Automatic generation of schedulers in the framework of the PAGODE system

Lucile Cognard, Monique Mazaud

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Automatic Generation of Schedulers in the Framework of the PAGODE System

Lucile COGNARD
Monique MAZAUD

N° 1950
Juin 1993

PROGRAMME 2

Calcul symbolique,
programmation
et génie logiciel

Automatic generation of schedulers in the framework of the PAGODE system

---

Génération automatique de réordonnanceurs d'instructions dans le cadre du système PAGODE

Lucile COGNARD and Monique MAZAUD
INRIA Rocquencourt
Domaine de Voluceau, BP 105
78153 Le Chesnay Cedex
FRANCE

Résumé

PAGODE est un constructeur de générateurs de code qui produit les différents moteurs (sélecteur d'instructions, module d'ordonnancement de code et allocateur de registres) d'un générateur de code de facon modulaire. Ce rapport présente d'une part les caractéristiques de la machine cible qui permettent de produire un réordonnanceur, d'autre part les heuristiques utilisées par le noyau de ce moteur.

Mots-Clés : génération de code, sélecteur d'instructions, allocation de registres, ordonnancement de code

Abstract

PAGODE is a back-end generator which produces automatically the various engines of a code generator (instruction selector, scheduler and register allocator) from a target machine specification. This report mainly focuses on the features of the target machine which aim at producing the scheduler, and on the heuristics used by the kernel of the scheduler.

Keywords: code generation, instruction selection, register allocation, pipeline, code scheduling

Contents

List of Figures

0.1 Introduction

Compilers must perform many kinds of code optimizations to effectively exploit the power of modern processors, in particular at the back-end level. One of them is called instruction scheduling. Its aim is to exploit the fine grain parallelism possibilities offered by architectures based on the RISC approach. These machines provide for concurrent instruction execution. The simplest and most common way to achieve this is to use a *pipeline*: each instruction is carried out by several stages and takes as many cycles to complete, but an instruction can enter the pipeline at every cycle and proceed concurrently with the one(s) issued before. Newer processor are based on the *superscalar* approach, in which the processor contains several independent functional units which can be used at the same time by different instructions. In this report we concentrate on pipelines architectures, leaving superscalar ones for future research.

The role of the instruction scheduler is to reorder the instructions in a program so as to maximize concurrency—and hence minimize execution time—while respecting its semantics. This problem has received considerable attention in the past ten years.

The purpose of the present work is to study this problem in the context of a code generator generator. The various engines of the back-end (such as instruction selector and register allocator) are generated automatically from an accurate description of the machine. We wish to extend such a machine description so as to be able to automatically generate instruction schedulers. The main problem is to determine which machine features are relevant to scheduling and hence must appear in the description, and how they can be used to drive a scheduler and the heuristics it uses.

0.1.1 Pipeline technique and scheduling

Pipelining is an implementation technique whereby multiple instructions overlap during execution. Each step in the pipeline completes a part of an instruction. Each of these steps is called a pipe stage.

The throughput of the pipeline is determined by how often an instruction exits the pipeline. Because the pipe stages are hooked together all the stages must be ready to proceed at the same time. The time required between moving instructions down the pipeline is one machine cycle. This is shown in figure 0.1.

IF is the first stage of an instruction execution :
- fetch and decode the instruction

ID is the second one : - read operands
EX is the thirth one : - execution
WB is the last one : - write back stage

There are situations, called *hazards*, which prevent the next instruction in the instruction stream from executing during its designated clock cycle. In that case, a hardware interlock occurs, i.e. the execution of the next instruction is suspended. There are three classes of *hazards*:

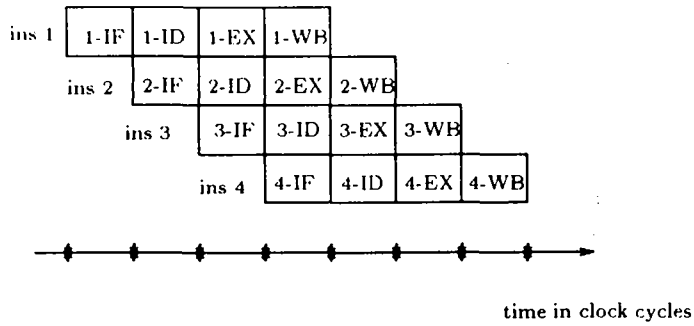


Figure 0.1: Throughput of a pipeline.

1. Data hazards arise when the second instruction reads or writes a datum not yet available.

2. Structural hazards arise from resource conflicts when the hardware cannot support all possible combinations of pairs of instructions in simultaneous overlapped execution.

3. Control hazards arise from dependencies on branches and other instructions which change the program counter.

0.1.2 The PAGODE scheduler

Currently, the generated scheduler works within a basic block on a list of instructions. Such a list may generate various hazards and doesn't take advantage of all the machine abilities. To optimize the execution speed, the scheduler must focus on the arrangement of the instructions in order to decrease the number of hazards and therefore improve program execution time. It is done provided that the read or write access order on the data of the initial sequence stays the same after this arrangement. A dag which defines a partial order which must be preserved during the scheduling.

There is a strong interaction between scheduling and register allocation since :

- if the register allocator is called first, it produces many dependencies which may lower the performance reachable from pipelining;

- on the contrary, if the scheduler is called first, it increases the live range of the registers.

The solution chosen in the PAGODE system is the following: a scheduling step is called first on virtual registers called temporaries[1], then the register allocation step is performed; and finally a new pass of scheduling is called, since register allocation may have produced spill sequences which introduce new dependencies that must be handled.

[1]See section 1.10, of the PAGODE reference manual [CCDM 93].

The input to the scheduler is a low level intermediate representation (denoted LIR in the sequel) including canonical representation of the instructions which may have actual registers or temporaries as parameters of their operands. The memory is viewed as one uniform resource; furthermore, since a main hypothesis is that there exists no alias analysis, a single keyword MEM denotes it.

The current scheduler of PAGODE runs on each basic block : it is an adaptation of a list-scheduler. It relies on the Gibbons and Muchnik algorithm [GM 86] which gives a cover of the "data dependence graph". The data dependence graph is built from the initial sequence. In that graph the nodes represent the instructions, the edges are decorated with the minimal number of cycles such that the second instruction can be issued after the first without data hazards. Our heuristic gives the highest priority to the instruction whose successors in the graph have a hazard, in order to attempt early to fill up the free cycles. The chosen instruction must have the largest number of successors, in order to open the greatest number of paths inside the graph.

A new sequence which must comply with the original semantics is built from a topological sort of the data dependence graph in order to obey precedence data constraints. Instructions are considered one by one and added to the end of the new sequence when no structural hazards with previously issued instructions exists. If no instruction satisfies this condition, a nop is emitted.

Branch hazards are avoided by filling up the wait cycles with independent instructions in the current block or one of the following blocks.

Pipeline management is entirely done by the PAGODE scheduler; this means that nop instructions are generated to avoid interlocks.

Throughout this document, examples are taken from the CYPRESS SPARC specification.

0.2 Data hazard issues

They arise between two instructions when the second instruction reads or writes a datum not yet used or produced by the first one.

0.2.1 Target machine specification

The delay table

There are three classes of data hazards : RAW (*Read After Write*), WAW (*Write After Write*), and WAR (*Write After Read*). In fact the RAW hazards are the most frequent on our target machines. The computer manufacters have encouraged the integration of forwarding processes. Forwarding is a hardware mechanism to reduce the number of RAW hazards. The second instruction can catch the result before the first one completes. This information can be summarised by a compact, global table, available for any kind of target machines. Without it, it would be necessary to consult informations like read or write access cycles, which are dispatched on all instructions. The structure of this table comes from the machines comparisons of [LS 92]. In PAGODE , it is called

Delay_table. An array element of this table is the minimal delay which must be satisfied to avoid this hazard between two instructions I_1 and I_2.

Let us consider two instructions I_1 and I_2 such that:

- I_2 follows I_1 in the semantic order of execution
- R is a register which is written in I_1 and read in I_2.

Kind of write access of R in I_1	Kind of read access of R in I_2	Use in ALU	Use in address	Use in store	Use in cc
	Result of Load	2	2	2	2
	Result of ALU	1	1	2	1
	Side-effect	0	0	0	0

Figure 0.2: Delay table for the SPARC.

The delay table presented in figure 0.2 has three lines which correspond to the kind of write access of R in I_1:

1. the value written in the register R is the result of a load.
2. the value written in the register R is the result of an arithmetic or logical computation.
3. the value written in the register R is the result of a computation occurring in an addressing mode with side-effect (e.g. auto-increment).

The delay table has four columns which correspond to the kinds of read access of R in I_2:

1. the value read in the register R is used for an arithmetic or logical computation.
2. the value read in the register R is used for an address computation.
3. the value read in the register R is used for a store in the memory.
4. the value read in the register R is used for a condition code computation.

Delay_table = $((2,2,2,2)\,(1,1,\mathbf{2},1)\,(0,0,0,0))$ is the SPARC specification in SCALA .

The value **2** means that I_1 is an arithmetic instruction which writes the register R and I_2 a store of the contents of R in memory and that I_2 must begin at least **2** cycles after I_1. Since the delay-table was general and has constant dimensions and since there exists no addressing mode with side effect on the SPARC, the last four "0" stand for undefined values.

Some instructions do not need any table search because their execution is longer and without bypass. Table search is only done when the boolean **$use_delay_table** is set to TRUE on a specific instruction. The default value is FALSE. So the delay-table is used only for some RAW data hazards.

