SALTO: System for Assembly-Language Transformation and Optimization

Erven Rohou, François Bodin, André Seznec
{erohou,bodin,seznec}@irisa.fr

Abstract

On critical applications 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). 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. High-level functions are provided 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 assembly code and from instruction profiling.

1 Introduction

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, assembly-code profiling and instrumentation, along with an accurate description of the target architecture and high retargetability.

SALTO is a first step towards the availability of a retargetable framework for developing the whole spectrum of tools that manipulate assembly-language. Building these tools typically implied a time consuming development phase (rewriting from scratch large pieces of code was often needed) and required much investment. The objective of such a system is to provide the user with a single environment that will automatically perform boring and repetitive tasks and allow him to implement the tools needed for performance tuning on low-level codes.

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 (CFG) of the program and a model of the target architecture. They are easily accessible through the user interface and provide a convenient way to implement algorithms without having to worry about infrastructure.

Section 2 reviews related works and points out how our tool provides a more general and integrated solution for building tracers, profilers and optimizers. An overview of the features of the system is given in section 3. Section 4 shows an example and section 5 concludes this paper.

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 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 (particularly interesting for determining critical sections on which to focus) whereas tracing also furnishes the order of instructions and addresses.

2.1 Analysis

Many techniques for analyzing program behavior depend on instrumentation of either the source code or the executable file. Pixie [20] 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 improve the technique by minimizing the number of counters required. They build a CFG 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 [9, 21] was proposed for the DEC Alpha. ATOM also depends on code instrumentation and provides common analysis and performance tools. A partial list given in [9] 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.

Larus and Schnarr [18] have built a library called EEL (Executable Editing Library). EEL enables the building of tools to analyze and edit compiled programs without having to worry about the underlying machine and operating system. EEL analyzes an executable and builds some abstractions like routine, CFG, instruction and *snippets*. A snippet is a piece of code to be included in the original executable for instrumentation purpose (when saving the registers state, calling a foreign function and restoring the previous values). This code is necessarily specific to each particular machine and cannot be avoided when adding code to an executable. It is similar to the architecture dependent code we add to an assembly source code.

Gordon Irlam's Spy [16] runs on Sparc architectures under SunOs 4.x. Spy exploits special features of the Sparc microprocessor [8] to fetch the instruction to be executed. Spy provides as output a trace composed of the instruction and data addresses. Other tools achieve analysis via instruction-set architecture (ISA) emulation.

Cmelik and Keppel chose another strategy with SHADE [7]: 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 ISA 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 [23].

2.2 Code Generation and Optimization

Extensive work has been done on fine-grain optimization [1, 2, 4, 6, 10, 11, 14, 17] 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 [6] 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 and a weak code scheduler that does not implement anything beyond basic-block scheduling. 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.

The university of Karlsruhe has developed the BEG system (Back-End Generator) [12]. 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 [5] is a code-generator generator been developed at the University of Washington. 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 [6]. 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. [22]. 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

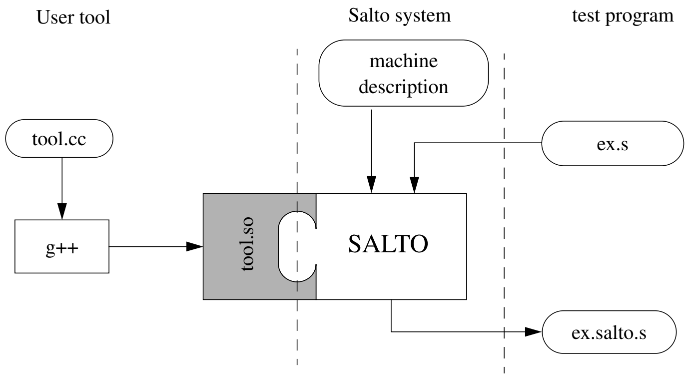


Figure 1: System Overview

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 [15] (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 System Overview

Salto is based on the former tool Oco developped at IRISA which was written using parts of the code parser from the GNU assembler Gas. A more complete description of Salto can be found in [19]. Salto is composed of three parts: a kernel, a machine description file and an optimization or instrumentation algorithm. Figure 1 illustrates the organization of these three components.

1. The kernel performs common required tasks that the user doesn't want to worry about like parsing the assembly-code and the machine-description file or the construction of the internal representation (see section 3.1). This internal representation is available via the user interface (see section 3.3).

2. The machine description file contains the hardware configuration and the complete description of the ISA with reservation-tables. Section 3.2 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.

In this section we summarize 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.

3.1 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.

