INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION INTERNE

N° 1032

# SALTO: SYSTEM FOR ASSEMBLY-LANGUAGE TRANSFORMATION AND OPTIMIZATION

ERVEN ROHOU, FRANÇOIS BODIN, ANDRÉ SEZNEC, GWENDAL LE FOL, FRANÇOIS CHAROT, FRÉDÉRIC RAIMBAULT

ISSN 1166-8687

IRISA

CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 99 84 71 00 – Fax : (33) 99 84 71 71

# SALTO: System for Assembly-Language Transformation and Optimization

Erven Rohou, François Bodin, André Seznec,

Gwendal Le Fol, François Charot*, Frédéric Raimbault**

Thème 1 — Réseaux et systèmes
Projet CAPS

Publication interne n° 1032 — Juin 1996 — 19 pages

**Abstract:** On critical applications, particularly embedded systems, the performance tuning requires multiple passes. SALTO (System for Assembly Language Transformation and Optimization) is a retargetable framework for developing all the spectrum of tools that are needed for performance tuning on low-level codes (assembly-languages) on uniprocessors. SALTO enables the building of profiling, tracing and optimization tools. The user is responsible for giving a machine description of the target architecture, which includes instruction-set of the processor, precise hardware configuration and reservation-tables for all instructions, but high-level functions are provided to him for writing any tool corresponding to his needs.

Moreover SALTO will be a part of a global solution for manipulating assembly-code to implement low-level code restructuration as well as to provide a high-level code restructurer with useful information collected from the assembler code and from instruction profiling.

SALTO has been tested on Intel platforms running Linux (i486) and Solaris (PentiumPro) and on a Sparcstation running SunOs 4.1. A machine description for the Sparc v7 architecture is currently available. Two examples, a basic block instrumentation and a local reordering optimization, are given in the paper as illustration.

**Key-words:** assembly language, optimization, embedded systems, reservation tables, user interface, object oriented, compilation process

*(Résumé : tsvp)*

* {erohou,bodin,seznec,lefol,charot}@irisa.fr
** raimbaul@univ-ubs.fr

Centre National de la Recherche Scientifique
(URA 227) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

# Salto : un système pour la transformation et l'optimisation des codes assembleurs

**Résumé :** Pour la plupart des applications critiques, notamment les systèmes embarqués, l'optimisation des performances requiert plusieurs passes. Salto est un environnement de travail reciblable qui permet le développement de l'ensemble des outils nécessaires pour l'analyse et l'optimisation de performances sur des codes assembleurs pour mono-processeurs. Salto facilite la construction d'outils pour le *profiling*, la génération de traces et l'optimisation. L'utilisateur doit fournir un fichier de description de la machine cible. Cette description, du jeu d'instructions du processeur, contient la configuration matérielle et les tables de réservations décrivant l'usage des ressources. Une interface de haut-niveau, orientée objet, est fournie pour l'écriture des outils dont il a besoin.

En outre, Salto est destiné à être un élément d'une solution globale de manipulation de code assembleur tant pour la restructuration de codes de bas-niveau que pour la production d'information à destination d'un optimiseur de haut-niveau.

Salto a été testé sur des plateformes Intel fonctionnant sous Linux (i486) et Solaris (PentiumPro) et sur Sparcstation sous SunOs 4.1. Une description de l'architecture Sparc v7 est disponible. Deux exemples, une instrumentation des blocs de base et un ordonnancement local, sont décrits dans cet article en guise d'illustration.

**Mots-clé :** assembleur, optimisation de performances, systèmes embarqués, tables de réservation, interface utilisateur, orienté objet, chaîne de compilation

# 1 Introduction

It is our belief that for many critical applications, particularly in embedded systems, the performance tuning of low-level code cannot be handled by a single-pass compiler. We believe that such performance tuning requires the cooperation of many tools including assembly-code schedulers targeted for the precise hardware configuration, tracing and profiling tools and instruction-layout optimizers. These tools should feed information back to the high-level compiler.

The increasing usage of high-performance embedded systems based on RISC/VLIW architectures has highlighted the need for tools that allow the easy implementation of fine-grain parallelism optimizations and assembly-code profiling and instrumentation, along with an accurate description of the target architecture and high retargetability. SALTO, presented in this paper, is a first step toward the availability of such a system.

SALTO is a retargetable framework for developing the whole spectrum of tools that manipulates assembly-language. The objective of such a system is to provide the user with a single environment that will allow him to implement the tools that are needed for performance tuning on low-level codes; this set of tools includes assembly-code schedulers, as well as profiling and tracing tools that provide the user with information on where to focus optimizations and how efficient they can be, therefore allowing tradeoff choices. Such a system is intended to address general computing as well as embedded systems for which optimizations are more critical and aggressive, but time-consuming techniques are more tolerable.

A large number of tools have been written to experiment with new optimizations or to try to point out particular mechanisms. This development phase is generally time consuming and requires much investment. Utilities able to trace or profile programs exist, but they are often provided “as is”: they are not at all flexible, and only work for specific architectures. Studying a particular problem is likely to require rewriting from scratch large pieces of code. As SALTO is retargetable with the instruction-set architecture as well as with the precise target hardware, it is likely to be a major help for such studies.

SALTO overcomes many limitations of previous solutions: it does not implement any algorithm by itself, but its scope is broad enough to allow implementation of either profiling or optimization techniques. A large amount of work needed while working at the assembly-code level (code parsing, construction of the dependence graph, etc.) is performed automatically by the system. To the user, SALTO provides an object-oriented interface to deal with assembly-code. The objects contain a complete description of the control-flow graph of the program (when available) and a model of the target architecture. They are easily accessible through the user interface and provide a comfortable way to implement algorithms without having to worry about infrastructure.

With SALTO we plan to address the field of software analysis and optimization for superscalar and VLIW architectures.

Section 2 reviews related works and points out how our tool provides a more general and integrated solution for building tracers, profilers and optimizers. The general approach leading to SALTO is presented in section 3. Section 4 gives an overview of the system. Section 5 details the features of our tool. Examples are given in section 6.

# 2 Related Work

To our knowledge no available single environment deals with the whole spectrum of code restructuration and execution profiling and tracing. Moreover most tools are not fully retargetable.

