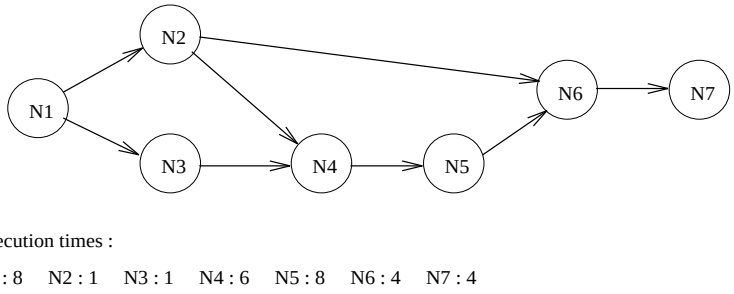


Figure 2.47: DAPEG of the parallelism/pipelining tradeoff example

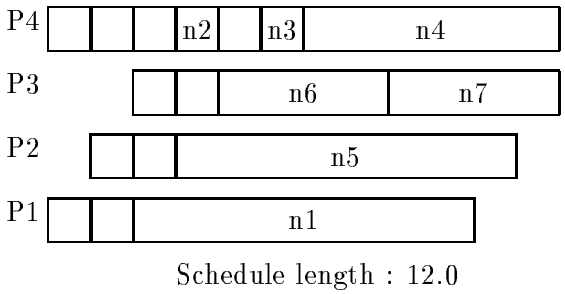


Figure 2.48: Schedule generated for the parallelism/pipelining tradeoff example

revealed the advantage of global optimization over local decision making as well. Springplay offers good-quality solutions for the heterogeneous scheduling problem for which there are not so many existing algorithms.

The problems with the Springplay approach are originated from the fact that it is not proven. Experiences with the neural networks show that such strongly nonlinear systems are very difficult (if not impossible) to prove. The effectiveness of this algorithm is shown only by statistical analysis. There are still several important questions to answer : stability, convergence to solution and local minimas. Despite its good performance in average, there were few cases when Springplay produced bad quality results. We intend to examine thoroughly these cases, refine the heuristic rule and the structure of the resolver system.

Chapter 3

Rafael SFG compiler

In this chapter we will present the SFG compiler called Rafael. The goals of the Rafael project are the following:

- Create an SFG compiler that can facilitate the task of prototyping DSP algorithms on multiprocessor signal processing system
- Rafael should require small computing resources so that it can be hosted on smaller and cheaper computers
- As we work with all kinds of DSPs, Rafael should support heterogeneous target systems
- Considering the fast development in the signal processor industry, the code generator layer should be easily reprogrammable so that new DSPs can be included by the user

In the following we present first the most important SFG compilers then we will deal with the user model and the internal structure of the Rafael system. In section 3.4 we compare our system with the presented ones then conclusions are drawn.

3.1 Existing SFG compilers

A number of block diagram-based design systems exist. We have chosen three of them for several reasons. Many ideas of the structure of Rafael was taken from the now historical Gabriel system (section 3.1.1). Ptolemy (section 3.1.2) is the most comprehensive, most flexible signal-flow framework

which has huge impact on the field. SynDEx is a relatively small system but it is interfaced with the SIGNAL language and critiques on the SynDEx execution scheme influenced greatly the Rafael algorithm model.

Nevertheless, we have to mention here the commercially available DSPlay (Burr-Brown) and SPW (Signal Processing Workstation) (Comdisco) systems. DSPlay is PC-based, it can simulate the input block-diagram and can generate code for the AT&T DSP32. The Comdisco system started as a simple simulator but actually it is able to produce highly optimized code for almost all the DSP types and can even generate circuit description. As of June 1994 the partitioning on multiprocessor DSP system must have been done by hand.

The Cathedral system [DeMan86, Lann93] devoted to circuit synthesis features SFG partitioning-scheduling but it uses the Silage functional language [Gen90] as its input.

Ritz et al. presented a block diagram oriented tool called DESCARTES [Ritz92, Ritz93] in which they introduced the idea of *scalable synchronous data flow* (SSDF). SSDF extends the usual SDF approach with the blocking factor N_b attached to each block in the schedule. Each time the block is invoked, its assigned operation is executed N_b times so it consumes $N_b \cdot W_p$ tokens from its input arcs and produces $N_b \cdot U_p$ tokens on its output arcs after each invocation. (see section 1.2.1 for discussion of SDF) This approach try to overcome the overhead of register-memory operations if an atomic operation is executed only once. For example an adder must load both of its arguments to registers, perform the addition then store the result. If we execute the same adder on a block of input samples producing a block of output samples, we can exploit the pipeline capabilities of DSPs thus executing the additions and register-memory transfers at the same time. DESCARTES is capable of generating C or assembly source text from the block diagram description and efficient algorithms were presented that for optimal blocking factor calculation and for memory management. DESCARTES still cannot handle the problem of run-time decisions.

3.1.1 Gabriel

Gabriel, presented in [Lee89b] was the first system capable of generating executable code at Berkeley in which the synchronous dataflow paradigm was implemented. Its predecessor, BLOSIM [Mess84] was only a simulator. The terminology of BLOSIM survived in Gabriel and later in Ptolemy so we present it briefly.

The operations (or *actors* by the terminology of the Berkeley team) are called *stars*. A cluster of stars forming an interconnected SFG is called *galaxy*. The final SFG can be hierarchical composed of a number of galaxies, a set of interconnected galaxies is called *universe*. Gabriel has two level of user interface. The graphical dataflow organization is used where appropriate: when describing the algorithm in dataflow format. The stars have textual definition. This mixed description form helps to avoid the common problem of the graphical description systems which use graphical terms where they are not handy.

Gabriel was realized on a Unix workstation under X-Windows. The graphical interface was borrowed from a CAD project. This graphical editor called VEM allows manipulating graphical objects, provides the standard pan, zoom, copy, drag features. The edited SFG is then passed to the Gabriel kernel which was implemented in Franz Lisp. This kernel schedules the graph and simulates it or generates code according to the user's wish. If code generation is requested, the generated code is downloaded into the hardware system where it can be executed. Figure 3.1 shows the structure of the Gabriel system.

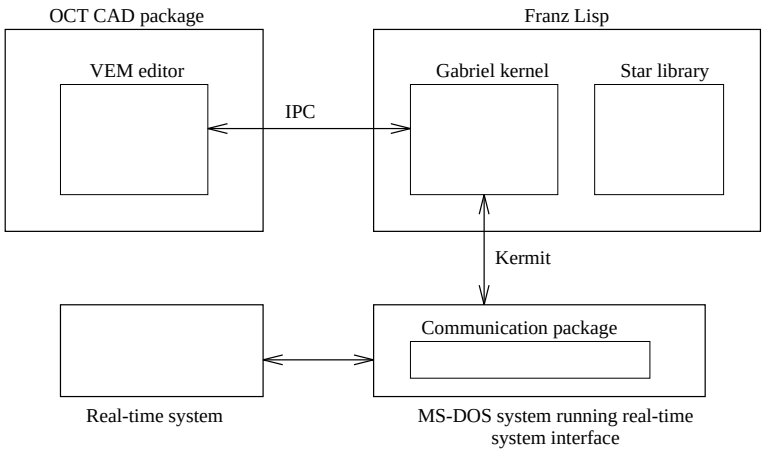


Figure 3.1: Structure of the Gabriel system

One of the most striking feature of Gabriel is its *programmable star library* which influenced a lot the database of our Rafael system. A Gabriel star is described by a Lisp structure which looks like the following in a simple example:

```
(defstar log
  (descriptor 'Computes the log of the input')
  (param base 10.0)
  (input in)
  (output out)
  (function gabriel-log)
)

(def_function gabriel-log()
  (output (quotient (log in) (log base)) out)
)
```

The star library entry has a *header* (marked by `defstar`) and a *function body* (marked by `def_function`). The header structure stores information about the inputs and outputs of the operation, a short textual description for human readers and the parameters and their default values. The last `function` entry points to the *star function* which gets executed whenever the star is invoked. This star function can actually execute the operation assigned with the star in simulation mode or can generate code for the actual target processor in code generation mode. It is important to note that the code generator star library is written in Lisp so a code generator function can be quite intelligent when it decides on the text to be generated depending on the parameters, size of the inputs, etc. Beside the star function, a Gabriel star can have initialization/termination functions that are called once before the first invocation and after the last invocation of a star. Processors are described in a similar way creating Lisp lists that contain the target system characteristics: number of processors, processor memory, special hardware units connected to processors, communication channel characteristics between the processors and communication code generator routines. The Gabriel system is strictly homogeneous: there can be only one star library in the memory.

The Gabriel system reads in first the source graph description and generates the schedule. It uses the algorithm in [Lee87] to construct acyclic precedence graph from the synchronous dataflow graph (section 1.2.3) then a simple list scheduler similar to that of Hu [Hu61] schedules this APEG onto the target architecture. This scheduler does not consider communication cost and is suitable only for homogeneous architectures. When the schedule is ready, Gabriel executes the star functions of the nodes one by one. If the actual star library is for simulating the graph (in this case the

schedule must be made onto one processor) this will actually execute the graph producing simulation results. If the star library is designed for code generation, invoking the star functions will result in generating code for the target system. In this case the target system may contain multiple processors.

The Gabriel system has the following interesting features:

- It handles multiple sample rates which results naturally from its input format, the synchronous dataflow graph.
- It has a second user level, the star library programming level in Lisp which allows the user to create new stars easily and to add intelligent optimization/code generation features to the existing star library.

The main weaknesses:

- It does not address the question of data dependent constructs, if-then-else, case, etc.
- It does not support heterogeneous systems.
- Its scheduler cannot be considered efficient.

Nevertheless, Gabriel's structure greatly influenced the design of the Rafael software (hence the similarity of the names). Gabriel was phased out by the Berkeley team in favor of a much more comprehensive framework, Ptolemy which will be presented in the next section.