Read and write stages

In order to deal with other hazards RAW, WAW and WAR, the code generator author must specify two attributes on each instruction which is not a branch one:

- The stage number where the source operands are read is denoted by the attribute **$init_read_cycle**. On a peculiar instruction if this action needs more than one stage, the first stage of this step must be considered since the PAGODE system assumes that fetching all the source registers begins at the same time.
- **$end_write_cycle** denotes the stage number where the result operand is written. If this action requires several cycles, the last cycle of this action must be specified.

In order to deal with branch instructions, the PAGODE system assumes that the read stage is the same for all branches. The value of the global variable **$Branch_init_read_cycle** is the stage number where the operands of the branch instructions are read.

The scheduler must be able to deal with target machines like i860 which uses addressing modes with side effects where the write back step (WB) does not execute at the same time as the side effect stage; the PAGODE system assumes that all the instructions using addressing mode with side effect have the same side-effect stage.

If an instruction uses a register occurring in an addressing mode with side effect, the scheduler looks into the Delay_table to avoid RAW data hazards on this register. WAR and WAW hazards can be managed using the value of the side effect stage which is denoted by the attribute **$side_effect_write_cycle** on each related instruction.

Let us at this point look at two examples:

- in the integer addition, the use of the Delay_table is specified.

```
Instruction
    Canonical_form
        add (<Gpr_access_W AM1>, <Reg_or_imm_access AM2>
                , <Gpr_access_W AM3>)
    Attributes
        ...
        $end_write_cycle   =   4
        $init_read_cycle   =   2
        $use_delay_table   =   true
        ...
    Template
        assign_W (add_32 ...
End
```

- on the contrary for the floating-point instruction *fdivd*, the **$use_delay_table** is not specified, it means that it is not used and a computation on the

read and write cycles must be done to know the delay. This computation is explained in detail later.

```
Instruction
    Canonical_form
        fdivd (<Freg_access_D AM1>, <Freg_access_D AM2>
              , <Freg_access_D AM3>)
        Attributes
              ...
              $end_write_cycle =   41
              $init_read_cycle =   3
              ...
        Template
              assign_D (fdivd ...
    End
```

The lack of specification of **$use_delay_table** in fdivd, means that the value is FALSE and that it is not necessary to access into the **Delay_table**.

0.2.2 Data dependency graph construction

A data dependency graph is built from the three kinds of dependencies, which occur in the input to the scheduler.

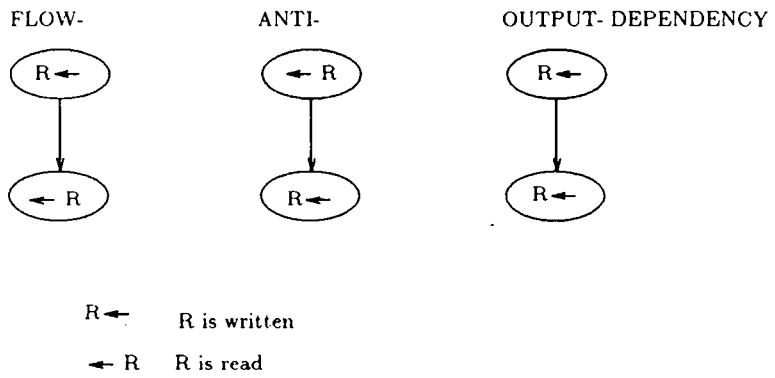
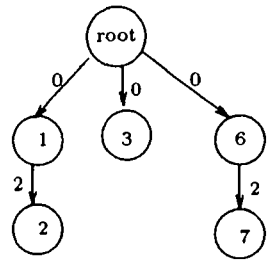


Figure 0.3: Kinds of data dependencies.

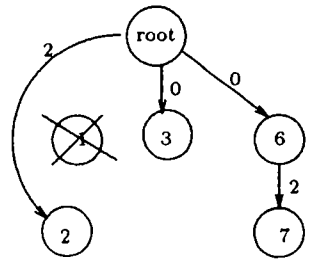
In the dag, each node stands for an instruction. There exists an edge between two nodes when the two related instructions use a common resource which can be either an actual register or a temporary or the memory. The minimal value k_{min} such that no hazard occurs between i and j decorates the edge $i \rightarrow j$. In fact, the k_{min} value is deduced from the delay table when $\$use\_delay\_table$ is true, or from a computation on the values of $\$init\_read\_cycle$ and $\$end\_write\_cycle$ of i and j otherwise. This computation is based on the equations given in section 2.2.1.

This DAG is modified by the scheduling process. The *root* node denotes the starting node of the sequence, and the current point of the new sequence after the start. The processed nodes are deleted. On each node which represents a potential hazard between *root* and this unprocessed node, the scheduler updates the k value. Each step of the scheduling process does so.

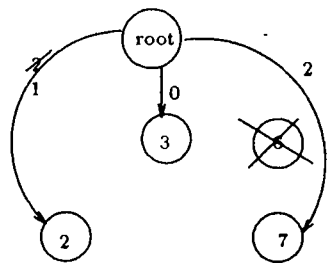
On the following dag, instructions 1, 3 and 6 may be issued since the delay from the beginning of the sequence (*root*) is 0.



If node 1 is chosen, it is necessary to draw an edge between *root* and the nodes which have no more predecessors. Thus the decoration of the edge *root* $\rightarrow$ 2 is the same than the previous decoration on the edge 1 $\rightarrow$ 2 because node 1 has replaced the initial root.



If node 6 is chosen, it is also necessary to update the decorations on previous edges with *root*. Since the throughput of the pipeline is one instruction per cycle, one can say that issuing instruction 6 takes one cycle, then the edge *root* $\xrightarrow{2}$ 2 is replaced by the edge *root* $\xrightarrow{1}$ 2.



Using read and write cycles

Let us consider two instructions i and j, such that j follows i in the initial sequence. k denotes the number of cycles produced by the pipelined execution between the i start cycle and the j start cycle. R denotes a resource used by both instructions. Conditions for no hazard to occur are explained in the following; when the inequation is true, there is no problem.

A flow data dependency is an other name for RAW, an anti data dependency for WAR, and an ouput data dependency for WAW.

1. There exists a flow data dependency between i and j when R is written during the execution of i (written by i) and read by j. There is no hazard on it when the read stage of R in j is executed after the write stage of R in i, i.e if:

$$\$init\_read\_cycle_j + k \geq \$end\_write\_cycle_i$$

2. There exists an anti data dependency between i and j when R is read by i and written by j. No hazard occurs on it when the write stage of R in j is executed after the read stage of R in i, i.e if:

$$\$end\_write\_cycle_j + k > \$init\_read\_cycle_i$$

3. There exists an output data dependency between i and j when R is written by both instructions. There is no hazard on it when the write stage of R in j is executed after the write stage of R in i, i.e if:

$$\$end\_write\_cycle_j + k > \$end\_write\_cycle_i$$

The validity of the first case (RAW) is illustrated in the figure 0.4. It corresponds to the two following assumptions:

- i is an instruction which computes a result and writes it in register R in the fourth stage denoted by *end_write_cycle*. It follows:

$$\$end\_write\_cycle_i = 4$$

- j_1 and j_2 are two instructions which read the register R in the second stage denoted by *init_read_cycle*. It follows:

$$\$init\_read\_cycle_{j_1} = 2$$
$$\$init\_read\_cycle_{j_2} = 2$$

Let us denote k, the number of cycles between start of both instructions which can clash. If k equals 1, j_1 is an immediate successor of i. j_1 reads the register R in cycle 3 but the register R has not been written yet by instruction i (it is done in cycle 4 only): this situation is called a RAW data hazard.

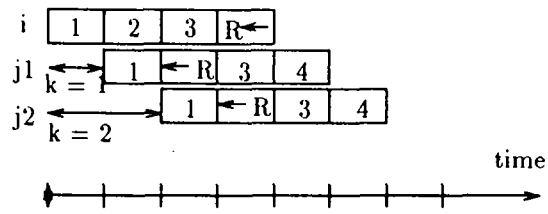


Figure 0.4: Flow dependency.

Let us consider the execution of i and j_2. One can say that this hazard doesn't occur remembering that the register file is accessed in both halves of a clock cycle. Then it is possible to do the register writes in the first half of WB and the reads in the second half of ID. [HP 90] consider that is a general feature, illustrated in figure 0.5.

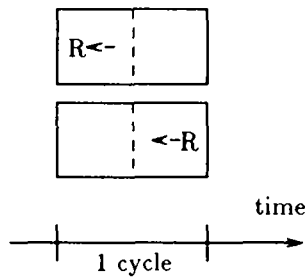


Figure 0.5: Decomposed cycle.

The minimal distance which must be observed between two instructions i and j is denoted k_{min}. k_{min} is a decoration of the edge between i and j. Its value can be computed by the functions RAW(i, j, R), WAW(i, j, R) and WAR(i, j, R), which corresponds respectively to the various cases of data dependencies which may occur between the instructions i and j.

The validity of the two other inequations (WAR and WAW) can be justified by a similar presentation.