Much work has been done in the field of low-level code optimization and analysis of dynamic program execution. Optimization can be done at different levels of granularity. They range from code transformations for improving the parallelism in each basic-block to procedure motion for improving the instruction-cache behavior. Dynamic program execution can be analyzed by profiling or tracing. Profiling deals with the number of times a piece of code (instruction, block or procedure) is executed. This is particularly interesting for determining critical sections in codes. As object-code optimization is CPU consuming, the optimizations should focus only on those sections. Tracing furnishes a more precise analysis: the order of instructions is known and precise addresses are obtained.

## 2.1 Analysis

Many techniques for analyzing program behavior depend on instrumentation of either the source code or the executable file. PIXIE [26] is an instrumentation utility running on MIPS machines for executables. The instrumented program, which contains additional code, counts the execution of each basic-block. A counter is added in each block. Ball and Larus [3] studied how to optimally use the same technique by minimizing the number of counters required. They build a control flow graph of the procedure being instrumented and compute the maximal spanning tree before deciding where to add code. They proved this technique to be optimal. They implemented their strategy in a tool called QPT [3].

A more flexible solution called ATOM [12, 27] was proposed for the DEC Alpha. ATOM also depends on code instrumentation and provides common analysis and performance tools. A partial list given in [12] contains instruction profiling, system-call summary, memory checker, and many others. The major advantage is the ability for the user to implement instrumentations in the C language enabling a high degree of flexibility. ATOM can be seen as a library of predefined functions that ease the instrumentation of the code.

Gordon Irlam's SPY [22] runs on Sparc architectures running SunOs 4.x. SPY exploits special features of the Sparc microprocessor [11] to fetch the instruction to be executed. SPY provides as output a trace composed of the instruction addresses and the data addresses, if any. Other tools achieve analysis via instruction-set emulation. Cmelik and Keppel chose another strategy with SHADE [9]: they *dynamically compile* each instruction of the program, i.e., they build a block semantically equivalent to each assembly instruction of the original code as if it were a complex instruction and then execute the block. This approach enables them to simulate an instruction-set on a different architecture (they can currently simulate Sparc and MIPS code on Sparc systems).

A survey of trace-driven memory simulation including trace collection, trace reduction and trace processing can be found in [29].

## 2.2 Code Generation and Optimization

Extensive work has been done on fine-grain optimization [23, 14, 1, 8, 24, 17, 13, 19, 25, 15, 18, 2, 30, 10, 5, 4] but very few on frameworks for enabling a fast implementation of such optimizations. Furthermore, we are not aware of a retargetable system that enables the implemention of assembly-code scheduling (with accurate resource usage models), partial-register allocation and instrumentation. Being able to "redo" partial-register allocation is a major capability that is necessary for achieving good performance when using software-pipeline techniques [2].

The main drawback of having SALTO as a separate tool is that it separates the code-generation and the optimization. Ideally, we want these phases to happen simultaneously [8] to be able to generate optimal code. In practice, mixing code-generation and scheduling (especially software-pipeline and global-scheduling techniques) is a very difficult task. Having code generation and scheduling performed at the same time, generally, results in having a non-optimized code-generator (compared to the one provided by the native compiler) and a weak code scheduler that does not implement anything beyond basic-block scheduling. Section 3 covers how we plan to couple the phases. In the remainder of this section we overview works that we believe are close to SALTO.

Code layout optimization is performed by CORD on MIPS based computers. CORD rearranges procedures in an executable to reduce paging and achieve better instruction-cache mapping. It uses a feedback file generated at run-time by an executable instrumented by PIXIE.

OCO [6] is an assembly-language optimizer which was developped at IRISA. It takes an assembly source code as input and produces an optimized version of the code. Optimization is based on local reordering and software pipelining. OCO has been used as a basis for SALTO.

The university of Karlsruhe has developed the BEG system (Back-End Generator) [21]. It produces a code-generator from rule based declarative descriptions. Basic blocks are reordered before and after global register allocation. The system is based on a simulation of the pipeline during instruction reordering. Pipelines that schedule more than one instruction per cycle are also supported. BEG can also produce local register allocators suitable for a great variety of target machines.

MARION [7] has been developed at the University of Washington. It is a code-generator generator. The MIPS R2000, Motorola 88000 and i860 architectures have been described with the specific language Maril. The hardware description includes registers, functional units, multiple issue and pipeline stages. Instructions are given properties that influence reordering. The description is based on reservation tables and on delays required to obtain the result of an instruction. MARION has been mainly used to study the interaction between reordering and register allocation [8]. Compared to SALTO, no user interface is provided to modify assembly-code and we believe that it cannot be used with compilers other than `lcc`.

GCC is a C compiler maintained by the Free Software Foundation, Inc. [28]. It uses the internal representation RTL. The good performance achieved by GCC comes from its ability to apply the right optimization at the right moment within the compilation process. It is improved by a last optimization specific to the target architecture at the end of the process. Instruction reordering within the basic-blocks is performed twice, once before and once after the register allocation stage. The machine description includes the size of delay slots and the annulation conditions, the time needed to compute the result of an instruction and resource conflicts between instructions.

A compiler system named SUIF [20] (which stands for *Stanford University Intermediate Format*) is being developed at the Stanford University. It is built on top of a kernel that preprocesses source code and produces a representation of the program called an *intermediate format*. This format contains low-level description of the structures without losing useful high-level information like arrays, if-then-else structures and loops. Many passes are performed on this representation, implementing a single optimization or transformation at a time. The compiler performs several passes on this representation, but insertion of new passes can easily be made. Features are provided to allow implementation of algorithms ranging from data-dependence analysis or register allocation to symbolic analysis or detection of parallelism.

# 3 Motivation for SALTO

The complexity of today's architectures require that the compiler masters many parameters to produce efficient code. The impact of the modification of some parameters might not be straightforward. We believe that an efficient optimization can be achieved only with an iterative process where different tools exchange information. This section explains how we conceive such a multiple pass code-generation process and why retargetability is an important feature to implement in optimizers.

## 3.1 Multiple Pass Code Generation Process

The effective performance of an application on a superscalar or VLIW processor (embedded or otherwise) depends on many parameters which cannot be easily managed by a single-pass code-generation process. For instance, the effective performance depends on the instruction-cache behavior and on the interaction of basic-blocks. Profiling and tracing tools may provide the user (or a high level code restructurer or a compiler) with information on critical-code sections that should be highly optimized or, for instance, information for an accurate memory layout of basic-blocks, etc.

It is our belief that the code-generation process for performance critical applications must be iterative. It is also our belief, that such an iterative process can be afforded for many embedded applications where time-consuming techniques may be used because of the performance requirement and the long lifetime of the code.