3.1.2 Ptolemy

Ptolemy introduced first in [Buck91] is rather a framework than a design system, it allows mixing of multiple computation models. Its objective was to combine descriptions so that complex systems can be designed heterogeneously. The aim of supporting heavily distinct computational models was that each subsystem of a complex, bigger system can be designed, simulated or prototyped in a model that is appropriate to that part of the system. Ptolemy imposes very few assumptions about these models, practically there is only one constraint: a model to be included shows itself to the Ptolemy kernel as a functional block accepting tokens on its inputs and producing tokens on its outputs.

Ptolemy has a Lisp-like and a Tcl-based [Oust90, Oust91] textual and a VEM-based graphical user interface which allows the easy manipulation of graphs, input data and simulation/execution results.

Ptolemy is based on the principles of object-oriented programming [Buck94] and these principles are carried through with big care in this system. Ptolemy is implemented in C++. The basic element of the object hierarchy is the Block. The functionality of the Blocks are very similar to the star library elements in Gabriel (section 3.1.1). Blocks support methods that allow blocks to be initialized, started, killed. (see fig. 3.2). Blocks are connected with Porthole objects that provide connection port for the object. The actual connection is established through Geodesic objects which connect Portholes to Portholes. Blocks create Particle objects which are passed through the Geodesic objects to the Portholes of the receiving Block. Blocks can be organized into Galaxies. A Galaxy can embed other Blocks or Galaxies as well. A whole application composed of Blocks and Galaxies is called Universe.

Blocks contain a go() method which include the functionality of that block. When this method is invoked the Block in question examines its Portholes for the presence of Particles, consumes them if the Block can be fired and produces particles on its outputs. Depending on the computation model "firing" can mean different activities ranging from actual computation to code generation. The firing of the blocks is managed by the Scheduler object which invokes the go() methods of the Blocks under its control.

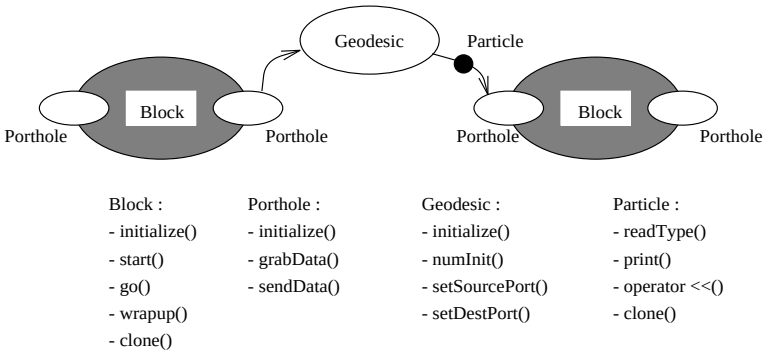


Figure 3.2: Blocks, Portholes, Geodesic objects, Particles

The execution control of a group of Blocks (a Galaxy for example) is controlled by the Target object that the Galaxy is attached to. Target is responsible for all the activities concerning the execution of the Galaxy on that Target: initialization of the hardware, setting up its operating mode,

provide the scheduling, execute the scheduled Galaxy on the hardware and shutting it down if necessary (fig. 3.3). Target will call its associated scheduler to control the execution order of the operations. This scheduler may decide that certain portion of the input Galaxy be executed on an other target. In this case it passes the Galaxy part to that target which in turn will call its scheduler to execute it. This way the different characteristics of different targets are totally separated.

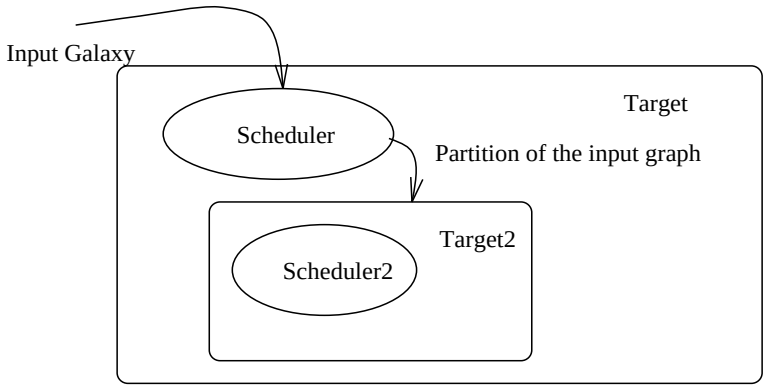


Figure 3.3: Schedulers and targets

A computational model includes a set of Blocks, Targets and their associated Schedulers. We use "computational model" in the sense that it describes how Blocks interact with each other. An implementation of such a model in Ptolemy is called *Domain*. Domains can be arranged into hierarchies, in this case all the Blocks and Targets of an inner Domain can be used in the outer Domain. Schedulers may belong to more Domains but a Scheduler in the inner block is not necessarily valid in the outer block. Ptolemy provides a mechanism so that a Block of a certain domain hide an other Domain.

The actual Ptolemy implementation (as of version 0.5) specifies the following Domains.

SDF Synchronous Dataflow as introduced in Gabriel.

DDF Dynamic Dataflow domain [Denn80] in which Blocks are enabled by data at their inputs that they may or may not consume and they may or may not produce data at their outputs. Disadvantage is the necessary run-time scheduling.

BDF Boolean Dataflow (section 1.2.2).

DE Discrete Event when only the system state transitions are modelled.

Thor A Stanford RTL (Register Transfer Level) Simulator

MQ Message Queue

CP Communicating Processes

CGC C code generation in SDF model

CG56 DSP56000 code generation in SDF model

CG96 DSP96000 code generation in SDF model

Silage A functional language [Gen90]

Vhdlf Functional modeling in VHDL

Vhdlb Behaviorial modeling in VHDL

Sproc Code generation for the Sproc DSP in SDF model.

Ptolemy is a very comprehensive framework and its generality and extendablity results in more and more new domains. One of the latest developments is the integration of ESTEREL tools (separate domain has not been presented yet however). The most frequent critique of Ptolemy is that it is too "liberal" in the sense that it allows mixing of design styles that cannot coexist well. The framework provides only the mechanism of coexistence but it does not solve any compatibilty problems of the models which were mixed in an application universe. It does not force the designer to follow a "good design style". For this reason Ptolemy itself is as good as the domains it supports. The most comprehensive domain in Ptolemy actually is the SDF domain which is supported by several code generators and simulators. As more and more developers include their computational models in Ptolemy, this situation is changing quickly.

3.1.3 SynDEx

SynDEx [Sor94] is a code generator environment designed to be interfaced with the synchronous language compilers (section 1.3). It has a graphical and textual user interface that allows users to construct the algorithm block

diagram entirely in SynDEx. It is designed, however, rather to receive the algorithm graph from a synchronous language compiler. Actually SynDEx is interfaced in such a way with SIGNAL [Bour94] and work is under way to create a common format for the SIGNAL, LUSTRE, ESTEREL languages so that they can send the result of compilation to SynDEx or other code generators. The algorithm model of SynDEx is the *conditioned signal-flow graph* already presented in sections 1.3.3,1.3.4. It means that each node has a clock it is associated to which results in a *condition input* for each node (fig. 3.4).

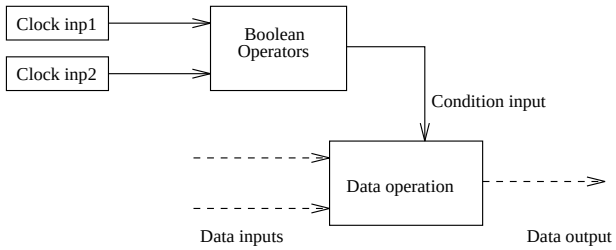


Figure 3.4: Conditioned signal-flow graph

A node is fired if all its input variables (including the control variable) has been produced by predecessor nodes and its control variable is *true*. The scheduler considers the condition input dependency as any other dependency: it is equivalent with supposing that each condition is true and each node can be executed. This way the original conditioned signal-flow graph is transformed to a synchronous signal-flow graph and static scheduling can be used. The original conditioned signal-flow graph is thus partitioned into a condition calculating part (which is unconditioned) and a data processing part (which can be conditioned). It is the responsability of the SIGNAL compiler (or the input graph designer) that a proper condition signal be assigned to each node. The intermediary format (called Sisy) contains node definitions :

```
(function add "calls" add "dt" 10 "i/o" integer ?zs ?'1' !zs1)
```

and condition definitions that describe the clock dependencies :

```
(exec hadd/hs memory S add default2 default1)
```

This latter line says that `memory`, `S`, `add`, `default2`, `default1` nodes should be executed only if the `hs` output of the `hadd` operation is true.

The actual SynDEx implementation does not group the conditions. It means that if `OP1` and `OP2` have the same condition (`C`) and they are scheduled one after the other, the following code will be generated :

```
if C then
  OP1
endif
if C then
  OP2
endif
```

SynDEx generates C code for the T800 transputer and TMS32C040 DSP. The environment supports only homogeneous target systems. The actual direction of its development aims heterogeneous targets as well.

The biggest problem about the SynDEx system is caused by the way it handles the conditions. The code generator does not group operations scheduled one after the other with the same conditions into one `if ... endif`. Other drawbacks are that SynDEx does not support heterogeneous architectures and it can generate only C code.