If the register R is not accessed via an addressing mode with a side effect, the previous functions can be defined as following:

$$
\begin{aligned}
\text{WAW}(i,j,r) = \quad & \$end\_write\_cycle_j - \$end\_write\_cycle_i \\
\text{WAR}(i,j,r) = \quad & \$end\_write\_cycle_j - \$init\_read\_cycle_i \\
\text{RAW}(i,j,r) = \quad & \text{if } use\_delay\_table_i = \text{true} \\
& \qquad Delay\_table\ [\ \$write\_kind(r,i),\ \$read\_kind(r,j)\ ] \\
& \text{else} \\
& \qquad \$init\_read\_cycle_j - \$end\_write\_cycle_i + 1
\end{aligned}
$$

Delay table access

In this section, we will explain the table searching and its application context. Let us consider the delay table information: the lines depict three cases of write access on R in i. The columns depict four cases of read access on R in j. The delay table access is done using two table indexes $\$write\_kind(r, i)$ and $\$read\_kind(r, j)$.

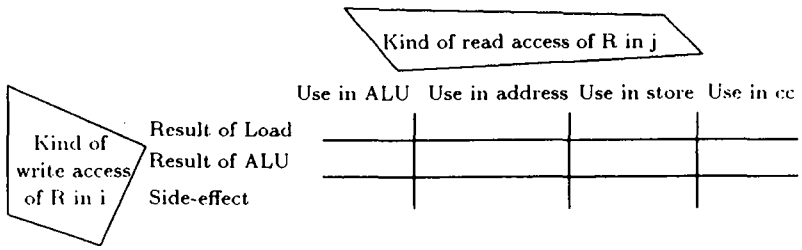


Figure 0.6: Delay table structure.

Both indexes can be determined using the *$modification*[2] information which describes the instruction, the *side effect* information, the *use*, *def* and *uses-for-adr* sets which describe the sets of resource which are used, resources which are defined, and the resources used for an address computation.

The first index can be determined when we know the i instruction and the register R, in order to determine which is the case of write access (i.e: $\$write\_kind(R, i)$).

Write access of R	**Symbol table entry condition**
Result of load	$\$modification = load$ **and** $R \in def$
Result of ALU or SU	$\$modification = arith$ **and** $R \in def$
Side-effect	($\$modification = load$ **or** $\$modification = store$) **and** $side\text{-}effect = true$

The second index can be determined when we know the j instruction and R. Since we know R, a unique line of the delay table is searched, and the column must be found. (i.e: $read\_kind(r, i)$). The four cases are mutually exclusive and a computation on the value and attributes previously mentioned allows to find the appropriate array element.

[2]See section 1.8.1 of the PAGODE reference manual [CCDM 93].

Use access of R	Symbol table entry condition
Use in ALU or for SU	$\$modification$ = arith **and** $R \in use$
Use for an address computation	($\$modification$ = cond-branch **or** $\$modification$ = branch **or** $\$modification$ = store **or** $\$modification$ = load) **and** $R \in$ *uses-for-adr*
Use for a store	$\$modification$ = store **and** $R \in use$ **and** $\neg(R \in$ *uses-for-adr* $)$
Use for a condition computation	($\$modification$ = cond-branch **or** $\$modification$ = test) **and** $R \in use$ **and** $\neg(R \in$ *uses-for-adr* $)$

For example, if the j instruction is *Store R1, a(R6)*, thus R = R1 and it is a case of use in a store, determined by:

If *$modification = store* **Then**
    If R **belongs-to** *uses-for-adr* **Then**
        use-in-address
    else *use-in-store*
    end if
else ...

0.3 Structural hazard issues

0.3.1 Target machine specification

These hazards are managed by means of a booking table. It is necessary to list all the functional units which are components of the processor and to specify the maximal number of uses in the same clock cycle. A **Interlock_table** must be specified. It includes the list of all functional units which appear in the programmer's model. The maximal number of uses without structural hazard is related to each of them.

For instance, on the SPARC, let us denote DATA_BUS and ADR_BUS the data bus and the address bus. READ_REG_PORT and WRITE_REG_PORT denote the read port and the write port on the *register file*. ALU, SU, FALU, et FMDCSU denote respectively the Arithmetic and Logic Unit, the Shift Unit, the Floating point Arithmetic and Logic Unit, the Floating point Multiply Divide Compare and Square-root Unit. PSR and FSR are respectively the Processor Status Register and the Floating point Status Register.

The interlock table specification is the following for the SPARC:

Interlock_table

DATA_BUS : 2
ADR_BUS : 1
READ_REG_PORT : 1
WRITE_REG_PORT : 1
DECODE : 1
ALU : 1

```
SU              :  1
FALU            :  2
FMDCSU          :  2
PSR             :  1
FSR             :  1
PROGRAM_COUNTER :  1
```

A table called **$Booking_table** must be specified for each instruction. It includes the list of stages which are necessary to execute the instruction. For each of them, the functional units which are used are merely listed using the same formalism as in the **Interlock_table**. A stage which uses no fonctional unit is only numbered and kept empty. The stage number may be followed by an asterisk(*), which means that in the execution context of the instruction a shift of one or more cycles can be introduced on it and its following stages. This will be explained later.

For instance in the following **$Booking_table** attribute, the first stage uses both ADR_BUS and DATA_BUS. The second stage uses both DECODE and READ_REG_PORT.

```
$Booking_table =   (1 : ADR_BUS / / DATA_BUS ;
                    2* : DECODE / / READ_REG_PORT ;
                    3 : ALU ; 4 : WRITE_REG_PORT / /
                    ADR_BUS / / DATA_BUS)
```

A critical point to detect structural hazards is to connect the stages of instructions in the pipe during execution. In a clean execution model, all the instructions have the same number of cycle and all stages are normally in the pipe cycle per cycle. On more realistic execution models, the number of cycle and as a consequence the feed can change. A solution to manage this is provided by PAGODE , its name comes from the corresponding SPARC feature, but its significance is more general.

The Cypress manufacturer implements a peculiar hardware system of pipeline execution using the Internal Op-code feature (IOP). It is a hardware mechanism which handles multi-cycle instructions. The aim is to differ data bus access for either loading a data or loading an instruction to execute. Multi-cycle instructions are usually loads and stores. They induce a shift of one or more clock cycles in some stages of some following instructions.

On the SPARC architecture, there exist three kinds of multi-cycle instructions. They define the number of IOPs on each of them and the behaviour of each of them:

- A double-cycle instruction has only one IOP, i.e the two instructions which follow it are delayed one cycle. That is the case of the "ldsb" specified below. "Single loads", "jump" and "Return from Trap" belong to the family of double cycle instructions. Their behaviour in the pipe is shown in figure 0.7. Arabic numbers denotes instructions and roman numbers denotes stage numbers. The notation *1-iop* means that the cycle is used by the IOP of the double-cycle instruction 1.

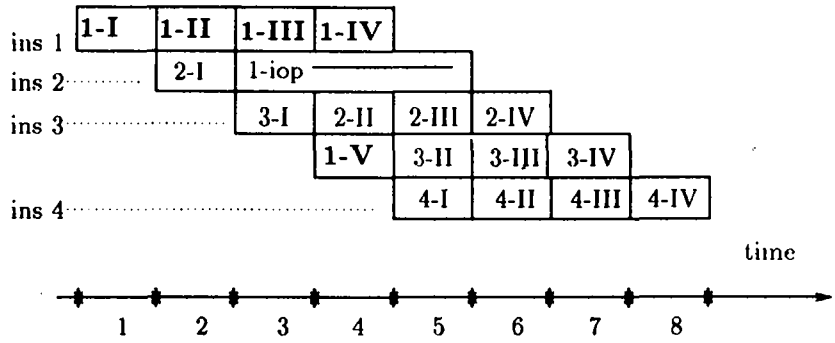


Figure 0.7: State of the pipeline with internals op-codes.

The example shows the 5 cycles of instruction 1, the shift of 1 cycle inside instructions 2 and 3, and the normal execution of instruction 4.

- A triple-cycle instruction has two IOPs, i.e the two instructions which follow it are delayed two cycles. "Single stores" and "Double loads" belong to that family.
- A quadruple-cycle instruction has three IOP, i.e the two instructions which follow it are shifted out of three cycles. "Double stores" and "atomic Load-Store" belong to that family.

The PAGODE language provides two attributes to take those features into account:

- **$IOP_cycle_shift** is the length of the delay (number of clock cycle) on the following instructions.
- **$IOP_patched_ins** is the number of following instructions which bear this delay.

Notice that the sequence of instructions producing IOPs induce a cascade of shifts that must be manage by the scheduler. The main issue is that the IOPs are only known at the decode stage, and that stage itself can be under a previous shift. The number of IOPs is currently limited to three IOPs per instruction, since we have not encountered more sophisticated cases in the target machines we modeled.

The following specification is a whole instruction. It is a double-cycle instruction.

Instruction
  Canonical_form
    ldsb (<Gpr_access_W AM1>, <Gpr_access_W AM2>)
  Attributes
    $length = W
    $fmt = ~ ldsb **$fmt** (<Gpr_access_W AM1>)
      ,**$fmt** (<Gpr_access_W AM2>) ~

```
        $Booking_table      =   (1 : ADR_BUS / / DATA_BUS ;
                                 2* : DECODE / / READ_REG_PORT ;
                                 3 : ALU ; 4 : WRITE_REG_PORT / /
                                 ADR_BUS / / DATA_BUS)
        $IOP_cycle_shift    =   1
        $IOP_patched_ins    =   2
        $end_write_cycle    =   4
        $init_read_cycle    =   2
        $use_delay_table    =   true
    Template
        assign_W (sign_extend (right_justify (src (<All_access_W AM1>)))
                , dst (<Gpr_access_W AM2>))
End
```

The second stage denoted with an asterisk and its following stages can be shifted by a previous IOP.