For such a multiple-pass code-generation process, a few tools directly working on the assembly-language are required: a code scheduler, a basic-block profiler, an instruction tracer, an instruction-layout optimizer, etc. The SALTO system we present in this paper is the main component of a unified framework for developing this spectrum of tools for manipulating assembly-languages, which we believe are needed for performance tuning. SALTO is highly retargetable with the assembly-language description as well as with the precise description of the targeted hardware.

## 3.2 The Optimization Loop

In this section, we illustrate the optimization loop of the code-generation of an application. Figure 1 illustrates the information flow between the different components in the optimization loop:

- Control flow and data dependencies are passed from the high-level code restructurer to the low-level code instrumenter. For feeding back information, links to the source code are also propagated during the compilation and code-generation phases.
- The low-level code optimizer feedbacks information to the high-level code restructurer: size, ideal execution time of basic-blocks, register pressure, etc.
- When instrumented, the execution forwards profiling information to the high-level code restructurer.

Figure 2 illustrates the optimization process:

1. The program is instrumented and profiled to determine statements execution time. The optimization will be focused on the most time consuming statements.
2. Information from the low-level code is forwarded to high-level code restructurer.

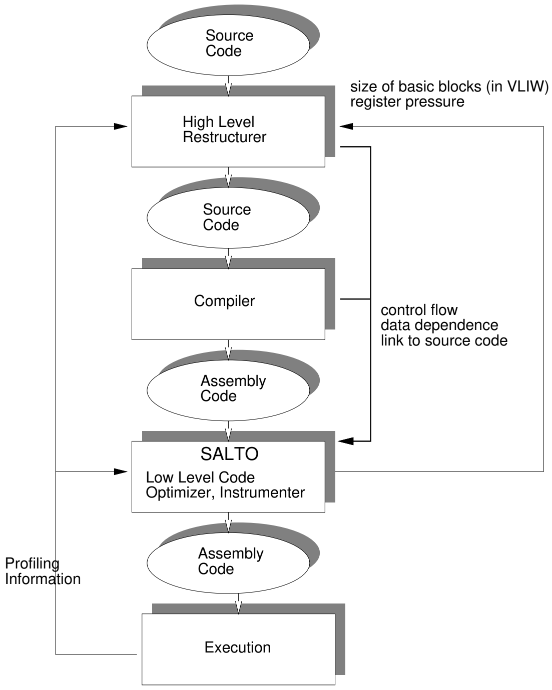



Figure 1: Iterative Compilation Process

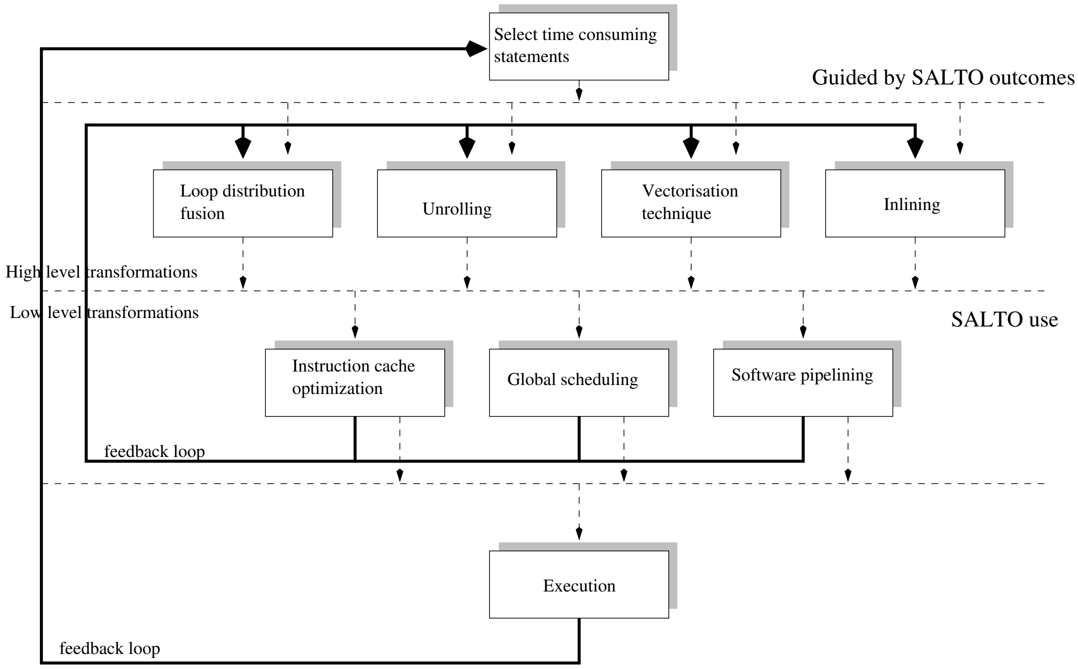



Figure 2: Optimization Process Used by SALTO

3. The high-level code restructurer uses this information as a guide for applying (or not applying) specific transformations. For instance, loop distribution or fusion is guided by information on register pressure as well as information on the execution time of the loop(s) body. Unrolling will be guided by the execution time of the loop body, but also by the size of the code.

   A code size criterion is applied to maintain correct behavior of the instruction-cache.

4. A new low-level code is generated. Then, low-level code optimizations are applied (basic-block scheduling, software-pipeline, etc.) and new information are fed back to the high-level code restructurer. Instruction-cache behavior is estimated. Phases 3 and 4 are reiterated to reach satisfactory performance.

Note that most constructors do not provide the user with the ability to use such an optimization loop.

### 3.3 Motivation for Retargetability

Retargetability is a major issue when building prototypes (for academic uses). It also becomes a major issue in industrial applications, particularly in the domain of embedded systems. Although with the advance of VLSI integration and CAD design reliability, the time to dedicate a new processor to a special-purpose application is shrinking, most object-code schedulers are only optimized for a single hardware configuration. Retargeting such an object-code scheduler generally implies a major rewriting of code.

We believe that tools manipulating assembly-codes (and eventually object code) should be highly retargetable with the precise hardware description of the targeted processor, but also with the instruction-set architecture (ISA).

Being able to retarget tools for manipulating assembler codes with precise hardware description of the targeted processor permits the object-code scheduler to focus on a precise hardware configuration. In an hardware/software codesign approach, it enables tuning the hardware to the correct performance level.