3.2 Rafael basic structure

3.2.1 Major design considerations

The Rafael structure was designed according to the four main goals introduced at the beginning of this chapter. The support of heterogeneous systems needed a flexible operation library or - even better - programmable code generator module. Considering the code generator programmer's convenience, compiled languages can be quickly eliminated because it would need the recompiling and relinking of the code generator modules each time the database is modified. A system constructed in this way would be much more prone to system crashes as compiled languages allow great liberty in manipulating the system resources. We decided that reprogrammable parts of the code generator be implemented in an *interactive, interpreted language*. As we intended to provide the possibility of important intelligence in these modules (as they determine the quality of the code generated) we wanted to choose a more powerful language. Considering the possible candidates we chose Lisp because of the following advantages :

- It is a very powerful language that allows run-time program creation and it is equipped with efficient database handling capabilities.
- Lisp interpreters are available in relatively small memory requirement versions which fits well to the small computer (PC) we planned the system to run on.
- Excellent quality public domain versions have been written and distributed for several platforms in source code.
- It is a common language in CAD systems.

We must consider, however, the slow execution speed of Lisp which is even more serious obstacle on a small PC system. Although in the sense of ease of programming it would have been more advantageous to realize the system entirely in Lisp, this solution would have resulted in inacceptable run time on the target system.

3.2.2 The structure of the Rafael system

For the reasons mentioned in the previous section reason we chose a hybrid structure depicted in fig. 3.5.

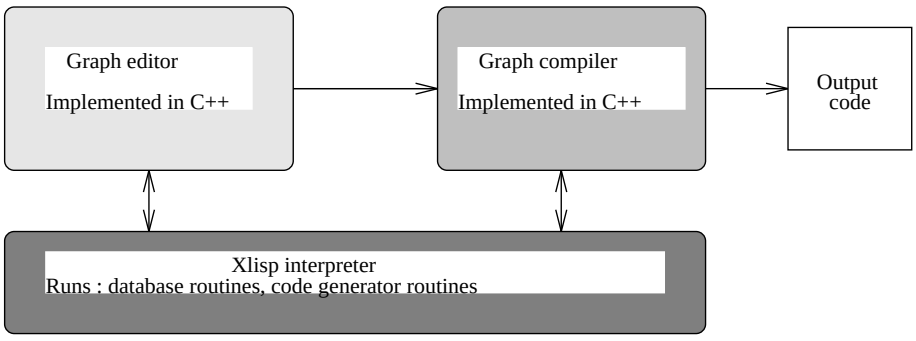


Figure 3.5: Structure of the Rafael software

Each part of the software where user modifications are not supposed was implemented in C++. This gives us a relatively powerful language with acceptable execution speed. Programmability is provided at Lisp level where an interface has been defined for the database and code generator programmer. By means of this interface the user can extend the database

and the code generator library. The compiler core calls these routines from C++ level and uses their return value appropriately.

This solution needed separate tasks and interprocess communication between the tasks. The minimal "operating system" that is sufficiently popular and needs small resources was the Microsoft Windows. At that time Linux (a small Unix version for PCs) was not in the state that we could have considered it as a alternative against Windows. By my personal opinion Windows is a poorly designed, inefficient "operating system", today we would choose some other platform.

Thus, Rafael was implemented under MS-Windows, parts of this software (fig. 3.5) run as separate Windows tasks and they are connected through the interprocess communication channels of Windows. The popular Xlisp was chosen as Lisp interpreter for Rafael because it is close to Common Lisp and it is available in C source. Xlisp was ported to Windows platform and the necessary interprocess routines were inserted that allows this Lisp interpreter to run as a server task.

The three Rafael software components have the following tasks.

Graph editor The name is a bit exaggerating as the Rafael framework is far from a confortable working environment. It features a multi-screen text editor for creating/modifying graphs in textual format, initializes the Xlisp server and launches the Rafael compiler on the actually edited graph.

Graph compiler It is the SFG compiler. This program analyzes graph description, makes the scheduling and generates the output text. It can run standalone as well, not only from the framework.

Lisp interpreter The operation database and its associated code generator routines are realized in Lisp. The client programs launch the server and send requests to it through interprocess links. Requests are actually Lisp commands which are executed by the server and the result of the Lisp command evaluation is returned to the caller C++ program.

As we can see the Rafael software architecture is very similar to that of Gabriel (section 3.1.1) hence the similarity of the names. Rafael is different from Gabriel at the following points :

- Rafael's whole structure is adapted to the small host systems it runs on. Not the whole compiler was implemented in Lisp, only a part of it.

- As we will see, Rafael's whole design including the database, the scheduler it uses is adapted to heterogeneous systems. Gabriel was *multi-target* as it supported multiple start libraries. Rafael is *truly heterogeneous* as multiple target processors can coexist in the same operation library.

- Rafael supports a limited form of run-time decisions as its importance has been underlined many times both in the literature and in the practical engineering work. It will be detailed in section 3.2.4.

- Rafael features more advanced and efficient scheduler algorithms.

3.2.3 Rafael nodes and connections

The Rafael software model defines *nodes* that represent certain operations and *connections* between them. Nodes can be of the following types.

Operations Operations cover functions attached to a certain node. An operation is a parametrizable function. The number of inputs, outputs, the execution time and the operation of the function itself can depend on constant parameters.

Probes Probes cover functions whose task is to acquire input data from the environment of the dataflow system and send output data to the environment of the dataflow system. Probes are treated as simple operations (with nonzero execution time, if necessary), the only difference is that they are explicitly forced to certain processors by the user. It derives from the fact that in a given hardware system the input and output hardware is assigned to prescribed processors.

Delays Delays are special operations in the sense that they consist of two parts: a delay input (where new data is put into the delay) and delay output (where new data is retrieved from the delay). Rafael always treats a delay parts as two distinct operations. It is guaranteed, however, that output of a delay be scheduled always before the input of the same delay.

Each node input/output can have a type. Type is a character string which is checked for matching when node inputs/outputs are connected. Rafael allows dynamic type names resolved in compile-time that match to every static type name and solves the type name ambiguities. In Rafael

dynamic type names start with the "TYPE" string, for example "TYPE23" is an dynamic type string. An adder that can add any type of data can have "TYPE23" type of each input/output node. When any of the input/outputs is connected to an output/input with static type, the dynamic type is replaced by the static type by the type checker. For example if the output of the hypotetical adder above is connected to an input node with "TIME" type, "TYPE23" is replaced by "TIME" for all the adder input/outputs and type checking continues on the inputs. Figure 3.6 illustrates the process.

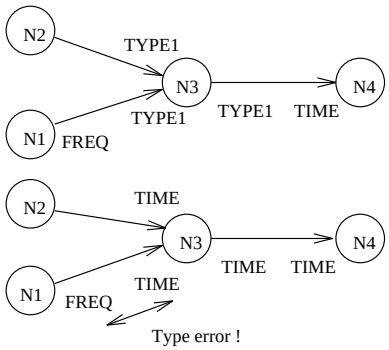


Figure 3.6: Propagating type names in Rafael

Depending on the operation library, "tokens" can have arbitrary size. The actual Rafael operation library supports one-dimensional vector tokens.

3.2.4 Rafael software model

Rafael accepts a restricted version of synchronous dataflow graphs for scheduling. This restriction means that if a node output produces or input consumes more than one tokens, it can be connected only to an input or output that consumes or produces one token. See figure 3.7 for examples. This simplified scheme allows Rafael to support practically relevant upsampling/downsampling operations without getting to a problematic loop scheduling problem [Bhat94a, Bhat94b].

Rafael has two software models. The first one is a classical synchronous dataflow model which does not allow run-time decisions. This model has been proved to be too restrictive but this is the most effective one. It allows all kinds of supported operations in the dataflow graph but no conditional structures are permitted, we will call it *static model* in the future. The static scheduler will be invoked for this graph and a single-block schedule will be

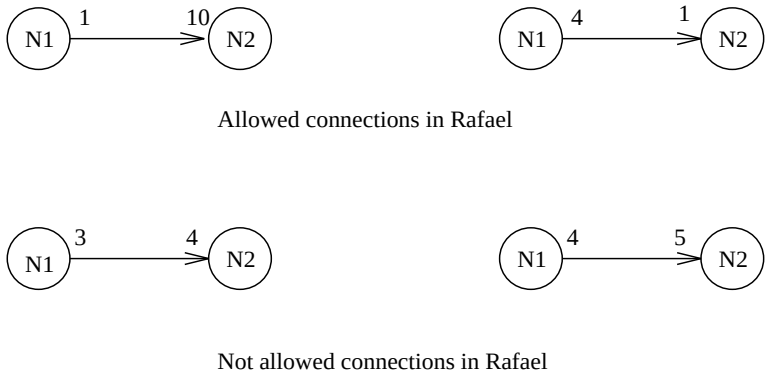


Figure 3.7: Rafael's restricted synchronous dataflow graph

generated. This model is the restricted version of the second one that allows runtime decisions.

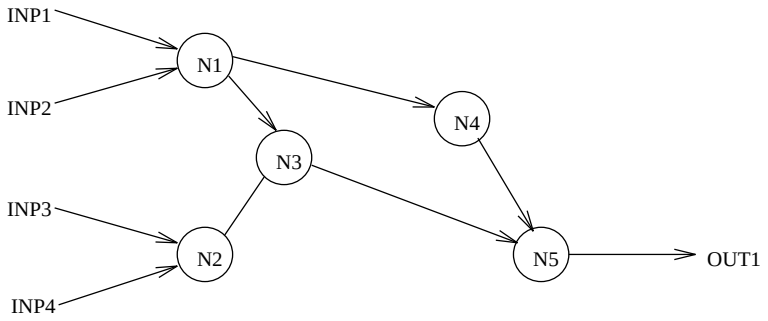


Figure 3.8: Example static model graph

Based on the conditioned dataflow model of synchronous languages (section 1.3.4) a *conditioned block dataflow model* was implemented in Rafael, we will call it *dynamic model*. As presented in section 3.1.3 inserting `if ... endif` constructs around each operation and considering all conditions *true* is an evident but not too efficient solution for the run-time decision problem. Instead Rafael forces the SFG designer to *group* parts of the graph to a *block*. A block contains a graph portion for which the following holds true :

1. The graph portion inside a block is a synchronous dataflow graph without run-time decisions
2. All the operations in this block depends on the *same condition*.