0.3.2 SPARC declaration examples

The various features dealing with data hazards and structural hazards can be summarized by the "add" instruction on general purpose registers and the floating-point divide instruction.

- addition instruction :

```
Instruction
    Canonical_form
        add (<Gpr_access_W AM1>, <Reg_or_imm_access AM2>
                , <Gpr_access_W AM3>)
    Attributes
        ...
        $Booking_table      =   (1: ADR_BUS / / DATA_BUS ;
                                 2* : DECODE / / READ_REG_PORT ;
                                 3 : ALU ; 4 : WRITE_REG_PORT)
        $end_write_cycle    =   4
        $init_read_cycle    =   2
        $use_delay_table    =   true
    Template
        assign_W (add_32 ...
End
```

- Floating-point divide instruction :

```
Instruction
    Canonical_form
        fdivd (<Freg_access_D AM1>, <Freg_access_D AM2>
                , <Freg_access_D AM3>)
    Attributes
        ...
        $Booking_table      =   (1 : ADR_BUS / / DATA_BUS ;
                                 2* : DECODE ; 3 : READ_REG_PORT ;
                                 4* : READ_REG_PORT ;
```

```
                                    5 → 40 : FMDCSU ;
                                    41 : FMDCSU / / WRITE_REG_PORT)
        $end_write_cycle    =   41
        $init_read_cycle    =   3
    Template
        assign_D (fdivd ...
End
```

Stages 5 to 40 are all identical.

0.3.3 Time representation for structural hazards

During the scheduling process, it is necessary to check whether a given instruction issued after a given sequence will cause a structural hazard. To achieve that purpose, the functional units which are involved in the pipe are connected together for all cycles of this new instruction. This connection is done by projection of the functional units on the time axis. The usage number of each functional unit per cycle is compared with the maximal usage allowed by the processor (information which can be retrieved from the *Interlock_table* global table) This test is necessary only for cycles which overlap with the newly added instruction. In figure 0.8, the initial sequence consists of the two instructions *And* and *Or*, and the scheduler tries to issue a new *Add*, checking whether it induces a structural hazard.

At cycle 3 : number-of(Adr-Bus) = $1 \leq 1$ (the value returned by the *Interlock-table* for Adr-Bus) and number-of(Data-Bus) = $1 \leq 2$ given, and number-of(Read-Reg) = $1 \leq 1$, and number-of(ALU) = $1 \leq 1$, then no structural hazard happens at this cycle.

A similar computation is done for the cycles 4, 5 and 6, then no hazard occurs, the sequence And-Or-Add does not need to be rearranged.

Now, let us consider 1)- a sequence already scheduled *seq* which induces a $shift_{seq}$ on its immediate successor j 2)- an instruction j which induces $shift_j$ cycles.

Then if j is added to *seq*, on any l following the j instruction, the stages are delayed $shift_{seq} + shift_j$ cycles.

As a consequence, the shifts induced by IOPs are contextual; they must be handled in the scheduling algorithm. Figure 0.9 shows that instruction 1 induces a delay of 3 cycles on the three following instructions. Instruction 2 again induces a delay of 2 cycles on the two following instructions. These information came from the target machine specification. Finally the instruction 3 come under the shift of instruction 1 and instruction 2 and the global effect is $3 + 2 = 5$ shifts.

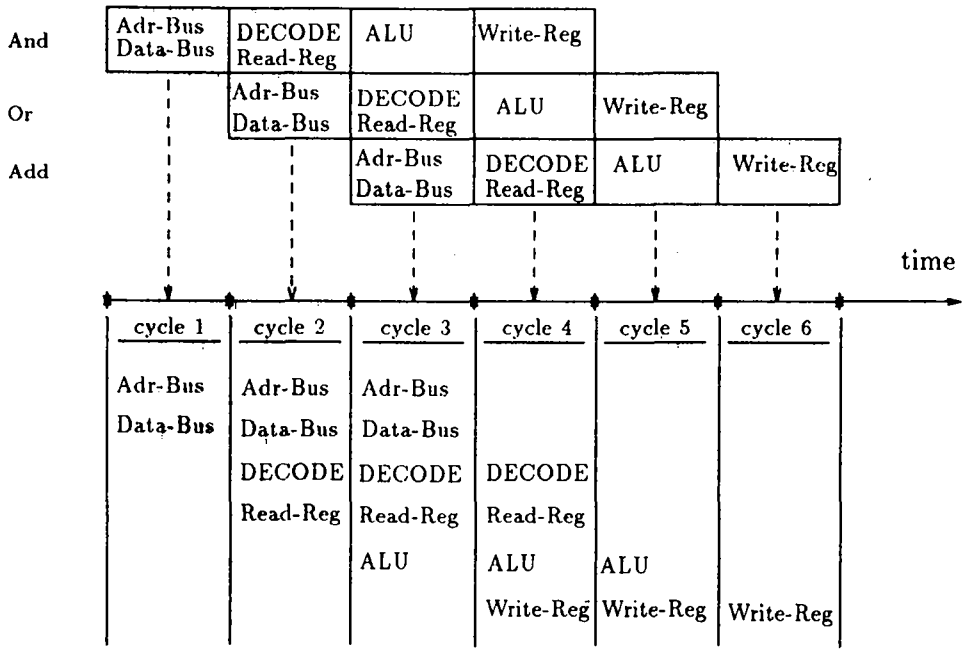


Figure 0.8: Functional units usage projection.

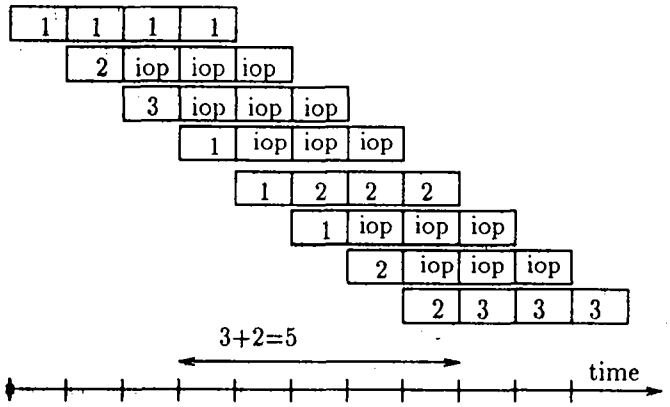


Figure 0.9: Nested shifts induced by IOPs.

0.4 Algorithms and heuristics for data and structural hazards

0.4.1 Introduction

From the target machine specification, the PAGODE constructor builds an AG for FNC-2 [JP 90]. The evaluation of the AG achieves the scheduling step. The syntactic part of the AG is related to the access modes and the instructions. The offsprings of each instruction rule are decorated by attributes which come from the pipe specification. The offsprings of access mode rules are decorated by attributes which define local data-flow analysis on registers, temporaries and memory references.

One can say that the scheduling engine mainly performs the evaluation of four phases of the AG. Those steps are mainly directed by the attribute rules on the sequence operator which roots a list of LIR instructions.

1. Instructions information gathering for a given basic block.
2. When the last instruction of a basic block is read, the dag is built using the information of step 1. The values k_{min} are computed and serve as decorations on the edges of the DAG.
3. The scheduler is called on a given basic block, it updates the DAG and finally returns a new tree as output in the same syntactic form than the input tree.
4. When the scheduling of a basic block is done, a new gathering information step is performed for a the next another basic block. The basic block barriers are found by the PAGODE list-scheduler itself using the $\$modification$ attribute which determines the family of a LIR instruction. It means that the branch instructions are included into the basic blocks.

0.4.2 Heuristics

The following heuristics are called to remove from the dag an instruction to issue :

1. **Candidates** $\leftarrow$ the instructions which can be issued at this point
   (without data hazard, without structural hazard between
   the begin of the sequence and these instructions)
   If empty(Candidates), nop is emitted and return to 1.
2. **First-selection** $\leftarrow$ $\{x \in$ Candidates such that
   $\exists\ y$ such that $(x, y) \in$ DAG and $k_{min}(x, y) > 0\}$
   The candidates which have a data hazard with one of
   their sons.
3. **Second-selection** $\leftarrow$ $\{x \in$ First-selection such that
   $\neg\exists\ y \in$ Second-selection such that
   number-of-children(y) > number-of-children(x) $\}$
   Among First-selection choose those that have the highest number of sons.

4. **Third-selection** $\leftarrow$ $\{x \in$ Second-selection such that
   $\neg\exists\ y \in$ Third-selection such that
   $\forall\ z,\ w$ such that $(x,z) \in DAG,\ (y,w) \in DAG$
   $k_{min}(y,w) > k_{min}(x,z)$ $\}$
   Those of Second-selection which have the latest data hazard with one of their sons.
5. **Fourth-selection** $\leftarrow$ $\{x \in$ Third-selection such that
   $\neg\exists\ y \in$ Fourth-selection such that
   max-path-length(y) < max-path-length(x) $\}$

max-path-length(x) is the maximal number of nodes to reach a terminal node of the dag from x.

0.4.3 Examples

The input to the scheduler is the basic block shown in figure 0.10. Data hazards are depicted by nops. Computing the data dependencies results in the dag shown in figure 0.11, where each node is decorated with its *use*, and *def* sets.

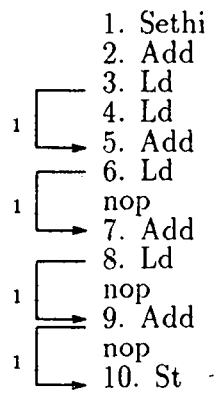


Figure 0.10: Initial list of instructions.