Retargeting tools for manipulating assembler codes with different ISAs is not a straightforward idea. Nevertheless, dedicated hardware solutions derived from commodity core processors may use special instructions. Being able to add such instructions in a "closed" ISA requires major code rewriting.

The costs of retargetability are not negligable, especially in terms of CPU time. However we feel that in many cases users can afford to pay for this retargetability. For instance: for determining the best processor configuration, for choosing a processor and for prototyping new optimization algorithms.

## 4 System Overview

SALTO is composed of three parts: a kernel, a machine description file and an optimization or instrumentation algorithm. Figure 3 illustrates the organization of these three components.

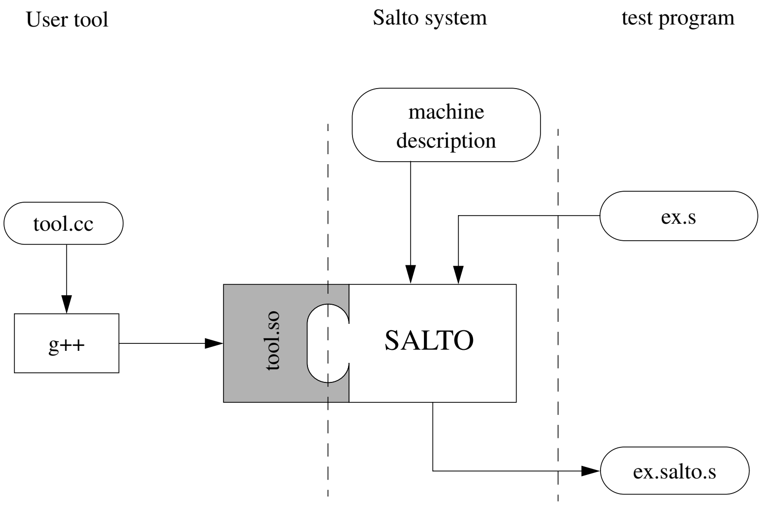



Figure 3: System Overview

1. The kernel performs common required tasks that the user doesn't want to worry about. The parsing of the assembly-code and of the machine-description file are done automatically, as are the construction of the internal representation (see section 5.1). This internal representation is available via the user interface (see section 5.3).

2. The machine description file contains the hardware configuration and the complete description of the instruction-set with reservation-tables. Section 5.2 details the format of this file.

3. The optimization or instrumentation algorithm is supplied by the user. Once the system has read the machine-description file and the assembly-code, an internal representation is built and control is given to the user through the call to a predefined function.

SALTO is based on the former tool OCO [6] developped at IRISA which was written using parts of the code parser from the GNU assembler GAS.

# 5 SALTO Features

In this section we summurize the main features of SALTO. It is built on three components, the data structures for representing the program, the machine description for declaring resources usage and finally the user interface that enables the writing of instrumentation or scheduling algorithms.

## 5.1 Data Structures

The data structures used in SALTO are divided into two groups, depending on their role. The first group represents the control flow of the program, the second group describes resource usage and data dependencies between instructions.

### 5.1.1 Control Flow Structures

A program written in assembly-language can be viewed as consisting of several procedures, each of which is a list of instructions. Within a procedure, instructions are grouped into basic-blocks.

While parsing the code, SALTO builds the list of the procedures it encounters. Each procedure has a list of basic-blocks, and each block "knows" its list of instructions. Figure 4 illustrates this organization. The procedures, basic-blocks and instructions are internally managed as cyclic lists, but the internal representation is hidden by the user interface, as shown in section 5.3. Markers are added by the analyzer to give useful information about the piece of code being processed: basic-blocks frontiers (shaded instructions in figure 4), or procedures frontiers, name of current segment, etc.

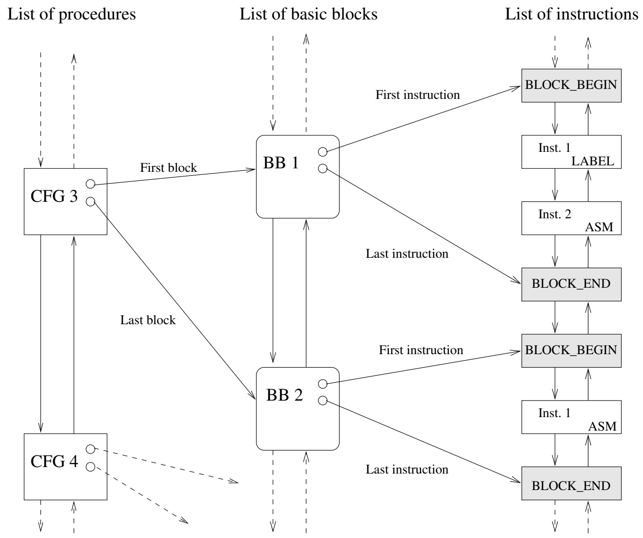
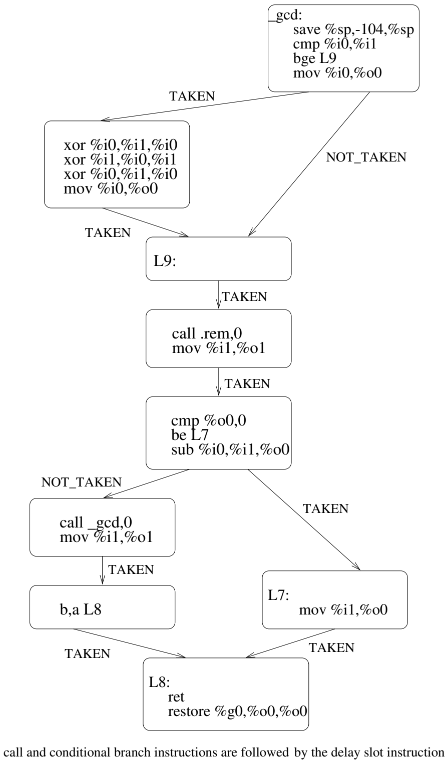



Figure 4: Organization of Control-Flow Structures

Additionally, the control flow graph (CFG) is built for each procedure. The vertices are the basic-blocks and the edges denote the execution order of the basic-blocks. Edges are labeled to indicate if they correspond to the taken or not-taken branch. See figure 5 for an example of the graph corresponding to a simple procedure written in C language. A known limitation is due to the static analysis of the assembly-code. If the target address of a branch instruction is a computed register value, SALTO leaves the target undetermined. However, the basic-blocks are constructed assuming computed branches can only target labels[1].




```
/*
 * Computes the GCD of two integers
 * using Euclide's method
 */
int gcd(a,b)
      int a, b;
{
   int result;

   if (a<b) a∧ =b, b∧ =a, a∧ =b;
   if (a%b) result = gcd(a-b,b);
   else result=b;
   return result;
}
```

Figure 5: Control Flow Graph of a Simple Program

### 5.1.2 Hardware Resources and Data Dependencies

The second part of the data structures provided by SALTO gives information about the resources needed by an instruction to complete execution. A resource is usually a register, a functional unit or the memory, but it could be any piece of hardware needed to describe the behavior of the machine (see section 5.2 for an explanation on how to define resources). Each instruction needs a resource during a number of cycles with a particular access mode: three modes are available for a resource: *read*, *write* or *use*.