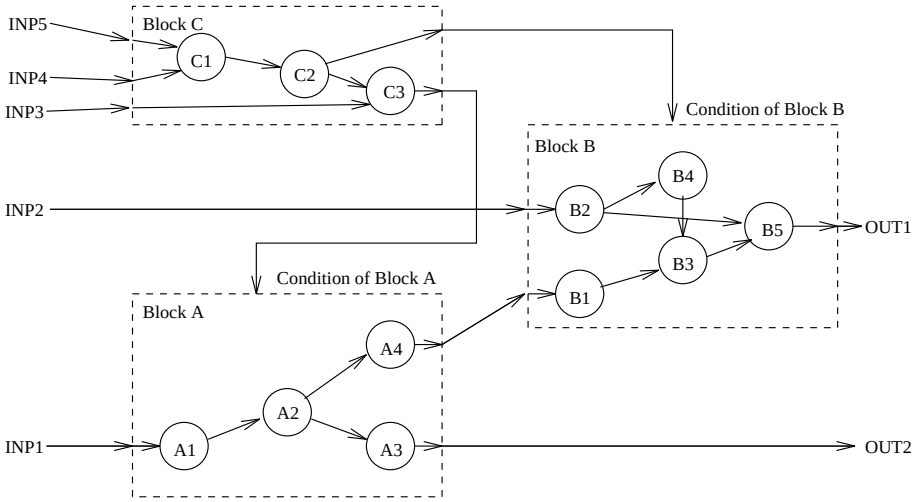


Figure 3.9: Example dynamic model graph

Outside of the blocks only probes and blocks are allowed. This is called *root* level. Operations are embedded into blocks, this is the *block* level.

The simple scheduling scheme used in Rafael solves the scheduling problem in two passes.

1. First it prepares static schedule for each block independently. Variables are propagated through the root level block connections and static scheduler is invoked for the block.

2. Dynamic root-level scheduling. Blocks are considered as operations which run on all the processors at the same time. A list scheduler traverses the block connections and builds the order of the block considering only dependency relations. During the execution a block may or may not be executed depending on its condition input variable (if any).

Figure 3.10 demonstrates this method on the example dynamic model graph in fig. 3.9.

Advantages of the conditioned block schedule are the following :

- We can provide conditional structures while preserving static scheduling.

- The user of the system is forced to group nodes with the same condition

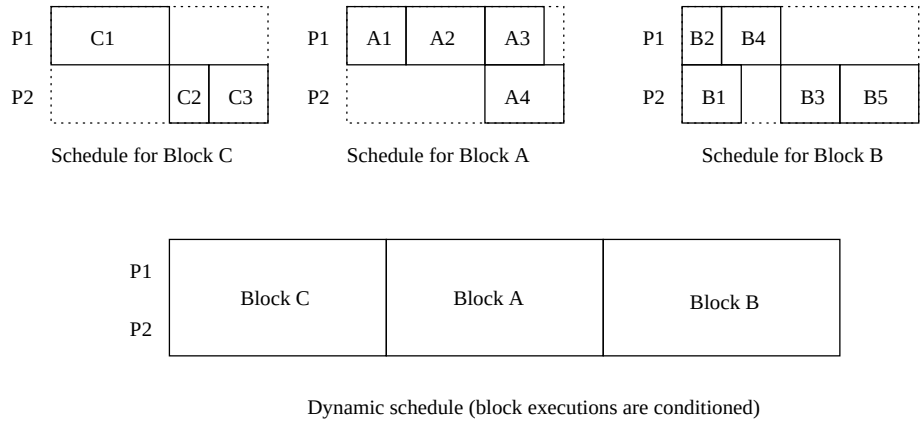


Figure 3.10: Example of dynamic model scheduling

together, the performance loss resulting from the repeated conditional statements is thus avoided.

- The static scheduling algorithm estimates the reality much better than in the SynDEx case. As a block contains only synchronous dataflow, the static scheduling is always exact, not only in the worst case as in SynDEx.
- SIGNAL compiler readily makes the operation grouping itself.

We have to mention the following disadvantages :

- If the blocks contain to few operations, static schedules of blocks can be too sparse. In this case even true dynamic scheduling could provide better solution.
- It is very easy to construct an incorrect graph. Consider the graph in fig. 3.11. In this example Block B depends on Block A and in the root-level dynamic scheduling it is scheduled after Block A. It cannot be guaranteed, however, that Block A was really executed because it depends on a run-time decision. If the condition of Block A is not true, Block B will get its input from obsolete temporary variables producing bad result. As Rafael makes no effort to check the calculation of condition variables, these situations cannot be signaled by the compiler.
- Other effect of the fact that Rafael does not analyze the condition calculation is that all the condition variables must be recalculated in each

iteration. We can recall that SIGNAL compiler laboriously optimizes the condition tree so that its output program can be the "laziest" which means that `if ... endif` structures belonging to a clock expression on the lower level of the clock tree will be appropriately nested into `if ... endif`s of upper level clocks. The scheme presented above will flatten the clock tree putting all clock expressions to level 1.

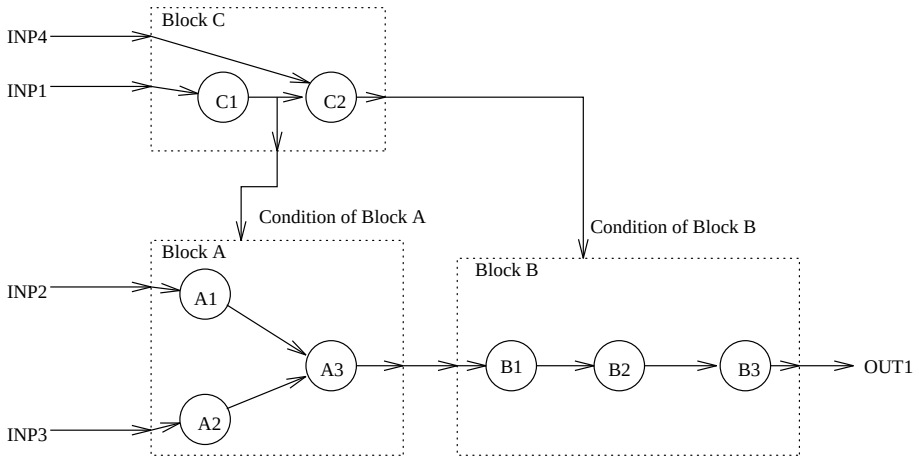


Figure 3.11: Example of possibly erroneous graph

In spite of the disadvantages we consider that the Rafael conditioned block model avoids successfully the dynamic scheduling and in the case of large static blocks and few decisions (which is often true at a DSP algorithm) it is sufficiently efficient.

3.2.5 Rafael hardware model

Rafael supposes arbitrary number of interconnected, heterogeneous processors as target system. The communication hardware connecting these processors can be heterogeneous as well. The static scheduling algorithm prescribes, however, that execution times of operations on all the processors of the target system and communication times on all the channels in the target system be known in advance. These calculation/communication times can depend on certain parameters, in the case of calculations these parameters are defined by the operation type, in the case of communication it depends on the amount of data units passed between the processors.

Rafael uses a simplified communication model, critiques say it is oversimplified. Rafael considers the communication structure totally interconnected but allows different communication costs for both directions of each channel. The actual Rafael implementation does not have router algorithm so if the target architecture is not totally interconnected, virtual communication layer must be provided by the operation library programmer.

The basic Rafael communication notion is the *channel*. Channels are resources that are shared by processor pairs willing to communicate. A channel is assigned to each processor pair and that channel is occupied for the length of the communication between that processor pair. Other processor pairs having the same channel number has to wait with their request until the channel is free. Channels represent hardware resources used for communication (bus, network, communication links, etc.). The processor pair-channel number asignment is fixed in the hardware database.

Each communication activity can have three property which are returned by the hardware database functions to the compiler core.

Activity time It is the time during which the communication activity occupies the processor it is scheduled on. If the communication hardware needs constant interaction with the processor (buffered serial line hardware, for example) the activity time is the same as the time required for the communication activity. In the case of DMA it is the DMA initialization time.

Survive time This is the time which is needed to finish the communication after the activity itself finishes. For example a DMA is initialized during the activity time then it accomplishes the task. During the survive time the variable which is sent cannot be reused and no new communication activities can be accomplished on that channel. On the receiving side all the calculations which need the received variable are delayed until the end of the survive time.

Synchronous flag This flag controls the scheduling of communication activities. If this flag is false for a certain communication activity, the scheduler can put the send activity before the receive activity of the same communication pair. No "crosses" are allowed, however (see fig 3.12). If the synchronous flag is true, the send and receive activities are scheduled strictly at the same time.

Figure 3.12: Allowed and not allowed communication schemes

3.3 The internal structure of the Rafael system

3.3.1 Graph description language

The actual Rafael implementation does not contain graph editor, the user must construct the input algorithm graph himself or herself. A simple graph description language is used for this purpose which will be described briefly in this section.

According to the two software models in Rafael, there are two variations of the graph description language. In the first variation (synchronous dataflow) only probes, nodes, delays and connections are allowed. Let's see an example graph:

```
PROBE I 1 1 A_TYPE 1 1
PROBE I 2 1 A_TYPE 1 1
PROBE O 7 1
NODE 4 ADD (4)
NODE 5 ADD (4)
NODE 6 ADD (4)
NODE 8 MUL (4)
NODE 3 CONST ((1 2 3 4))
DELAY 9 4 1
CONNECTION 1_1 4_1
CONNECTION 2_1 4_2
CONNECTION 2_1 5_1
CONNECTION 3_1 5_2
CONNECTION 4_1 6_1
```

```
CONNECTION 5_1 6_2
CONNECTION 6_1 8_1
CONNECTION 3_1 9_1
CONNECTION 9_1 8_2
CONNECTION 8_1 7_1
```

PROBE <I/O> <nodenum> <type> <upsample> <downsample>
<I/O> is the input/output probe type, <nodenum> is the number of the node, <type> is its type name. For convenience of the compiler, Rafael stores the relative sample rate of the node in rational form. <upsample> is the nominator, <downsample> is the denominator of the relative sample rate. (see section 3.3.4).