RAW(x) = y means that the value k_{min} related to the potential data hazard on resource x is equal to y. Similar properties hold for WAR(x) = y and WAW(x) = y. When there exist several dependencies on an edge, the highest value is kept for k_{min}.

Let us show the computation of RAW(x), WAW(x) and WAR(x) in figure 0.12.

Running the scheduling algorithm to emit the first LIR instruction of the new sequence, the following steps of the algorithm are performed :

1. **Candidates** $\leftarrow$ $\{1,3,6,8\}$
   If empty(Candidates), nop is emitted and return to 1.

2. **First-selection** $\leftarrow$ $\{3,6,8\}$
   Those of the candidates which have a data hazard with one of their sons.

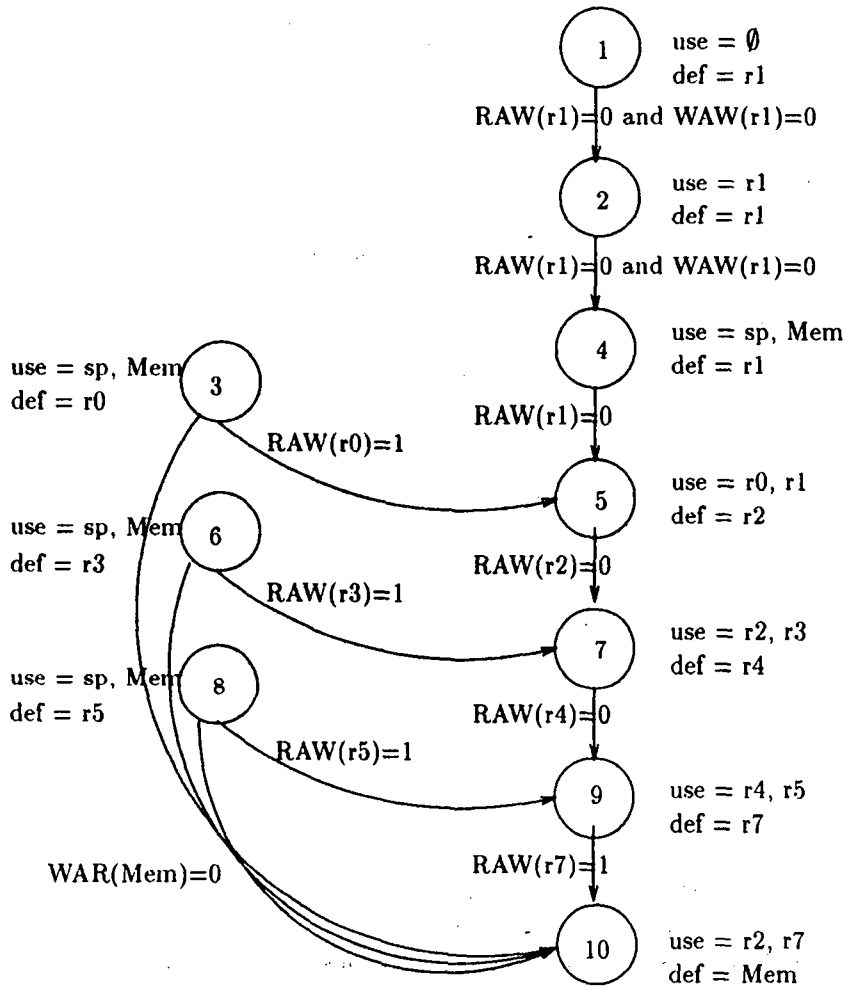
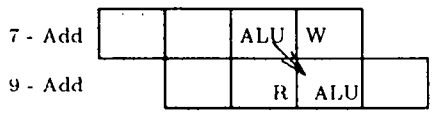


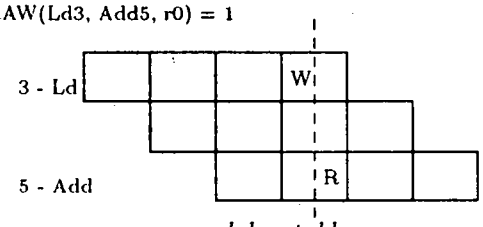
Figure 0.11: Dag of the initial list of instructions.

- RAW(Add7, Add9, r4) = 0



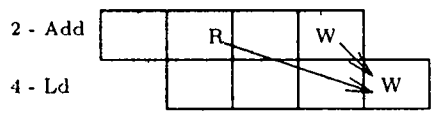
Result of the *delay-table* searching.

- RAW(Ld3, Add5, r0) = 1



Result of the *delay-table* searching.

- WAW(Add2, Ld4, r1) = 0 and WAR(Add2, Ld4, r1) = 0



Using the *end_write_cycle* and *init_read_cycle* values.

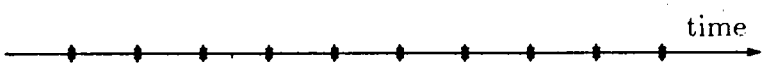


Figure 0.12: Delay of data hazards.

3. **Second-selection** $\leftarrow \{3,6,8\}$
   Those of First-selection which have the highest number of sons.

4. **Third-selection** $\leftarrow \{3,6,8\}$
   Those of Second-selection which have the highest data hazard with one of their sons.

5. **Fourth-selection** $\leftarrow \{3\}$
   Those of Third-selection which have the longest path to a terminal node. Issue one node of the Fourth-selection set and return to step 1.

After reorganization, the sequence shown in figure 0.13 is obtained.

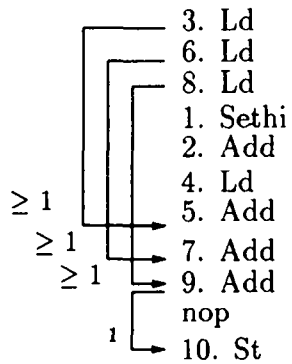


Figure 0.13: Result of scheduling.

The two pictures in figure 0.14 show what happens in the pipe for a part of a new sequence. They show how to manage the IOP's context. One must point out that figure A is intuitive but false, since a structual hazard occurs in it that does not exit in fact. There is a cycle with two stars. i.e. two Adr_bus access, while this bus cannot sustain more than one access. The first stage is denoted I(i), the second II(i) and so on.

0.5 Control hazard issues

A control hazard arises when a branch updates the program counter with a new address and the processor must wait until the new value of the program counter is ready to fetch the next instruction.

0.5.1 Target machine specifications

The delay is called branch slot. In order to fill up this cycle with independent instructions, it is necessary to know the size of the branch slot. The optimizer can be more efficient if the scheduler knows whether the slot instructions will be executed whether the branch is taken or not. We assume that the two following informations on branches are available :

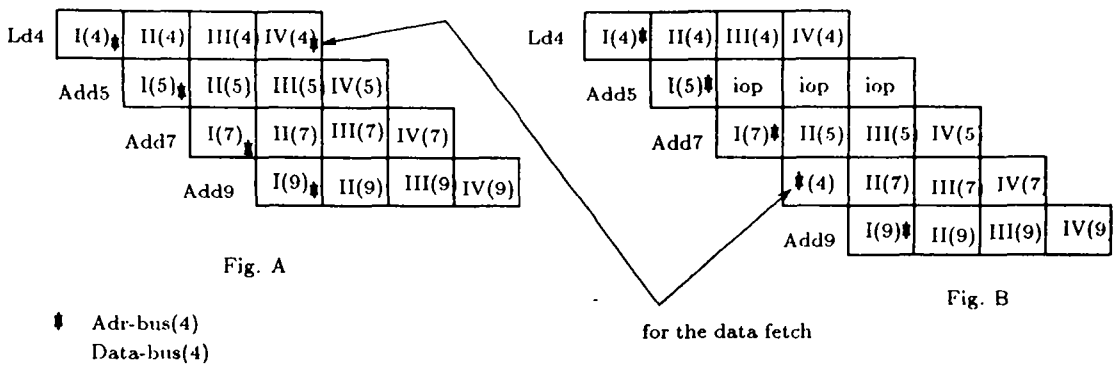


Figure 0.14: Context management.

- The size in number of instructions of the slot is called **$delay_slot**. In most cases its value is 1.
- The behaviour of the instructions which fill up the slot is depicted by **$executed_slot**. Four values are allowed :
  1. *always*: the instructions are always executed. This case happens on all machines for unconditional branches.
  2. *taken*: the instructions are only executed when the branch is taken. This case can occur on conditionals branches and is irrelevant for unconditional branches. It occurs on the SPARC.
  3. *untaken*: the instructions are only executed when the branch is not taken. This case can occur on conditionals branches and is irrelevant for unconditional branches. It occurs on the i860 and M88000 but not on the SPARC.
  4. *never*: the instructions are never executed. This case happens rarely and in very peculiar situations. It may occur for the SPARC.

The branch instruction "bne,a" with "annul" bit field "a" can be specified as follows :

```
Instruction
    Canonical_form
        bne (<Relative_access AM>)
    Attributes
    $length =   W
    $fmt    =   ~ bne,a   $fmt (<Relative_access AM>) ~
    $Booking_table      =   (1 : ADR_BUS / / DATA_BUS ;
                             2* : DECODE / / PSR / / PROGRAM_COUNTER)
    $executed_slot      =   taken
    $delay_slot         =   1
Template
```

branch_if_ne (dst(<Relative_access AM >))
End

0.5.2 Implementation of control hazard issues

It relies on Thomas Gross' and John L. Hennessy's algorithm [GH 82]. It fills up the slot with the following assumptions on the conditional branch : if the target precedes the branch, then it probably comes from a loop case, with a high probability to be taken. If the target is after the branch, it is rather an if-then-else and then the chances to be taken or untaken are similar.