Each instruction is described by a reservation-table, which indicates the list of resources it needs and the mode and cycle a resource is accessed. This information is used when determining the type of dependence between two instructions: RAW (*read after write*), WAW (*write after write*), WAR (*write after read*).

The memory is currently seen as a unique resource and all memory accesses are considered to be to the same memory location. SALTO is conservative when checking data dependencies and thus two memory accesses, one of which is a write, always leads to a dependence. However, the functions of the user interface make it possible for the user to write his own alias analysis algorithm to detect such situations and to implement a link with the compiler data dependence analysis.

## 5.2 Machine Description

SALTO is designed to be a retargetable tool. Thus, the target machine must be described in a flexible way, permitting an accurate description while retaining the ability to easily modify parameters. This is achieved

[1]This may lead to incorrect basic block computation in some circumstances.

with a Lisp-like language based on the reservation-tables formalism. The description file is parsed by SALTO and an internal representation is built using RTL (Register Transfer Language) [28]. The machine description contains:

1. the syntax of the assembly-language used, that is, how does a comment start, how many registers are there, and what are their names;
2. all the resources needed for the computation of data dependencies;
3. the list of the instructions recognized by the assembler with the applicable formats and the associated reservation-tables;
4. semantical information to warn SALTO about special features implemented in the processor like *bypass-mechanism* or *delayed branch*.

The following paragraphs describe more precisely the different steps involved in writing a machine-description file.

### 5.2.1 Definition of the Available Resources

The language describes the available resources as precisely as desired. A high-level description including only an integer unit, a memory access unit and a floating point unit may be enough to perform local reordering techniques. A lower-level description, with bus access and cache memories is also possible. Three types of resources are currently recognized: registers, memory and functional units. SALTO also supports symmetric superscalar architectures: any functional unit can be replicated any number of times. Figure 6 shows an example of such a description for the Sparc architecture [11].

```
; Floating point registers
(def_ress (base_name "%f" 0 31) [(type "reg") (width 32)])

; Integer unit
(def_ress (name "alu") [(type "functional_unit")])

; Memory access
(def_ress (name "mem") [(type "memory")])

; Class of the out-registers (%o0 - %o7)
(def_class "OUT_reg" "reg" [ (ress (base_name "%o" 0 7)  []) ])
```

Figure 6: Definition of Hardware Resources of the Sparc

### 5.2.2 Instruction Set

The definition of an instruction-set includes a declaration of the instruction names, all the applicable formats, and the associated reservation-tables. These reservation-tables are used when resolving data dependencies. They store information about which functional units are needed by an instruction, for how many cycles and when the result is written. Figure 7 illustrates the reservation-table of the `ld [%o1],%o2` instruction of the Sparc [11]. Although the reservation-table formalism we choose to use to describe instructions is almost always sufficient, it has some limitations which prevent it from describing special features of particular processors. The *push pipeline* instruction of the Intel i860 is an example of such a restrictions. The tool can still be used [30], however in this case the particular behavior is handled at the scheduling algorithm level.

Figures 8 and 9 are extracted from our machine description file for the Sparc architecture. The tables are first defined as macros (fig. 8) and then used (fig. 9).

### 5.2.3 Semantical Information

Some instructions of an instruction-set may have a particular behavior. "Semantic" properties can be added to these instructions. In the case of the Sparc processor, this concerns *delayed branch*. The corresponding declaration can be seen in figure 10.

| Resources | | Cycles | | | |
|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 |
| FU | issue | use | | | |
| register | %o1 | read | | | |
| FU | alu | | use | | |
| Memory | mem | | | read | |
| register | %o2 | | | | write |

Figure 7: Reservation-Table of the `ld` instruction of a Model of the Sparc

```
; Every instruction uses ISSUE at cycle 1 : superscalar degree of the
; architecture is given by the number of ISSUE units.
#define ISSUE ( ress (name "issue") [(use) (at_cycle 1)] )

; Reservation table for integer instructions,   reg+imm -> reg
#define R_ALU_1 (reser_table [\
  ISSUE \
        (ress (match_arg 0) [(read)  (at_cycle 1) INT_REG])\
        (ress (name "alu")  [(use)   (at_cycle 2)])\
        (ress (match_arg 2) [(write) (at_cycle 2) INT_REG])\
        ])

; Reservation table for memory access instructions,  [reg] -> reg
#define R_LD_1(RTYPE) (reser_table [\
  ISSUE \
        (ress (match_arg 0) [(read)  (at_cycle 1) INT_REG])\
        (ress (name "alu")  [(use)   (at_cycle 2) ])\
        (ress (name "mem")  [(read)  (at_cycle 3)])\
        (ress (match_arg 1) [(write) (from_cycle 2) (to_cycle LOAD_DELAY )\
                RTYPE LOAD_INTERLOCK])\
        ])

; Reservation table for branch instructions
#define R_BR (reser_table [\
  ISSUE \
        (ress (name "icc")  [(read)  (at_cycle 1)])\
        (ress (name "alu")  [(use)   (at_cycle 2)])\
        ])
```

Figure 8: Description of Reservation-Tables

```
(def_asm  "ld"  [ (input  "[1],g")  R_LD_1(FP_REG) ])
(def_asm  "add" [ (input  "1,i,d")  R_ALU_1  ])
(def_asm  "bge" [ (input  "l")   (sem [S_BRANCH(0) DELAY_SLOT]) R_BR])
```

Figure 9: Definition of the Processor Instruction-Set

```
;-- Special case for the delay slot --
#define DELAY_SLOT      (delay_slot 1)

;-- Instructions in the delay slot must not be reordered
#define NOREORDER       (noreorder)
```

Figure 10: Semantics of *Special* Instructions

### 5.3 User-Interface

The object-oriented user interface provides a flexible way to deal with the internal data structures. Features of SALTO include:

- access to the code at three different levels : procedure, basic-block or instruction;
- modification of the code : insertion, deletion;
- unparsing;
- access to the reservation-tables;
- computation of dependencies and delays between instructions caused by pipeline stalls;
- addition of attributes : attributes are a flexible way to put any kind of information on a particular basic-block or instruction.