NODE <nodenum> <operation> <parameters> <nodenum> is the node number, <operation> is the function attached to the node, <parameters> is the parameter list which depends on the function. In the case of the example ADD operation it deteremines the size of the vectors to be added.

DELAY <nodenum> <delay size> <delay length> <nodenum> is the number of the node, <delay size> is the size of one token it stores, <delay length> is the number of delay stages data fed into the delay goes through. Delays exlicitly have TYPE1 inputs/output types.

CONNECTION <onode>_<onum> <inode>_<inum> Defines a connection between the output numbered <onum> of the node having <onode> node number and an input described by similar parameters.

The conditioned block dataflow model allows block definitions beside the elements above. In this model only probes, block definitions and connection definitions are permitted at root level.

```
BLOCK MADD2 I1->6_1:TYPE1 I2->5_2:TYPE1 I3->5_1:TYPE1 \
            O1->6_1:TYPE1
NODE 5 MUL (4)
NODE 6 ADD (4)
CONNECTION 5_1 6_2
ENDBLOCK MADD2

BLOCK MUL2 C:BOOL I1->6_1:TYPE1 I2->5_2:TYPE1 I3->5_1:TYPE1 \
```

```
          O1->6_1:TYPE1
NODE 5 MUL (4)
NODE 6 MUL (4)
CONNECTION 5_1 6_2
ENDBLOCK MUL2

PROBE I 1 1 A_TYPE 1 1
PROBE I 2 1 A_TYPE 1 1
PROBE I 3 1 A_TYPE 1 1
PROBE I 10 1 BOOL 1 1
PROBE O 7 1

NODE 4 MADD2
NODE 5 MUL2
CONNECTION 10_1 5_C
CONNECTION 1_1 4_1
CONNECTION 2_1 4_2
CONNECTION 3_1 4_3
CONNECTION 1_1 5_1
CONNECTION 2_1 5_2
CONNECTION 4_1 5_3
CONNECTION 5_1 7_1
```

The only new element is the `BLOCK ... ENDBLOCK` definition pair. Blocks group their internal nodes into one virtual operation that can be placed by a `NODE` definition. An internal node in a block is identified by its block name and node number, two blocks can have internal nodes with the same node number as internal nodes are invisible outside of a block. The block header contains the following elements :

I<inputnum> – ><inp nodenum>_<inp inputnum>:<typename>
Connects <inputnum> input of the virtual operation represented by the block to <inp inputnum> input of <inp nodenum> internal node. Type of the block's input is set to <typename>. Data fed into that input of the block will be propagated to the internal node's input.

O<onum> – ><onodenum> _<out outputnum>:<typename> Connects <onum> output of the virtual operation represented by the block to <out outputnum> output of <onodenum> internal node. Type of the block's output is set to <typename>. Data produced by that output of

the internal node will be propagated through the output of the virtual node.

C:<typename> Indicates that the block has condition input and the type of the condition input is <typename>. Condition input can be referenced as "C" in the `CONNECTION` definition.

3.3.2 The database

Rafael provides a programmable operation and hardware database stored in Lisp. The database is accessed by the compiler core through Lisp functions. The interface of these Lisp functions is documented so that the database programmer can interface to the compiler core.

The database consists of two parts: operation database and hardware database. Operation database stores the actual function set for all the supported hardware devices while hardware database provides Lisp functions that can calculate every characteristic of the target hardware system which is necessary for scheduling and code generation.

The database is handled and maintained through the XLisp interpreter and stored in Lisp lists. Because XLisp runs under Windows , all its memory is virtualized so we can store the whole database in the memory of XLisp. This simplifies greatly the implementation of the database management because we simply use the built-in list manipulating functions of LISP.

The operation database

The operation database has two parts : operation headers and compilation strategy functions. The operation headers are stored in lists which are bound to the operation name. This list stores the following information:

- The name of the compilation strategy routine.
- The description of the input(s) (type, size).
- The description of the output(s) (type, size, storage class, sample rate factor).
- The execution time in system clock beats.
- Parameters. The parameters and their meaning are defined by the creator of the operation library. For example the parameters for the

FIR operation can be the length of the filter and the filter coefficients. The actual value of the parameters are supplied when the user places an operation, it is passed in the SFG script.

- Constructor and destructor routines. The compiler creates constructor function for each operation which requests it. The constructors are invoked before the operation is executed first time. Similarly, before the SFG execution terminates, destructor functions are called for the operations which need it.

The data structure above is described in a list like the following :

```
( ( strategy list)
  ( inputs )
  ( outputs )
  ( time function )
  ( parameters ) )
  ( constructor strategy list )
  ( destructor strategy list )
)
```

The strateqy list contains the names of the compilation strategy functions for each hardware devices. It has the following format :

```
( (device1 function1) (device2 function2) ... (deviceN functionN) )
```

The compilation strategy function is called each time during the code generation pass when the schedule contains a reference to that function and its program text must be generated. This LISP function gets the label lists of the input and output branch descriptors (effectively labels of data areas where the compiler allocated space for the temporary variables), the parameter list (which contains data like coefficient vector of a filter, e.t.c.) and returns the program text to the compiler which writes it into the output file. The strategy function can decide on the subroutine chosen or the form of the generated program text depending on the input and output connections and the actual parameters. The subroutine bodies can be stored in an ordinary object library, in this case Rafael will place only references into the code which can be resolved by the linker which belongs to the DSP's development system. This subroutine library can be created and maintained by the assembler and library manager tools of the DSP development software

package. Other design style is to inline all the operation bodies which results in slightly faster code but larger code size.

The excellent symbol handling capability of the LISP which makes this language so appropriate for the artifical intelligence applications can be exploited in this system and we can build significant intelligence into the strategy functions.

The input list stores the description of the operation's input. Its format is the following :

```
( ( type1 size1 ) (type2 size2) ... (typeN sizeN) )
```

where type is the freely chosen signal type (for example time for time domain signals) and size is the size of input vector accepted by this node. This size can also be a symbol from the parameter list (for example the size of an FFT input can be N where N is a parameter supplied by the SFG designer) or even a lambda function of the parameters. The type name can be either static or dynamic. Dynamic type names have the form of "TYPEn" where n is an integer number. Dynamic type names are resolved when they are connected to a statical one.

The output list is similar, but beside type and size it also contains the storage class specifier and the upsample and downsample factors. Its format is the following :

```
( ( type1 size1 st1 us1 ds1) (type2 size2 st2 us2 ds2) ...
  (typeN sizeN stN usN dsN) )
```

The storage class specifier shows whether the compiler has to allocate space for the output variable or the space is reserved by the operation. The us and ds values describe the change in sampling frequency caused by the operation. The us denotes the multiplication, ds is the division of the sampling frequency. For example the pair 2 1 means interpolation by 2.

The time function list stores a Lisp functions which get the bound parameter list and return the execution time of the operation on a given hardware. The list has the following format :

```
( ( device1 lambda1 ) ( device2 lambda2 ) ...
  ( deviceN lambdaN ) )
```

where lambda1 ... lambdaN are lambda expressions (no-header Lisp functions) which compute the execution time for the given device.

The parameter list contains operation-dependent data. For example in the case of an IIR filter it contains the size of the nominator and denominator coefficient vectors and the vectors themselves. In the operation header the list is stored in unbound form (without parameter values), the editor evaluates this list when placing an operation. The IIR parameter list would look like the following in unbound form :

```
(N COEF1 M COEF2) )
```

and in bound form (after the operation has been placed)

```
(3 (0.34 -0.2 2.12) 4 (0.23 0.77 0.192 2.94) )
```

This bound form is stored in the SFG description file and is passed to the execution time computing and stategy functions when necessary.

The constructor and destructor strategy lists have the same format as the strategy function. An operation may have constructor and/or destructor functions - pieces of code which are executed before the operation first runs and after the operation's last run. If the operation does not need such functions, NIL is stored instead of the name.

The following small code piece shows the implementation of the `ADD` database entry for the TMS320C30 and DSP96002.

```
(setq add '((
; c30add is C30 strategy function
             (c30 c30add)
; dsp96kadd is 96K strategy function
     (dsp96k dsp96kadd)
            )
; Has two inputs, each of size n (n is the operation parameter)
            ( (type1 n) (type1 n) )
; Has one output, size n, automatic storage,
; interpolating factor: 1
            ( (type1 n a 1 1) )
; Time functions for C30 ...
            ( (c30 (+ (* 2 n) 10) )
; and 96K
              (dsp96k (+ (* 2 n) 5) )
            )
; Has only one parameter (n)
```

```
            ( n )
; No constructor for C30 and 96K
            ( (c30 nil) (dsp96k nil) )
; No destructor for C30 and 96K
            ( (c30 nil) (dsp96k nil) )
  )
)
```

Target hardware database

The target hardware database provides the following information to the compiler core:

- Processor numbers and processor types in the target system.
- Activity, survive times and synchronization flag for any communication activity.
- Communication cost estimation for any communication path in the target system (for the scheduler)
- Channel-processor pair assignment for any processor pair.

A set of Lisp functions must be written for each target system. It is a relatively inconvenient solution but allows greater flexibility.

3.3.3 Rafael memory management

Rafael allocates memory for temporary variables in compile time. When the generated program runs on the target system, every variable is already assigned a memory address. Rafael implements a simple "first fit" dynamic memory allocation scheme when compiling the graph.