Notations and conditions

Let us denote i a conditional branch, B_i the block which ends with i, t^+ the target block, and t^- the next block.

In order to fill up the delay slots following the branches according to this algorithm, the following information must be known or computable :

1. the control flow graph of the LIR,
2. the DAG of each basic block,
3. the set IN of the resources which are read in the basic block before they are written.
4. the size of the branch slot.

The following set is built from these data :

$$\text{F-IN}(B_i) = \text{IN}(B_i) \cup (\bigcup_{B_j \in Follow(B_i)} \text{IN}(B_j))$$

where $Follow(B_i)$ is the set of basic blocks which can follow B_i in an execution sequence. One can consider the ways in which the branch delay can be scheduled.

The branch delay is filled up by copy or by move, using instructions which exist somewhere in the sequence. The three following strategies are correct provided that in their new position they do not cause new hazards with the previous or the next instrutions.

1. An instruction of the block B_i is moved in the slot under the condition that there is no flow dependency between this instruction and the branch one. This means that moving this instruction does not update the condition code on the address computation.
2. This strategy is preferred when the branch is taken. An instruction of t^+ is copied in the slot provided that it does not update the memory and that its target register does not belong to the F-IN(t^-) set. These conditions preserve the execution state if the branch is in fact not taken.

3. The last strategy is preferred when the branch is not taken. An instruction of t^- is moved in the slot provided that it does not update the memory and that its target register does not belong to the F-IN(t^+) set. These conditions preserve the execution state if the branch is in fact taken.

An instruction which is chosen to be moved must comply with the partial order in the dag of its basic block. In the first optimization, the instruction needs to have no successors; on the contrary in the third optimization the instruction must have no predecessors. The dag is updated while the sequence is changed by these various optimizations.

Heuristics

Knowing the breakdown between taken and not taken branches is important because this will affect strategies for filling up the branch delay. Some results in [HP 90] show that a backward branch has a high probability to be a loop case, and as consequence to be a branch-taken case. Otherwise (forward branch) the probability is around 50%.

Optimization 1 is preferred first under any behaviour of the conditional branch because it is safe and the slot is filled up without increasing the size of the code.

Otherwise, for the previous reasons, it is preferable to fill up the branch delay by the second optimization for a backward branch, and by the third for a forward branch because it does not increase the size of the code, and finally by the optimisation 2 if it fails.

The *$executed_slot* attribute has the following effect on the algorithm; in the branch-taken case, it is sufficient to copy the instructions pointed by the label without checking that the registers which are written does not belong to F-IN. The reason is that instructions are only executed if the conditional branch is taken. For similar reasons, it is sufficient to move the instructions of the following block in the branch-not-taken case.

Figures 0.15, 0.16 and 0.17 show how the semantics of the initial sequence can be preserved by generating new labels.

0.6 Automatic generation of data-gathering rules for a list-scheduler

The PAGODE scheduler generation mainly generates an attribute grammar in the special purpose language OLGA designed for the description of all aspects of an attribute grammar in the FNC-2 system.

This of course involves constructs to declare attributes and access them in semantic rules. OLGA is a strongly typed language which supports the notion of modules, in which one can define a set of related objects.

Thus one can mainly distinguish three kinds of generation tasks:

1. the generation of OLGA constants,

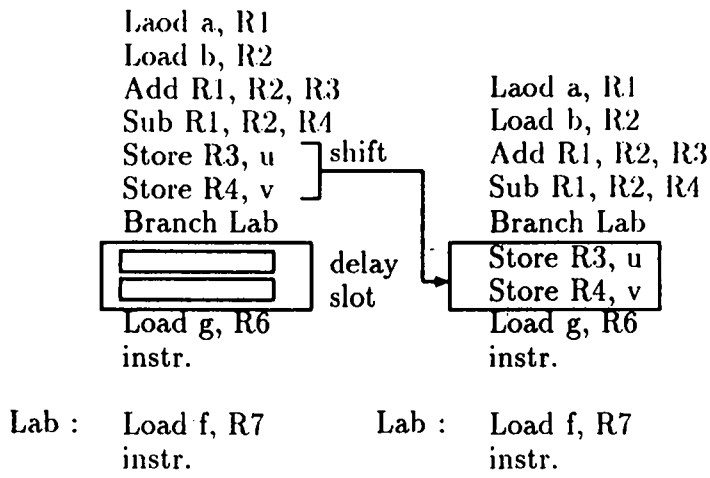


Figure 0.15: Optimisation 1.

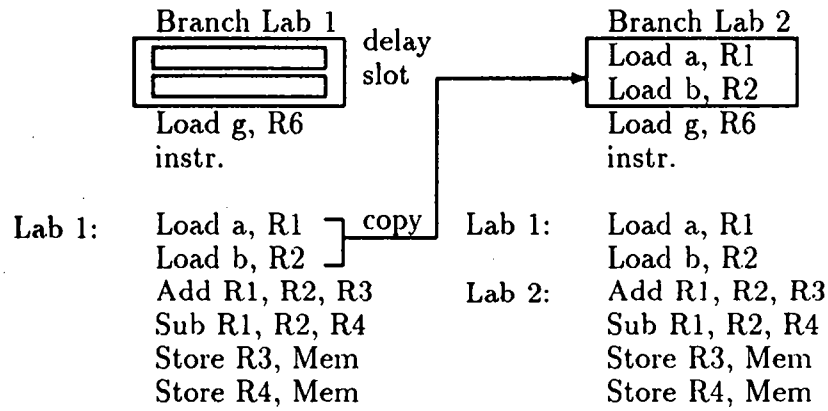


Figure 0.16: Optimisation 2.

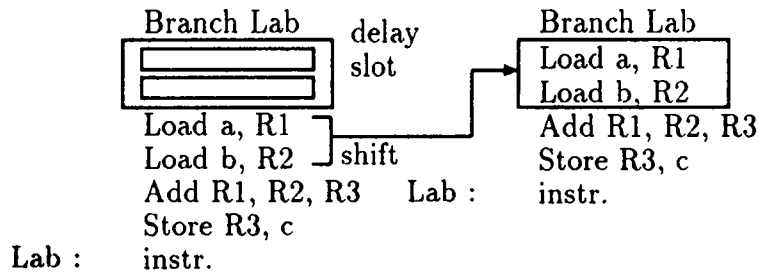


Figure 0.17: Optimisation 3.

2. the generation of the semantics rules related to the LIR instructions,

3. the generation of the semantics rules related to the LIR access modes.

Among the constant declarations which can be found in an OLGA file for the list-scheduler, one can find the Delay_table, the Interlock_table and the Branch_init_read_cycle declarations. For instance the Delay_table declaration in the SCALA language:

$$\textbf{Delay\_table} = ((2,2,2,2)\,(1,1,2,1)\,(0,0,0,0))$$

is translated into the OLGA declaration:

```
delay-table := tab-tab-delay(
        tab-delay(2,2,2,2)
        tab-delay(1,1,2,1)
        tab-delay(0,0,0,0))
```

The semantic rules related to the instructions compute attributes which are a straightforward translation of the SCALA attributes into their specification in the OLGA formalism. That is the case of the translation of the SCALA attributes: $modification, $Booking-table, $init-read-cycle, $end-write-cycle, and optionally $IOP_cycle_shift, $IOP_patched_ins, $side_effect_write_cycle.

```
Instruction
    Canonical_form
        add (<Gpr_access_W AM1>, <Reg_or_imm_access AM2>
                , <Gpr_access_W AM3>)
    Attributes
        ...
        $modification      =   arith
        $Booking_table     =   (1: ADR_BUS / / DATA_BUS ;
                                2* : DECODE / / READ_REG_PORT ;
                                3 : ALU ; 4 : WRITE_REG_PORT)
        $end_write_cycle   =   4
        $init_read_cycle   =   2
        $use_delay_table   =   true
    Template
        assign_W (add_32 ...
End
```

is translated into the OLGA declaration:

```
Where add → Operand1, Operand2, Operand3 use
    $modification := arith ;
    $Booking-table := tab-book[|
      box-book(1,false,tab-UFs[|ADR_BUS,DATA_BUS|]),
      box-book(2,true, tab-UFs[|DECODE,READ_REG_PORT|]),
      box-book(3,true, tab-UFs[|ALU|]),
      box-book(4,true, tab-UFs[|WRITE_REG_PORT|])|] ;
    $init-read-cycle := 2 ;
```

```
    $end-write-cycle := 4 ;
    $use-delay-table := true ;
    $destination(Operand1):= src ;
    $destination(Operand2):= src ;
    $destination(Operand3):= dst ;
end where ;
```

Attributes related to IOP features are initialized to 0 when they do not appear in the SCALA specification. Similary the $use-delay-table attribute is set to false when its declaration does not appear in the SCALA specification of the instruction.

Furthermore the position of each operand of an instruction is stored into the $destination attribute which takes either the *src* value when it is in source position in the template of the instruction or *dst* in the template of the instruction.

The *use* and *def* sets related to an instruction are built from attributes which compute for each access mode the sets of resources which are read and the set of resources which are written by a given access mode.