The object classes and types provided by the interface are the following:

**CFG** This object represents a function. It is the control flow graph. The vertices of this graph are objects of class `BB` (see below) and the edges are labeled depending on the type the branch instruction (`TAKEN` or `NOT_TAKEN`).

**BB** This object represents basic-blocks.

**INST** corresponds to an instruction of the assembly-code being processed. It is not necessarily a mnemonic of the processor's instruction set, but it can be a macro instruction, a pseudo instruction, a label or some information added by SALTO for further analysis (by example: basic-block begin , basic-block end, loop begin, etc.). An instruction can compute the type of dependence it has with another one.

**RES** This object represents one of the user-defined hardware resources present in the simulated architecture. It may be a functional unit, a register, a bus, etc.

**CRES** Classes of resources provide a means for grouping some resources and to refer to them with a single name. They are defined by the user in the machine-description file. For example: registers are often grouped together within a *class of registers*.

**TRES** This object implements a reservation-table. It is used for code-scheduling algorithms.

**SaltoAttribute** Attributes can be seen as hooks on instructions (`INST`) and basic-blocks (`BB`). They permit any kind of information to be attached to any particular block or instruction. A number of attributes are predefined for internal use. SALTO attributes are similar to the attributes in the Sage++ system [16]. Some attributes have a predefined value:

**COMMENT_ATT:** the string attached to the instruction or block is considered as a comment and is printed before the block or at the end of the line while unparsing;

**UNPARSE_ATT:** the string is printed instead of the real assembly-code while unparsing;

**NOCYCLE_ATT:** an integer is attached to the instruction or block, it is the cycle when the instruction must be scheduled within the basic block. It is used by reorder algorithms.

A representative subset of the general and member functions is shown is appendix A. Figure 11 illustrates the use of the user interface with a simple example. More complex examples are given in section 6.

# 6 Two Examples of Tools Written with SALTO

To illustrate SALTO usage, this section presents in greater detail two programs: basic-block count instrumentation and local scheduler. These examples give an idea of the range of tools that can be implemented using SALTO, but are not representative of state-of-the-art profiling or scheduling techniques.

```
// renames register reg_s into reg_d for instruction inst
// reg_s and reg_d can be obtained for example using
// RES *getResbyName(char *)
void rename(INST *inst, RES *reg_s, RES *reg_d) {
  int ninput, i;
  RES *res;

  ninput = inst → numberOfInput();
  for(i=1; i ≤ ninput; i++) {
    res = inst → getInput(i);
    if (res == reg_s) {
      inst → setInput(i, reg_d);
  }
    ... // same for output
}
```

Figure 11: Use of the User-Interface

### 6.1 Instrumentation

Appendix B.1 shows a small example of assembly-code instrumentation. The goal of this piece of code is to instrument basic-blocks to obtain a basic-block count. SALTO adds a comment before each basic-block using the `COMMENT_ATT` attribute and adds instrumentation code at the beginning of each block. This code saves some register values before calling the function `bbcount` which performs the counting. This function may be written in a high-level language. In our example, it is written in the C language. The function is linked with the generated assembly-code to produce an executable.

### 6.2 Local Reordering

As an example of local reordering we have implemented a list-scheduling algorithm using SALTO. Our program is shown in appendix B.2. The main function is `reorder`. It builds the dependence matrix for the instructions of the current block: `dep[i][j]` equals to 1 if instruction number i depends on instruction number j. The main loop computes the scheduling cycle for each instruction until a branch is seen: `verify_predecessors` checks if all instructions that have a data dependence have already been scheduled. `earliest_cycle` then computes the delays before all previous results are obtained. `IsConflict` is used to detect resource conflicts. The branch instructions, if they exist, and the delay slot are processed afterwards. The blocks are effectively reordered by `orderAccordingToCycles` and nops are added if necessary by `addNecessaryNops`.

## 7 Conclusion

Performance tuning of critical applications is a major issue that cannot be by a one pass compiler. For many applications, e.g. embedded applications, one can afford to pay long performance tuning costs accross a multiple-pass code generation.

SALTO is a framework for developing a large range of tools dealing with performance tuning and optimizations of low-level codes. The major advantages of SALTO compared with already existing tools is its ability to generate performance-analysis tools as well as optimization tools and in its retargetability towards any instruction-set.

The main goals of SALTO are to be part of a global solution for manipulating assembly-code to implement low-level code restructuration as well as to provide a high-level code restructurer with useful information collected from the assembler code and from instruction profiling. To achieve this, we implemented several features that ease integration within a compilation process:

- a high-level user interface which allows powerful transformations of the assembly-code in a simple manner,
- the ability to analyze, profile and optimize a program using the same tool and hence using the same user interface,

- the possibility to feed gathered information to an analyzer with a simple procedure call, which is more effective than using temporary files and overcomes the problem of large traces (which typically measure in gigabytes),
- to give the user full control over the modifications he adds to the original program and avoid problems like register reallocation created by Pixie, and
- to provide a highly-retargetable tool by means of accurate description of the hardware and of the assembly-language used, giving full access to the resources needed by an instruction.

The two examples presented in the paper and a variety of others have shown the usage of our system. Salto yet robust enough to be used on real applications. We expect it to be available soon by FTP at address `ftp.irisa.fr`.

We strongly believe that a Salto-like framework is a major software piece that is required in hardware/software codesign of dedicated versions of processors where not only time-to-market, but also software development cost is critical.

# References

[1] A. Aiken and A. Nicolau. Perfect pipelining : a new loop parallelization technique. In *Lecture Note In Computer Science*, pages 221–235, 1988.

[2] Vicki H Allan, Reese B Jones, Randall M Lee, and Stephen J Allan. Software pipelining. *ACM Computing Surveys*, 27:367–432, September 1995.

[3] Thomas Ball and James R Larus. Optimally profiling and tracing programs. In *ACM Transactions on Programming Languages and Systems*, volume 16, pages 1319–1360, July 1994.