When a node is scheduled, Rafael allocates its output variables (the input variables must have already been allocated). The scheduler keeps track of the actual state of memory map by the means of chunk lists which describe, actually what size of blocks are occupied at what address in the memory of the target processor. When allocating a variable the memory manager simply walks this chain and finds the memory block with the lowest address which is big enough to accomodate the variable to be allocated.

When an output variable is created, its "scope" is established. A variable goes out of scope if all the operations that consume this variable has already

been executed. In this case the memory chunk assigned to the variable is freed and the place the variable occupied can be reused. As the scheduler cannot know when allocating the variable, on which processor(s) that variable will be consumed, every instance (variable sent to other processors) of that variable stays "alive" on every processor until all operations that consume that variable terminate.

A variable can be local or global. Local variables are used internally by blocks. A variable is local if it is created in a block not at root level and it is consumed only by the operations of that block (so it is not connected to a block output). Every other variable is global. Blocks have their own address maps that start at relative address 0. At the end of the scheduling when we know, how much memory is required for the global variables, local variable addresses are relocated so that these variables be allocated starting at the end of the memory allocated for global variables. Local variables of blocks thus overlay each other. (figure 3.13)

3.3.4 Compiler passes

Rafael compiler works in 5 passes.

Reading graph description file

The compiler reads in the SFG file and parses it syntactically. Then it analyzes the connection definitions and signals connection errors (connecting to nonexisting node, nonexisting input, etc.) During this phase the compiler rebuilds the tree in the memory of the computer, ready for analysis.

Type checking

The compiler resolves the dynamic type names and checks if there are type errors (see section 3.2.2 for further explanation). The type checker is a recursive routine that propagates the static type names from node to node substituting dynamic type names with static ones and signaling errors if type name violation is found. This algorithm is the following in pseudocode (omitting now the block handling logic):

```
; Node is a reference to a node input/output, CType is
; the type name to be checked against
function CheckTypeNames( Node , CType )
```

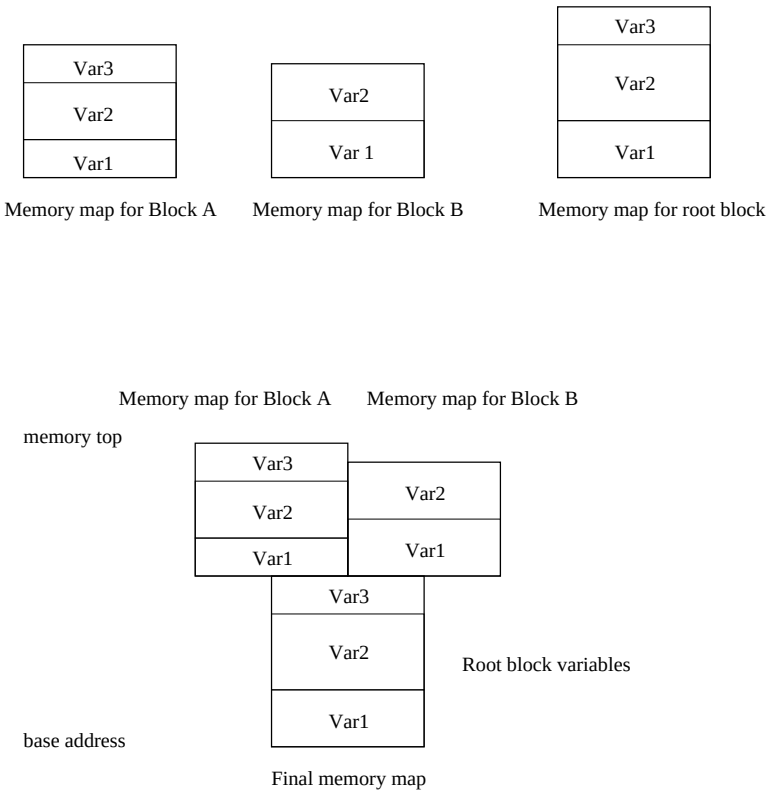


Figure 3.13: Block memory overlaying in Rafael (supposing 1 processor)

```
begin
  if Node's type is dynamic
    substitute all instances of that dynamic type name
    with ctype on that node;
  else
    compare Node's type with CType, signal error
    if not equal;
  endif
end

; Node is a reference to a node input/output,
; CType is the static type name of a node connected
; to that input/output
function CheckNodeType( Node, CType )
begin
  CheckTypeNames( Node, CType );
  mark Node typechecked;
  for i:=all input/outputs of the node Node belongs to
    for n:=all nodes connected to i
      if not typechecked before
        CheckNodeType( n,type of i );
      endif
    endfor
  endfor
end;
```

The type checking starts at descendants of probes as they are the only nodes that surely do not have dynamic types.

IPF checking

IPF stands for interpolation factor and is used to support Rafael's multirate features (section 3.2.4). IPF is the rate of the node's execution in the multirate model. IPF is represented by two distinct numbers, the nominator and the denominator so IPF:1,4 means 1/4 execution rate.

Rafael uses a recursive subroutine similar to the typechecker to propagate IPFs along the graph and looks for the minimal IPF factor. Propagating IPF means that the IPF at the input of the operation is multiplied by the sample frequency multiplication factor stored in the database at the output

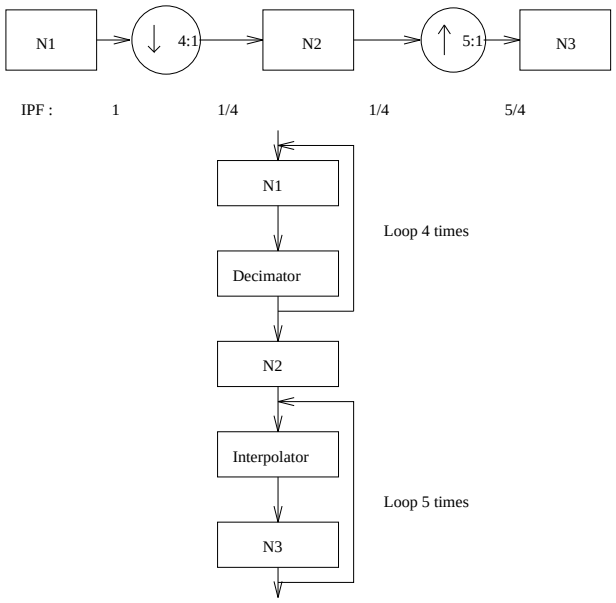


Figure 3.14: IPFs in an example graph and looped schedule

description yielding output IPF then it is passed to all the nodes connected to the outputs. The actual implementation of Rafael prescribes that the output sample on all the outputs be the same. During the IPF propagation the minimal IPF in the graph is recorded. As IPF is calculated by division or multiplication by integer factor, all IPFs in the graph must be integer multiple of the minimal IPF. So the factor

$$c_{loop} = \frac{IPF_{node}}{IPF_{min}}$$

is the loop count that determines, how many times an operation with IPF IPF_{node} must be repeated if the minimal IPF is IPF_{min}. Note that operations that change IPF are always executed on the higher sample rate of input and output sample rates. (fig. 3.14)

Scheduling

The formally correct, typechecked graph with IPF values for all the nodes calculated is then passed to the scheduler algorithm. The actual version of Rafael contains only the RHLS scheduler (section 2.3) but work is under way

to implement the much more efficient Springplay schedulers in the software. The only difference to the already presented version of RHLS is that it considers node repetition resulted by the multiple sample rate loops (see IPF checking section). The schedulers considers effective node execution time as $c_{loop} \cdot t^e_{n,p}$ and tries to group nodes with same IPF together.

Code generation

The scheduling done, Rafael generates the output text for each processor. The code generator walks the activity list on each processor then asks the Lisp code generator database functions to produce output text for them which is then sent to the output file. Separate output files are generated for each processor. The model of output text will be discussed in detail in the next section.

3.3.5 Code generation model

Rafael has a parametrizable code generator that allows each section of the text generated to be redefined. The code generator invokes Lisp functions that receive the parameters of the text section and the device for which the code will be generated then it is the responsability of these Lisp functions to produce the appropriate text. These code pieces are called *code generator service functions* and they complement the operation strategy routines. Every text section that Rafael writes to the output text file can be redefined by modifying either the operation strategy functions (in the case of operation texts) or the code generation service functions (headers, communication routine codes, etc.).

Rafael expects the database functions to produce ready assembly text that is not modified further by the core. As it was pointed out in [Lee89b] simply concatenating text pieces that correspond to blocks and not considering postoptimization possibilities that "smooth" the block borders can lead to important inefficiencies. For example two consecutive blocks where one of them is connected to the output of the second can result in a code fragment like this:

```
            ...
      move   r0,buffer     ; Storing the result of block 1
; End of block 1, head of block 2
      move   buffer,r0     ; Fetching first operand of block 2
            ...
```

Postoptimizer is proposed to filter out similar code sequences in the Gabriel system. [Pow92] presents another approach based on a kind of "meta-assembly" that allows multiple optimization strategies. For example it uses symbolic register names in this metalanguage that are substituted by real register names during the code generation in order to optimize register usage. A block supposes that it always gets inputs in registers and it always produce results in registers, "register-memory spilling" operations (register-memory transfers that save/retrieve register contents to/from temporary variables) are inserted by the compiler. The complicated strategies are reported to result in excellent quality output text. Rafael code generator functions can perform a lot of optimization but they do not know about the blocks that precede and follow the actual block so Rafael cannot exploit block interferences.

Rafael generates three text sections for each processor (that may be empty as well). For programmable processor-like devices that Rafael was designed for, the database programmer may wish to realize these three sections as subroutines. These sections are the following:

1. Constructor section. Called only once from the user program before the first iteration of the dataflow computation.

2. Operation section. Called once for each iteration. Calling the operation section entry label will actually execute the program generated from the SFG.