Among these resources, one can find either the memory (denoted by the unique name MEM) or the temporaries or the actual registers which may appear in the LIR input to the scheduler. The PAGODE constructor knows whether an access mode corresponds to a memory reference via an analysis of the template of this access mode declared in the SCALA specification.

Let us consider the indirect with displacement access mode for the SPARC:

```
Access_mode
    Canonical_form              - - Indirect with displacement access mode
        disp_am!size ( <gpr_W reg> , <value_13 val>)
    Attributes
        $length      =    size
        $fmt         =    ~ reg + val ~
        $space_cost  =    0
        $time_cost   =    1
        Template
            index ( cont_of_gpr_W (<gpr_W reg>)
                  , const_value_13 (<value_13 val>))
End
```

The PAGODE constructor derives from this specification two templates[3].

- One in source position:

```
cont_of_address!size(index(cont_of_gpr_W(<gpr_W reg>),
                           const_value_13(<value_13 val>)))
```

- One in destination position:

```
designates_address!size(index(cont_of_gpr_W(<gpr_W reg>),
                              const_value_13(<value_13 val>)))
```

[3] see section of the PAGODE reference manual 1.6.2 [CCDM 93].

Since the dereference operator (respectively the cell constructor operator) returns an address, the PAGODE sytem deduces that this access mode references the memory.

Let us recall that there must exist in the target machine specification a storage class called address based on the storage base MEM (both address and MEM are keywords of the PAGODE system).

Finally the following semantic rules are generated for an access mode such as disp_am which references the memory:

```
where disp_am → gpr value_13 use
    $s.read := if (($destination = src) or ($destination = both))
               then  tab-resources(tab-resource($h.read, MEM), $ident(gpr))
               else  tab-resources($h.read, $ident(gpr))
               end if ;

    $s.write := if (($destination = src) or ($destination = both))
               then  tab-resource($h.write, MEM)
               else  $h.write
               end if ;
end where ;
```

When the access mode references a register or a temporary, the following rules are generated:

```
where gpr_am → gpr use
    $s.read := if (($destination = src) or ($destination = both))
               then  tab-resources($h.read, $ident(gpr))
               else  $h.read
               end if ;

    $s.write := if (($destination = src) or ($destination = both))
               then  tab-resource($h.write, $ident(gpr))
               else  $h.write
               end if ;
end where ;
```

On each related access mode, an attribute rule is generated in order to add the register or the temporary in the set of resources which are used to compute a memory reference *uses-for-adr.*

```
Where disp_am - > gpr value_13 use
        $s.uses-for-adr := tab-resources( $h.uses-for-adr, $ident(gpr))
end where;
```

0.7 Integration of scheduling

In the classical framework, the scheduler is the final pass of the code generator which is called to avoid stalls of the processor. It is normally called after the

register allocator and just before the code emitter. In that case, the register allocator produces many dependencies and the scheduler is hampered. On the contrary, if the scheduler is called first, it changes the live range of the resources and usually lengthens them.

Basically, we chose to integrate these two engines according to the following basic framework (see figure 0.18):

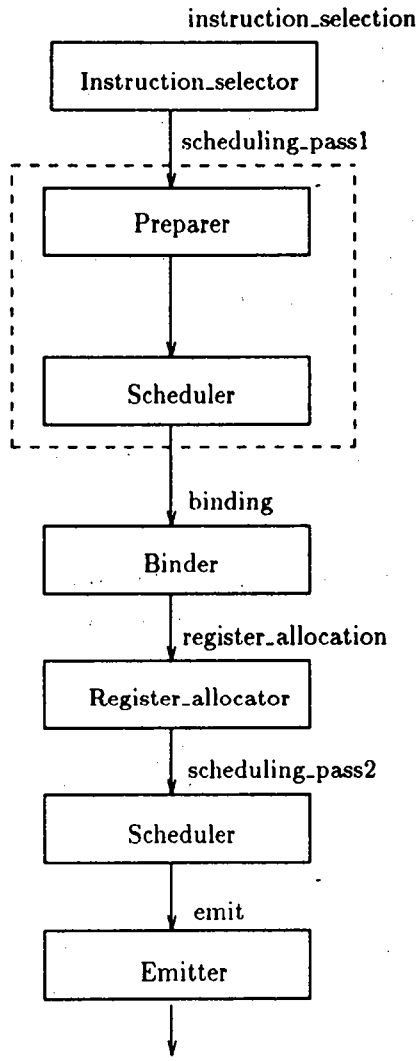


Figure 0.18: The overall structure of a RISC back-end.

1. A *first pass of scheduling* works on the term obtained as output of the prepare engine after the instruction selection step.
2. Then *register allocation* is performed.
3. Since the register allocator produces spill code, i.e new sequences which

have not been dealt by the scheduler and which can induce new data dependencies with the previous sequence, a *second pass of scheduling* is necessary.

Before the first pass of scheduling, universal stores (Univ_assign operators) and temporaries still occur in the LIR. Basically they correspond to several degrees of freedom for the corresponding access modes. A basic goal of the PAGODE system is to keep these degrees of freedom as long as possible, and especially until the register allocator so that this last engine can take advantage of them.

A main consequence of this choice is that no scheduling can be done on the Univ_assign operators, because they cannot be related to any other load, store or move instruction. That is the reason why three pseudo-instructions must be specified by the compiler writer, they must be called **Univ_load**, **Univ_store**, **Univ_move**, that are three keywords of the PAGODE system.

These three pseudo-instructions are specified as real instructions such that the behaviour of the Univ_load is the average behaviour of the loads of the processor from the point of view of the pipelined execution. The Univ_store is specified similarly regarding the stores of the processor. Finally, the Univ_move is specified regarding the moves of the processor.

Thus, the prepare engine rewrites the Univ_assign operators into either Univ_load or Univ_store or Univ_move. This overloading mechanism strongly relies on the fact that a temporary_am access mode depicts only an access to a resource. It also uses an information built by the PAGODE constructor which tells whether or not an access mode is a memory reference. Since the first operand of an Univ_assign is the source operand and the second one the destination operand, the following algorithm is applied :

- if the first operand is a memory reference the Univ_assign operator is overloaded by a Univ_load,
- if the second operand is a memory reference the Univ_assign operator is overloaded by a Univ_store,
- otherwise, the Univ_assign operator is overloaded by a Univ_move.

The prepare engine is then called after the instruction selector and before the first pass of scheduling.

Currently, the code generators produced by the PAGODE system run sequentially, performing the following steps after instruction selection :

1. The prepare engine overloads the Univ_assign operators of the term built by the instruction selector.
2. A first pass of scheduling works on the term obtained from the prepare engine.
3. The binding step is performed. It computes the live range sets for the register allocator.

4. Register allocation is performed.

5. Univ_load, Univ_store and Univ_move operators are overloaded by true operators of the processor by a special purpose engine called "assign_rewriter".

6. Since register allocation produces spill code, i.e new sequences which have not been dealt by the scheduler, a new pass of scheduling is necessary.

7. The code is emitted by the emit engine.

0.7.1 Specification for integration

Transfer instruction specifications

Three instruction specifications related to Univ_load, Univ_store, Univ_move must exist in the target machine specification.

The Univ_load specification is an instruction which has the average behaviour of the instructions of the "load" family. It uses the average length of the pipe. Thus for the SPARC architecture, this specification is the same as that of the "ldd" instruction and for the rest it gets the specifications of the "ld" instruction.

```
Instruction
    Canonical_form
        Univ_load (<All_access_W AM1>, <Gpr_access_W AM2>)
    Attributes
        $length =   {B, H, W, D}
        $fmt    =   ~ ld $fmt(<All_access_W AM1>)
                         ,$fmt(<Gpr_access_W AM2>) ~
        $Booking_table          =   (1 : ADR_BUS / / DATA_BUS ;
                                     2* : DECODE / / READ_REG_PORT ;
                                     3 : ; 4 : WRITE_REG_PORT / /
                                     ADR_BUS / / DATA_BUS
                                     5 : ADR_BUS / / DATA_BUS)
        $IOP_cycle_shift        =   2
        $IOP_patched_ins        =   2
        $end_write_cycle        =   5
        $init_read_cycle        =   2
        $use_delay_table        =   true
    Template
        assign_W (src( <All_access_W AM1>)
                  , extend_long (dst (<Gpr_access_W AM2>)))
End
```

Similarly, the Univ_store specification is a copy of the "st" instruction with pipe attributes which are those of the "std" instruction.

```
Instruction
    Canonical_form
        Univ_store (<Gpr_access_W AM1>, <All_access_W AM2>)
    Attributes
        $length =   {B, H, W, D}
```

```
        $fmt    =   ~ st  $fmt (<Gpr_access_W AM1>)
                          ,$fmt  (<All_access_W AM2>) ~
        $Booking_table        =   (1 : ADR_BUS / / DATA_BUS ;
                                   2* : DECODE / / READ_REG_PORT ;
                                   3 : READ_REG_PORT ;
                                   4 : ; 6 : ADR_BUS / / DATA_BUS/
        $IOP_cycle_shift      =   3
        $IOP_patched_ins      =   2
        $end_write_cycle      =   6
        $init_read_cycle      =   2
        $use_delay_table      =   true
    Template
        assign_W (src (<Gpr_access_W AM1>)
                  , (dst (<All_access_W AM2>)))
End
```

Finally, the Univ_move specification is a copy of the "mov" instruction.

0.7.2 Input and output of the prepare engine

An example is given to illustrate the prepare engine effect. First is the input, and secondly the result.