[4] F. Bodin, F. Charot, and C. Wagner. Overview of a high-performance programmable pipeline architecture. In *ACM Supercomputing*, 1988.

[5] François Bodin. *Optimisation de microcode pour une architecture horizontale et synchrone, Étude et mise en œuvre d'un compilateur.* PhD thesis, Université de Rennes 1, June 1989.

[6] François Bodin, Gwendal Le Fol, and Frédérique Raimbault. Oco manuel de l'utilisateur (version préliminaire). Technical Report 930, Irisa, May 1995.

[7] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. The marion system for retargetable instruction scheduling. In *Programming Languages and Systems*, 1991.

[8] David G. Bradlee, Robert R. Henry, and Susan J. Eggers. Integrating register allocation and instruction scheduling for riscs. In $4^{th}$ *International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 122–131, April 1991.

[9] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1994.

[10] Lucile Cognard. *Modélisation du comportement dynamique des processeurs pour la génération automatique de réordonnanceurs de code.* PhD thesis, Université d'Orléans, 1995.

[11] LSI Logic Corporation. *SPARC Architecture Manual (Version 7)*, 1990.

[12] DEC. *ATOM User Manual*, March 1994.

[13] K. Ebcioglu and T. Nakatani. A new compilation technique for parallelizing loops with unpredictable branches on a vliw architecture. In *Languages and Compilers for Parallel Comuting*, pages 213–229, 1989.

[14] Christine Eisenbeis, William Jalby, and Alain Lichnewski. Compiler techniques for optimizing memory and register usage on the cray 2. In *International Journal on High Speed Computing*, June 1990.

[15] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, 1985.

[16] Bodin F., Beckman P., Gannon D., and Srinivas J.G.S. Sage++: A class library for building fortran and c++ restructuring tools. In *Object-Oriented Numerics Conference*, April 1994.

[17] J.A. Fisher. Trace scheduling: a technique for global microcode compaction. In *IEEE Transactions on Computers*, pages 478–490, July 1981.

[18] Franco Gasperoni. *Scheduling for horizontal systems : the VLIW paradigm in perspective.* PhD thesis, New-York University, 1991.

[19] R. Govindarajan, Erik R. Altman, and Guang R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In $27^{th}$ *Annual International Symposium on Microarchitecture*, pages 85–94, 1994.

[20] Stanford Compiler Group. Suif compiler system. Web pages available from `http://suif.stanford.edu`.

[21] H.Emmelmann, F-W.Schroeer, and R.Landwehr. Beg - a generator for efficient back ends. In *SIGPLAN Conference on Programming Language Design and Implementation*, volume 24, July 1989.

[22] Gordon Irlam. Spa package. Electronically available at `ftp://chook.cs.adelaide.edu.au/pub/sparc/spa-1.0.tar.gz`, 1991.

[23] Monica Lam. Software pipelining: an effective scheduling technique for vliw machines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 318–328. ACM SIGPLAN, June 1988.

[24] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In $27^{th}$ *ternational Symposium on Microarchitecture*, pages 63–74, December 1994.

[25] J. Ruttenberg, G. Gao, A. Stoutchinin, and W. Lichtenstein. Software pipelining showdown: Optimal vs. heuristic methods in a production compiler. In *Sigplan Conference on Programming Language Design and Implementation*, May 1996.

[26] Michael D. Smith. Tracing with pixie. Technical report, Harvard university, 1991.

[27] Amitabh Srivastava and Alan Eustace. Atom: a system for building customized program analysis tools. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1994.

[28] R. M. Stallman. *Using and porting GNU CC.* Free Software Foundation, Jun 1993.

[29] Richard Uhlig and Trevor Mudge. Trace-driven memory simulation : A survey. Technical report, University of Michigan, 1995.

[30] Daniel Windheiser. *Optimisation de la localité de données et du parallélisme à grain fin.* PhD thesis, Université de Rennes 1, May 1992.

# A Member Functions Provided by the User Interface

The member functions described in this appendix are a subset of the complete user interface. It is intended to show the extent of capabilities provided by SALTO.

## A.1 General Functions

| | |
|---|---|
| `char *getTargetFlag()` | returns the name of the target architecture (for example “Sparc”). |
| `int numberOfCFG()` | gives the number of procedures. |
| `CFG *getCFG(int i)` | returns a pointer on the control flow graph of the $i^{th}$ procedure. |
| `void removeCFG(int i)` | removes a procedure. |
| `void produceCode(FILE *fg)` | writes in the file *fg* the complete code of the program |
| `RES *getResbyName(char *n)` | returns the RES object corresponding to the given name |

## A.2 Class CFG

These functions are member functions of the class CFG.

| | |
|---|---|
| `int numberOfBB()` | number of basic-blocks of this procedure. |
| `BB *getBB(int i)` | returns a pointer on the block number $i$. |
| `void deleteBB(int i)` | deletes a basic-block and its associated instructions. |
| `BB *createNewBB()` | create a new basic-block. |
| `BB *extractBB(int i)` | the $i^{th}$ block is extracted from the list of blocks of this procedure. |
| `void insertBB(int pos, BB *b)` | insert a block at the $pos^{th}$ position within the procedure. |
| `void linkBB(BB *source, BB *sink, enum cft_type t)` | adds a link between the basic-blocks *source* and *sink*. The parameter $t$ specifies if the branch is taken (`TAKEN`) or not (`NOT_TAKEN`). |
| `void produceCode(FILE *fg)` | writes in the file *fg* the assembly code of this procedure. |

## A.3 Class TRES

These functions are member functions of the class TRES.

| | |
|---|---|
| `void reset()` | destroys all the reservations of the table. |
| `markRes(INST *inst, int cycle)` | adds entries in the reservation-table corresponding to the instruction resources usage. |
| `int IsConflict(int from, int to, enum access_mode mode, RES *res)` | checks if the given reservation would lead to a resource conflict. |

## A.4 Class INST

These functions are member functions of the class INST.