3. Destructor section. Called once after the last iteration of the operation section.

Each section has a start and end header that probably contain section head label in the start header and "return" instruction in the end header. The sections contain the text generated by the operation constructor, strategy and destructor functions.

If the compiled SFG was written in block conditioned model, each section has a separated part for each block. In the constructor and destructor sections it is rather a formality as Rafael guarantees no specific order among the operations when it generates constructor and destructor sections. In the operation section each block has a start and end header. The current operation library realizes blocks as subroutines so the start header defines a block entry point label and the end header contains a "return" statement. The block subroutine contains the operation body texts in the schedule order. After block subroutines were generated, Rafael emits the text for the

root block that contains probe calls and block invocations. Block invocations in the current operation library result in subroutine "calls" to block subroutines.

An example program generated from the second example SFG in section 3.3.1 is shown below.

```
;1.asm
;             Device type : TMS320C30
;Rafael code generator, Version 1.1
;Task requires 13 words of memory and takes 17 cycles
; to execute



VAR0    .set    arena+00h  ; Node 1 Type : A_TYPE, output 1
VAR7    .set    arena+04h  ; Node 6 Type : TYPE1, output 1
VAR1    .set    arena+04h  ; Node 2 Type : A_TYPE, output 1
VAR4    .set    arena+08h  ; Node 6 Type : TYPE1, output 1
VAR2    .set    arena+08h  ; Node 3 Type : A_TYPE, output 1
VAR5    .set    arena+0Ch  ; Node 10 Type : BOOL, output 1
;  Variables for block MUL2
VAR6    .set    arena+0Dh  ; Node 5 Type : TYPE1, output 1
;  Variables for block MADD2
VAR3    .set    arena+0Dh  ; Node 5 Type : TYPE1, output 1

; Constructor section
        .def    constr
constr:;  Constructors for block MADD2
;  Constructors for block MUL2
        retsu

; Operation section
        .def operators
operators:
; Head of block MADD2
MADD2:

; Node 5,scheduled at 0
        .data
```

```
ptraMADD25:     .word   VAR2
ptrbMADD25:     .word   VAR1
ptroMADD25:     .word   VAR3
        .text
        ldi     @ptraMADD25,ar0
        ldi     @ptrbMADD25,ar1
        ldi     @ptroMADD25,ar2
        ldi     3,rc
        rptb    ll0
        mpyf3   *ar0++(1),*ar1++(1),r1
ll0:    stf     r1,*ar2++(1)
; Node 6,scheduled at 8
        .data
ptraMADD26:     .word   VAR0
ptrbMADD26:     .word   VAR3
ptroMADD26:     .word   VAR4
        .text
        ldi     @ptraMADD26,ar0
        ldi     @ptrbMADD26,ar1
        ldi     @ptroMADD26,ar2
        ldi     3,rc
        rptb    ll1
        addf3   *ar0++(1),*ar1++(1),r1
ll1:    stf     r1,*ar2++(1)
        retsu
; End of block MADD2
; Head of block MUL2
MUL2:

; Node 5,scheduled at 0
        .data
ptraMUL25:      .word   VAR4
ptrbMUL25:      .word   VAR1
ptroMUL25:      .word   VAR6
.text
        ldi     @ptraMUL25,ar0
        ldi     @ptrbMUL25,ar1
        ldi     @ptroMUL25,ar2
        ldi     3,rc
```

```
        rptb    ll2
        mpyf3   *ar0++(1),*ar1++(1),r1
ll2:    stf     r1,*ar2++(1)

; Node 6,scheduled at 8
        .data
ptraMUL26:      .word   VAR0
ptrbMUL26:      .word   VAR6
ptroMUL26:      .word   VAR7
        .text
        ldi     @ptraMUL26,ar0
        ldi     @ptrbMUL26,ar1
        ldi     @ptroMUL26,ar2
        ldi     3,rc
        rptb    ll3
        mpyf3   *ar0++(1),*ar1++(1),r1
ll3:    stf     r1,*ar2++(1)
        retsu
; End of block MUL2
; Head of block ROOT
ROOT:

; Probe 1
        ldi     VAR0,ar0
        call    probe_ROOT1

; Probe 2
        ldi     VAR1,ar0
        call    probe_ROOT2

; Probe 3
        ldi     VAR2,ar0
        call    probe_ROOT3
; Calling block MADD2
        call    MADD2

; Probe 10
        ldi     VAR5,ar0
        call    probe_ROOT10
```

```
; Calling block MUL2
        if      VAR5
        call    MUL2
        endif

; Probe 7
        ldi     VAR7,ar0
        call    probe_ROOT7
        retsu
; End of block ROOT
        retsu

; Destructor section
        .def    destr
destr:
;  Destructors for block MADD2
;  Destructors for block MUL2
        retsu
```

3.4 Conclusions on the Rafael project

Well, when I say a last farewell to the Rafael project I think I could be a little less formal. The project was launched with ambitious aims keeping in sight the existing systems. As it turned out quite soon, Rafael cannot compete in complexity with the most advanced systems partly because of the limited capabilities of the host computer we chose, partly because of the significantly less human resources we could devote to the project. The final product, the compiler itself has been implemented but many support programs that would make its usage convenient have not been even planned. For this reason the actual Rafael system is not so "user-friendly". As all the resources were concentrated on the compiler development, important parts of the system have not achieved yet the necessary level. The most important among them is the operation database that contains only about a dozen operations only for the TMS320C30 and DSP96002 DSPs. A brave user of Rafael must face the immediate task of filling up the database which requires Lisp programming. Lisp is considered a difficult language among the users although the simple functions needed by the compiler core should be easy to implement for a bit more experienced programmer.

Two distinct influences can be discovered in the Rafael design. The first one is Lee's synchronous dataflow approach and the Gabriel system which gave us the first notions, how Rafael should look like. We quickly faced, however, the need of run-time decisions and the difficulties it causes in a system based on synchronous dataflow. The second influence that we embedded into Rafael was the way the synchronous language compilers work and SynDEx transformes their output to distributed code. Critique of the SynDEx approach was given and a model that was easy to implement to an existing synchronous dataflow system was developed and realized. Limits of this model were pointed out but we consider that in many practical cases, notably in the DSP case they are acceptable. Further researches are conducted to find a better way for handling dynamic structures in a dataflow system.

So Rafael project achieved its aims at the following points :

- A flexible multi-target SDF compiler has been realized on PC platform.
- Effective scheduling algorithms have been developed for the heterogeneous case.

Rafael still has a long way to go at the following fields :

- More user-friendly environment. (graph editor, database editor tools, etc.)
- Complete databases for various DSP processors.

I leave this work, however, to others ...

Bibliography

[Amag94] T. Amagbegnon, L. Besnard and P. Le Guernic, "Arborescent Canonical Form of Boolean Expressions," *INRIA Research Report No. 2290*, June 1994.

[Benv91] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems", *Proceedings of IEEE,* Vol. 79, No. 9, September 1991.

[Berr83] G. Berry, S. Moisan and J-P. Rigault, "Esterel: Toward a synchronous and semantically sound high level language for real time applications," in *Proc. IEEE Real Time Systems Symp.,* 1983.

[Berr93] G. Berry, S. Ramesh and R. K. Shyamasundar, "Communicating Reactive Processes," *Proc. 20th. ACM Conf. on Principles of Programming Languages,* Charleston, VA, 1993.

[Bhat94a] S. S. Bhattacharyya *Compiling Dataflow Programs for Digital Signal Processing,* Ph. D. dissertation, University of California at Berkeley, 1994.

[Bhat94b] S. S. Bhattacharyya and E. A. Lee, "Memory Management for Dataflow Programming of Multirate Signal Processing Algorithms," *IEEE Transactions on Signal Processing,* Vol. 42, No. 5, pp. 1190-1201, May 1994.

[Bic91] L. Bic, M. D. Nagel and J. M. A. Roy, "On Array Partitioning in PODS," *Advanced Topics in Dataflow Computing*, ed. L. Bic and J.-L. Gaudiot, Prentice Hall, 1991.

[Bour94] P. Bournai, C. Lavarenne, P. Le Guernic, O. Maffeïs, Y. Sorel, "Interface SIGNAL-SynDEx," *INRIA Research Report No. 2206,* March 1994.

[Bous91] F. Boussinot and R. de Simone, "The ESTEREL Language," *Proceedings of IEEE,* vol. 79, No. 9, pp. 1293-1303, September 1991.

[Buck91] J. T. Buck, S. Ha, E. A. Lee and D. Messerschmitt, "Multirate Signal Processing in Ptolemy," *Proc. IEEE ICASSP-91,* Toronto, Canada, April 1991.

[Buck93a] J. T. Buck and E. A. Lee, "Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token-flow model," *Proc. IEEE ICASSP-93,* Minneapolis, April 1993.

[Buck93b] J. T. Buck, *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model,* Ph. D. dissertation, University of California at Berkeley, 1993.

[Buck94] J. T. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, "Ptolemy : a Framework for Simulating and Prototyping Heterogeneous Systems", *International Journal of Computer Simulation,* special issue on "Simulation Software Development," January, 1994.

[Chow91] K-W. Chow and B. Liu, "On Mapping Signal Processing Algorithms to a Heterogeneous Multiprocessor System," *ICASSP-91*, pp. 1585-1588. Toronto, 1991.

[Dav78] A. L. Davis, "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine," *Proc. of the Fifth Annual Symposium on Computer Architecture,* pp. 210-215, ACM, April 1978.

[DeMan86] H. De Man, J. Rabaey, P. Six and L. Claesen, "Cathedral-II, A Silicon Compiler for Digital Signal Processing," *IEEE Design & Test*, pp. 13-24, December 1986.

[Denn75] J. B. Dennis, *First Version of a Data Flow Procedure Language,* MAC Technical Memorandum 61, Laboratory for Computer Science, Massachusetts Institute of Technology, May, 1975.