Input of the prepare engine

```
seq (
    Univ_assign ( disp_am <gpr %fp> <value_13 -36>
    ,       temporary_am <temporary tmp8> )
    Univ_assign ( value_13_am <value_13 12>
    ,       temporary_am <temporary tmp9>)
    Univ_assign ( disp_am <gpr %fp> <value_13 -32>
    ,       temporary_am <temporary tmp10> )
    fmuld ( temporary_am <temporary tmp9>
    ,       temporary_am <temporary tmp10>
    ,       temporary_am <temporary tmp11>)
    add ( temporary_am <temporary tmp8>
    ,       temporary_am <temporary tmp11>
    ,       temporary_am <temporary tmp12>)
    Univ_assign ( temporary_am <temporary tmp12>
    ,       disp_am <gpr %fp> <value_13 -28>)
    ba ( relative_am <code_label l2>)
)
```

Output of the prepare engine

All Univ_assign operators have been overloaded by operators on which there exists attributes for scheduling, i.e Univ_load, Univ_store and Univ_move.

```
seq (
    Univ_load ( disp_am <gpr %fp> <value_13 -36>
    ,       temporary_am <temporary tmp8> )
    Univ_move ( value_13_am <value_13 12>
```

```
    ,       temporary_am <temporary tmp9>)
    Univ_load ( disp_am <gpr %fp> <value_13 -32>
    ,       temporary_am <temporary tmp10> )
    fmuld ( temporary_am <temporary tmp9>
    ,       temporary_am <temporary tmp10>
    ,       temporary_am <temporary tmp11>)
    add ( temporary_am <temporary tmp8>
    ,       temporary_am <temporary tmp11>
    ,       temporary_am <temporary tmp12>)
    Univ_store ( temporary_am <temporary tmp12>
    ,       disp_am <gpr %fp> <value_13 -28>)
    ba ( relative_am <code_label l2>)
)
```

0.7.3 Input and output of the scheduler engine

Let us consider the following LIR term:

```
seq (
    1   ld ( disp_am <gpr_W fp> <value_13 -16>
            , temporary_am <temporary tmp1> )
    2   ld ( disp_am <gpr_W fp> <value_13 -4>
            , temporary_am <temporary tmp2>)
    3   add ( temporary_am <temporary tmp1>
            , temporary_am <temporary tmp2>
            , temporary_am <temporary tmp3>)
    4   add ( temporary_am <temporary tmp3>
            , temporary_am <temporary tmp1>
            , temporary_am <temporary tmp4>)
    5   st ( temporary_am <temporary tmp4>
            , disp_am <gpr_W fp> <value_13 -32>)
    6   cmp ( temporary_am <temporary tmp1>
            , temporary_am <temporary tmp2>)
    7   bg ( relative_am <code_label l1>)
    )
```

The data dependence graph is built from the LIR by the schduler engine and given as input to the list-scheduling algorithm.

The result of the scheduling is the following:

```
seq (
    1   ld ( disp_am <gpr_W fp> <value_13 -16>
            , temporary_am <temporary tmp1> )
    2   ld ( disp_am <gpr_W fp> <value_13 -4>
            , temporary_am <temporary tmp2>)
    3   add ( temporary_am <temporary tmp1>
            , temporary_am <temporary tmp2>
            , temporary_am <temporary tmp3>)
    4   add ( temporary_am <temporary tmp3>
            , temporary_am <temporary tmp1>
            , temporary_am <temporary tmp4>)
    6   cmp ( temporary_am <temporary tmp1>
            , temporary_am <temporary tmp2>)
    5   st ( temporary_am <temporary tmp4>
```

```
        , disp_am <gpr_W fp> <value_13 -32>)
7   bg ( relative_am <code_label 11>)
)
```

The following figure shows what happens in the pipe. Let us denote I, II, III, IV, V and VI the pipeline stages. There exists a delay between the instructions 4 and 5 in the data dependence graph.

The following two dimension table represents pipeline usage. Lines stand for instruction issues and columns stand for clock cycles.

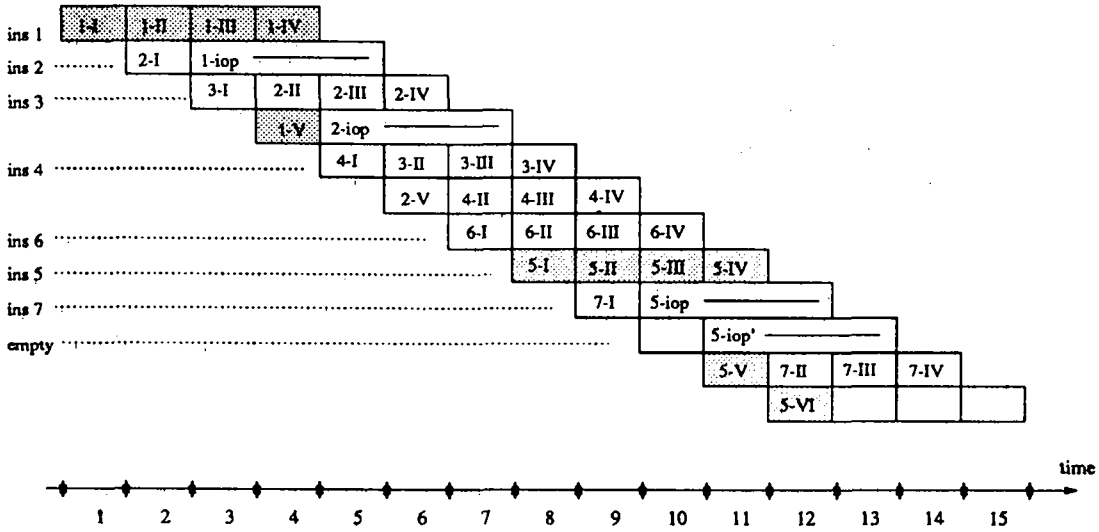


Figure 0.19: State of the pipeline during an execution.

The 5-II stage ought to be executed after or at the same time than the 4-IV. The insertion of instruction 6 avoids a nop.

0.8 Conclusion

The aim of deriving automatically a scheduler from a target machine description has been reached. Such a generation is based on a model of a RISC architecture, which is a pipelined architecture with possible forwarding mechanism and side effect addressing modes. Notice that such architectures may be implemented with hardware mechanisms which induce new behaviours. That is the case of the IOPs in the CYPRESS SPARC. The MIPS RS2000 also causes new hazards related to the special purpose registers HI and LO. The kernel of the scheduling algorithm must be able to deal with such peculiar features.

This work will be extended in order to deal with more advanced architectures; the concepts required to deal with superscalar architectures will probably lead to a new generation of target machine specifications.

Finally, let us conclude on two main points of the current version. First it brings a solution to the issue of the interaction of register allocation and scheduling. Secondly the current version does not perform aliasing analysis but the scheduler is ready to integrate aliasing information which may be produced by a previous step of the compiler.

Bibliography

[BHE 91] D.G. Bradlee, R.R. Henry, and S.J. Eggers: The Marion system for retargetable instruction scheduling. *Proceedings of the ACM SIGPLAN 91, Conference on Programming Language Design and Implementation. Toronto, Ontario, Canada, June 26-28, 1991.*

[CCDM 93] Canalda P., Cognard L., Despland A., Mazaud M.: The PAGODE System User's Guide and Reference Manual Release 1.1 RT 152 *INRIA July 1993.*

[CH 90] Chow F., Hennessy J.L.: The priority-Based Coloring Approach to Register Allocation. *Transactions on Programming Languages and Systems Vol. 12,4,, October 1990, pp501-536.*

[GH 82] Thomas R.Gross, John L. Hennessy: Optimizing Delayed Branches. *Proceedings 15th Annual Workshop on Microprogramming 1982* pp 114-120.

[HG 83] John L.Hennessy, Thomas R.Gross: Postpass Code Optimization of Pipeline constraints. *ACM Transactions on Programming Languages and Systems, Vol.5, No.3, July 1983, pp 422-448.*

[HP 90] Hennessy and Patterson: Computer Architecture A Quantitative Approach. *1990 by Morgan Kaufmann Publishers, Inc.*

[GM 86] Plilip B. Gibbons, Steven S. Muchnick: Efficient Instruction Scheduling for a pipelined Architecture. *Proceedings of the SIGPLAN'86, pp 11-16.*

[JP 90] Jourdan M., Parigot D., Julie C., Durin O., Le Bellec C.: Dessign, Implementation and Evaluation of the FNC-2 Attribute Grammar system. *Proceedings SIGPLAN'90, pp 209-222.*

[LS 92] Laporte P., Seznec A.: Une Etude Comparative des Microprocesseurs MIPS R3000, SPARC Version 7, et IBM Power: Architectures et performances *Cellule d'Evaluation des Microprocesseurs Rapides - Fevrier 1992 - IRISA*

[SKV 92] Seznec A., Kermarrec A-M., Vauleon T.: Etude Comparée des Architectures des Microprocesseurs MIPS R4000, DEC 21064, et

T.I. SUPERSPARC. *Publication interne 692 - Décembre 1992 - IRISA*

Unité de Recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

Unité de Recherche INRIA Lorraine Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 VILLERS LES NANCY Cedex (France)
Unité de Recherche INRIA Rennes IRISA, Campus Universitaire de Beaulieu 35042 RENNES Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 GRENOBLE Cedex (France)
Unité de Recherche INRIA Sophia Antipolis 2004, route des Lucioles - B.P. 93 - 06902 SOPHIA ANTIPOLIS Cedex (France)

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399