| | |
|---|---|
| `INST *copy()` | duplicates the instruction and returns a pointer to the copy. Attributes attached to the instruction are not copied. |
| `int [get/set]Cycle()` | [reads/sets] the scheduling cycle of this instruction. |
| `char *getName()` | gets the name of the instruction, i.e. the mnemonic without parameters. |
| `int numberOf[Input/Output/Use]()` | gets the number of resources [read/written/used] by this instruction while executing. |
| `RES *get[Input/Output/Use](int i)` | returns a pointer to the $i^{th}$ [input/output/use] resource. |
| `int isBranch()` | checks if this instruction is a conditional branch instruction. |
| `int isCTI()` | checks if this instruction has an effect on the control flow. |
| `int isNop()` | checks if it is a *no-operation* instruction. |
| `enum dependence IsDep(INST *ii)` | computes the data dependence between the current instruction and the parameter (NONE, RAW, WAW, WAR). |
| `int getDelay(INST *ii)` | computes the lowest number of cycles needed to resolve the data dependencies. |
| `void addAttribute(int type, void *a, int size)` | adds an attribute of type *type* to this instruction whose data field is pointed to by $a$. |

## A.5 Class BB

These functions are member functions of the class BB.

| | |
|---|---|
| `int numberOfAsm()` | returns the number of assembly mnemonics in the block. |
| `INST *getAsm(int i)` | returns a pointer to the $i^{th}$ instruction in this basic-block. |
| `void removeAsm(int i)` | deletes the $i^{th}$ instruction of this basic-block. |
| `void insertAsm(int pos, INST *st)` | inserts a new instruction at position *pos* within the block. |
| `void orderAccordingToCycles()` | swaps instructions within the block according to the scheduling cycles specified with `setCycle()`. |
| `void addNecessaryNops()` | add *nop-operation* instructions in the block to fill the cycles for which no instruction is to be scheduled. |
| `int numberOf[Suc/Pred]()` | returns the number of [successors/predecessors] of a block in the control flow graph. |
| `BB *get[Suc/Pred](int i)` | returns the $i^{th}$ [successor/predecessor] of this block. |
| `int isMemberOf(INST *st)` | checks if the given instruction is part of this basic-block. |
| `void addAttribute(int type, void *a, int size)` | adds an attribute to this block whose data field is pointed to by $a$. |

# B Examples

## B.1 Basic-Block Instrumentation

Here is the code of a SALTO tool which instruments the basic-blocks of a program. The function called for each block is listed below.

```
#include "salto.h"

void add_code(BB *bb, int cpt) {
  INST *nop;
  char instcode[STR_MAX];

  nop = bb → newNOP();
  sprintf(ch, "\t save register values"
           "\t mov %d,%%o0"
           "\t call _bbcount \n"
           "\t restore values", cpt);
  nop →addAttribute(UNPARSE_ATT, strdup(ch), strlen(ch));
  bb → insertAsm(1, nop);
}

void Salto_hook() {
  CFG *proc;
  BB *bb;
  int i, j, ncfg, nbb, cpt=0;
  char ch[20];

  ncfg = numberOfCFG();
  for(i=1; i ≤ ncfg; i++) { // for each procedure
    proc = getCFG(i);
    nbb = proc → numberOfBB();
    for(j=1; j ≤ nbb; j++) { // for each basic block
      bb = proc → getBB(j);
      sprintf(ch, "bb %d", ++cpt);
      bb → addAttribute(COMMENT_ATT, strdup(ch), strlen(ch)); // adds a comment at the beginning of the BB
      add_code(bb, cpt);
    }
  }
  produceCode(stdout); // Finished, now dumps instrumented code on stdout
}
```

Function `bbcount`. It must be linked with the produced code.

```
#include <stdio.h>

void bbcount(int n)
{
  printf("block %d reached\n", n);
}
```

## B.2 Optimization with List Scheduling Algorithm

Implementation of a list scheduling algorithm using SALTO.

```
#include "salto.h"

int verify_predecessors(int verif, int **dep, INST **inst)
{
  for(int i=0; i<verif; i++)
    if ( (dep[verif][i]) && (inst[i]→getCycle() < 0) ) return 0;
  return 1;
}
```

```
int earliest_cycle(int s, int **dep, INST **inst)
{
  int i, z, max = 0;
  for(i=0; i<s; i++) {
    if (dep[s][i]) {
      z = inst[i] → getCycle() + inst[s] → getDelay(inst[i]);
      if (z>max) max = z;
    }
  }
  return max;
}

void build_dep_matrix(int **dep, INST **inst, int n)
{
  for(int i=0; i<n; i++) {
    dep[i][i]=0;
    for(int j=0; j<i; j++) {
      if (inst[i]→IsDep(inst[j])) dep[i][j]=1;
      else dep[i][j]=0;
    }
  }
}

INST **instr;
int **dep;

void reorder(BB *bb)
{
  int i, to_be_scheduled, cycle_min, nasm, offset, branch_seen,brindex,last_cycle;
  TRES *res_table = new TRES; // need a reservation table

  nasm = bb → numberOfAsm();
  instr = new (INST *)[nasm]; // to avoid calling getAsm each time we need it
  for(i=0; i<nasm; i++) instr[i] = bb → getAsm(i+1);

  build_dep_matrix(dep, instr, nasm); // build the dependence matrix
  branch_seen = last_cycle = 0;

  to_be_scheduled = nasm; // number of instructions to be scheduled
  while (to_be_scheduled && !branch_seen) { // before a branch instruction
    for(i=0; i < nasm; i++) {
      if (instr[i] → isCTI()) {
        brindex = i;
        branch_seen = 1;
        break;
      }
      if ( (instr[i]→getCycle()) < 0 ) { // not already scheduled ?
        if (verify_predecessors(i, dep, instr)) { // Do all the predecessors have been scheduled ?
          cycle_min = earliest_cycle(i, dep, instr); // wait for data dependencies to be resolved
          offset = 0; // now wait for resources to be available...
          while (res_table→IsConflict(instr[i],cycle_min+offset)) offset++;
          res_table → markRes(instr[i], cycle_min + offset); // mark resources occupancy into reservation table
          if (cycle_min + offset > last_cycle) last_cycle = cycle_min+offset;
          instr[i] → setCycle(cycle_min + offset); // specify the cycle
          to_be_scheduled--;
          break;
        }
      }
    }
  }
  if (branch_seen) { // the branch instruction
    instr[brindex] → setCycle(last_cycle + 1);
```

```
    to_be_scheduled--;
  }
  if (to_be_scheduled) instr[nasm-1] → setCycle(last_cycle + 2); //delay slot instruction
  bb → orderAccordingToCycles(); // reorder block according to values specified by setCycle()
  bb → addNecessaryNops();
  delete res_table;
  delete instr;
}
```