[Denn80] J. B. Dennis, "Data Flow Supercomputers," *IEEE Computer,* Vol. 13, No. 11, November 1980.

[Est88] *Esterelv3 manuals,* Ecole de Mines, Centre de Mathematiques Appliques, Sophia-Antipolis, 1988.

[Gajs92] D. Gajski, N. Dutt, A. Wu and Steve Lin, *High-Level Synthesis.* Kluwer Academic Publishers, 1992.

[Gao92] G. R. Gao, R. Govindarajan, P. Panangaden, "Well-behaved Dataflow Programs for DSP Computation," *Proc. IEEE ICASSP-92,* Vol. 5, pp. 561-564, San Francisco, California, March 1992.

[Gen90] D. Genin, P. Hilfinger, J. Rabaey, C. Scheers and H. De Man, "DSP Specification Using the Silage Language," *ICASSP-90*, pp. 1057-1060, Albuquerque, April 1990.

[Green87] B. Greenblatt and C. J. Linn, "Branch and Bound Style Algorithms for Scheduling Communicating Tasks in a Distributed System," *IEEE Comp. Society Int. Conf. (Compcon-87)*, pp. 12-17, San Francisco, February 1987.

[Gonth88] G. Gonthier, *Semanthiques et Modeles d'Execution des Languages Reactifs Synchrones; Application a ESTEREL,* These de Doctorat en Informatique, Univ. d'Orsay, 1988.

[Goos87] G. Goosens et al, "An efficient microcode compiler for custom multiprocessor DSP systems," *Proc. Int. Conf. CAD,* pp. 24-27, November 1987.

[Halb91] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud, "The Synchronous Data Flow Programming Language LUSTRE," *Proceedings of IEEE,* Vol. 79, No. 9, pp. 1305-1319, September 1991.

[Hoan93] P. D. Hoang and J. M. Rabaey, "Scheduling of DSP Programs onto Multiprocessors for Maximum Throughput," *IEEE Trans. on Signal Processing*, vol. 41, No. 6, pp. 2225-2235, June 1993.

[Hoar85] C. A. R. Hoare, *Communicating Sequential Processes,* Prentice Hall, 1985.

[Hopf85] J. J. Hopfield and D. W. Tank, "Neural computation of Decisions in Optimization Problems," *Biol. Cybern.* 52,141 (1985).

[Hu61] T. C. Hu, "Parallel Sequencing and Assembly Line problems," *Oper. Res.* 9, pp. 841-848, November 1961.

[Hwang93] C. T. Hwang and Y. C. Hsu, "Zone Scheduling," *IEEE Tran. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, No. 7, pp. 926-934, July 1993.

[Kala93] A. Kalavade and E. A. Lee, "A Hardware/Software Codesign Methodology for DSP Applications," *IEEE Design and Test,* September 1993.

[Karp66] R. M. Karp and R. E. Miller, "Properties of a Model for Parallel Computation: Determinancy, Termination and Queueing," *SIAM Journal of Applied Math,* 14(6), pp. 1390-1411, November 1966.

[Kim88] S. J. Kim and J. C. Browne, "A general approach to mapping of parallel computations upon multiprocessor architectures," *ICPP Proc.,* vol. 3, pp. 1-8, Aug. 1988.

[Konst90] K. Konstantinides et al, "Task Allocation and scheduling Models for Multiprocessor Digital Signal Processing," *IEEE. Trans. Acoust., Speech., Signal Processing,* vol. 38, No. 12, pp. 2151-2161, December 1990.

[Kung85] S. Y. Kung, H. J. Whitehouse and T. Kailath, *VLSI and Modern Signal Processing,* Englewood Cliffs, NJ; Prentice Hall 1985, pp. 258-264.

[Lann93] D. Lanneer, *Design Models and Data-Path Mapping for Signal Processing Architectures,* Ph. D. dissertation, Katholieke Universiteit Leuven, March 1993.

[Lear90] K. W. Leary and W. Waddington, "DSP/C : A Standard High Level Language for DSP and Numeric Processing," *IEEE ICASSP-90,* pp. 1065-1068, Albuquerque, New Mexico, April 1990.

[Lee87] E. A. Lee and D. G. Messerschmitt, "Static scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Transactions on Computers*, pp. 25-35, Vol. C-36, No. 1, January 1987.

[Lee89a] J-H. Lee, Y-C. Hsu and Y-L. Lin, "A New Integer Linear Programming Formulation for Scheduling Problem in Data Path Synthesis," *Proc. IEEE ICCAD-89,* Santa Clara, California, November 1989.

[Lee89b] E. A. Lee, W-H. Ho, E. E. Goei, J. C. Bier and S. Bhattacharyya, "Gabriel: A Design Environment for DSP," *IEEE Trans. on*

Acoustics, Speech and Signal Processing, vol. 37. No. 11, November 1989.

[LeG86] P. Le Guernic, A. Benveniste, P. Bournai and T. Gautier, "SIGNAL: A data-flow oriented language for signal processing," *IEEE Trans. Acoust., Speech, Signal Processing,* vol. ASSP-34, pp. 362-374, 1986.

[LeG91] P. Le Guernic, T. Gautier, Michel Le Borgne and Claude Le Maire, "Programming Real-Time Applications with SIGNAL," *Proceedings of IEEE,* vol. 79, No. 9, pp. 1321-1336, September 1991

[Madi94] V. K. Madisetti and B. A. Curtis, "A Quantitative Methodology for Rapid Prototyping and High-Level Synthesis of Signal Processing Algorithms," *IEEE Trans. on Signal Processing*, vol. 42, No. 11, November 1994.

[March95] H. Marchand, E. Rutten and M. Samaan, "Specifying and Verifying a Transformer Station in SIGNAL and SIGNALGTI," *IRISA Internal Publication No. 916*, March 1995.

[Mess84] D. G. Messerschmitt, "A tool for structured functional simulation," *IEEE J. Selected Areas of Communication,* vol. SAC-2, Jan. 1984.

[Mirch88] G. Mirchandani and D. D. Ogden, "Experiments in Partitioning and Scheduling Signal Processing Algorithms for Parallel Processing," *Proc. IEEE ICASSP-88* pp. 1690-1693, New York, April 1988.

[Mitt93] M. Mittler and P. Tran-Gia, "Performance of a Neural Net Scheduler used in Packet Switching Interconnection Networks," *IEEE ICNN-93,* pp. 695-700, San Francisco, California, March 1993.

[Oust90] J. K. Ousterhout, "Tcl: An Embeddable Command Language," *USENIX Conference Proceedings,* Winter, 1990.

[Oust91] J. K. Ousterhout, "An X11 Toolkit Based on the Tcl Language," *USENIX Conference Proceedings,* Winter, 1991.

[Pall95] G. Paller and C. Wolinski, "Springplay : A New Class of Compile-Time Scheduling Algorithm for Heterogeneous Target Architec-

tures," *IFAC/IFIP Workshop on Real Time Programming*, Fort Lauderdale, November 1995.

[Pang87] B. M. Pangrle and D. D. Gajski, "Slicer: A state synthesizer for intelligent silicon compilation," *Proc. IEEE Int. Conf. on Computer Design,* October 1987.

[Pars87] B. Kamgar-Parsi and B. Kamgar-Parsi, "An Efficient Model of Neural Networks for Optimization," *Proc. IEEE ICNN-87,* Vol. 3, pp. 785-789, San Diego, California, June 1987.

[Paul89] P. G. Paulin and J. P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's," *IEEE Trans. on Computer Aided Design,* vol. 8, No. 6, pp. 661-679, June 1989.

[Plot81] G. D. Plotkin, "A structural approach to operational semantics," Lectures Notes, Aarhus Univ., 1981.

[Pow92] D. B. Powell, E. A. Lee and W. Newman, "Direct Synthesis of Optimized Assembly Code from Signal Flow Block Diagrams," *IEEE ICASSP-92,* vol. 5, pp. 553-556, San Francisco, California, March 1992.

[Rabe93] L. Rabelo, Y. Yih, A. Jones and G. Witzgall, "Intelligent FMS Scheduling Using Modular Neural Networks," *IEEE ICNN-93,* pp. 1224-1229, San Francisco, California, March 1993.

[Ritz92] S. Ritz, M. Pankert and H. Meyr, "High Level Software Synthesis for Signal Processing Systems," *Proc. Int. Conf. on Application-Specific Array Processors,* pp. 679-693, Berkeley, California, August 1992.

[Ritz93] S. Ritz, M. Pankert, V. Zivojnovic and H. Meyr, "Optimum Vectorization of Scalable Synchronous Dataflow Graphs," *Proc. Int. Conf. on Application-Specific Array Processors,* pp. 285-296, Venice, Italy, October 1993.

[Sark89] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors,* Cambridge, MA: M.I.T. Press, 1989.

[Sih93a] G. C. Sih and E. A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Archi-

tectures, *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, No. 2, pp. 175-187, February 1993.

[Sih93b] G. C. Sih and E. A. Lee, "Declustering : A New Multiprocessor Scheduling Technique," *IEEE Trans. on Parallel and Distributed Systems,* vol. 4, No. 6, pp. 625-637, June 1993.

[Sor94] Y. Sorel, "Massively Parallel Computing Systems with Real Time Constraints - The "Algorithm Architecture Adequation" Methodology," *Proc. Massively Parallel Computing Systems,* Ischia, May 1994.

[Spox88] *SPOX Programming Reference Manual,* Spectron Microsystem, 1988.

[Tseng86] C. Tseng and D. P. Siewiorek, "Automated Synthesis of data paths in digital systems," *IEEE Trans. Computer-Aided Design,* vol. CAD-5, pp. 379-395, July 1986.

[Virt93] "Virtuoso SSP", *ISI News,* No. 11, September 1993.