3.1.1 Control Flow Structures

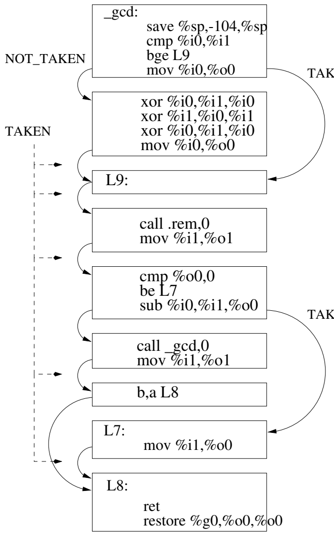
While parsing the code, SALTO builds the list of the procedures it encounters. Each procedure has a list of basic-blocks, and each block "knows" its list of instructions. Additionally, the CFG is built for each procedure. The vertices are the basic-blocks and the edges denote the execution order of the basic-blocks. Edges are labeled to indicate if they correspond to the taken or not-taken branch. See figure 2 for an example of the CFG corresponding to a simple procedure written in C language and compiled on a Sparc workstation. 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 assumes every block of the current procedure can potentially be the target. However, the basic-blocks are constructed assuming computed branches can only target labels (this may lead to incorrect basic block computation in some circumstances).

3.1.2 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 3.2 for an explanation on how to define resources). Each instruction needs a resource during a number of cycles with a particular access mode: *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.




```
/*
 * 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 2: Control Flow Graph of a Simple Program

3.2 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 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) [22]. The machine description contains:

1. the syntax of the assembly-language used;
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.

3.2.1 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

```
; 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")])
```

Figure 3: Definition of Hardware Resources of the Sparc

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 3 shows an example of such a description for the Sparc architecture [8].

3.2.2 Instruction Set

The definition of an ISA 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 and resource conflicts. They store information about which functional units are needed by an instruction, the first and last cycle it is required and when the result is written.

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 restriction. The tool can still be used, however in this case the particular behavior is handled at the scheduling algorithm level.

Figure 4 is extracted from our machine description file for the Sparc architecture. The tables are first defined as macros and then used.

```
; 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,  register+immediat->register
#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]) ])

(def_asm  "add" [ (input  "1,i,d")  R_ALU_1 ] )
```

Figure 4: Description of Reservation-Tables

3.2.3 Semantical Information

Some instructions of an ISA might have a particular behavior. "Semantic" properties can be added to these instructions. In the case of the Sparc processor, this concerns *delayed branch*: the instruction immediately following a branch is speculatively executed, and thus is part of the same basic-block.

3.3 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; with possibility of modification: insertion, deletion;
- unparsing;
- access to the reservation-tables and 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 most important classes defined by the interface are: `CFG`, `BB`, `INST` which represent respectively a procedure, a basic-block and an instruction, `RES` which represents a user-defined hardware resource, and `TRES` which implements a reservation table. The additional class `SaltoAttribute` permit any kind of information to be attached to any particular block or instruction. SALTO attributes are similar to the attributes in the Sage++ system [13].

Figure 5 illustrates the use of the user interface with a simple example. A more complex example is given in section 4.

```
// Renames register reg_s into reg_d for instruction inst.
// reg_s and reg_d are obtained for example using RES *getResbyName(char *)
void rename(INST *inst, RES *reg_s, RES *reg_d) {
  int ninput, noutput, i;
  ninput = inst → numberOfInput();
  for(i=1; i ≤ ninput; i++)
    if (inst → getInput(i) == reg_s)
      inst → setInput(i, reg_d); ;
    ... // same for output
}
```

Figure 5: Use of the User-Interface

4 An Example: Local Reordering with SALTO

To illustrate SALTO usage, this section presents an application in greater detail: a local scheduler. This example gives an idea of the range of tools that can be implemented using SALTO, but is not intended to be representative of state-of-the-art scheduling techniques. As an example of local reordering we have implemented a list-scheduling algorithm using SALTO. Our program is shown in appendix A. The main function is `reorder`. It builds the dependence matrix `dep[i][j]` for the instructions of the current block. 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 needed results are obtained. `IsConflict` is used to detect resource conflicts. The branch instruction, if it exists, and the delay slot are processed afterwards. The blocks are effectively reordered by `orderAccordingToCycles` and nops are added if necessary by `addNecessaryNops` to fill the empty slots.

5 Conclusion

Performance tuning of critical applications is a major issue that cannot be done by a one-pass compiler. For many applications, e.g. embedded applications, one can afford to pay long performance tuning costs across 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 its retargetability towards any ISA.

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 assembly 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 (ability to analyze, profile and optimize a program) of the assembly-code in a simple manner using a single tool;
- the possibility to feed gathered information to an analyzer with a simple procedure call, which is far 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;
- 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 is 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] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. The Marion system for retargetable instruction scheduling. In *Programming Languages and Systems*, 1991.

[6] 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.

[7] 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.

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

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

[10] 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.

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

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

[13] 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.

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

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

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

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

[18] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Conference on Programming Language Design and Implementation*, June 1995.

[19] Erven Rohou, François Bodin, André Seznec, Gwendal Le Fol, François Charot, and Frédéric Raimbault. SALTO: System for assembly-language transormation and optimization. Technical Report 1032, IRISA, June 1996.

[20] Michael D. Smith. Tracing with pixie. Technical report, Stanford university, 1991.

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

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

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

A Example: Optimization with List Scheduling Algorithm

```
#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++)
      dep[i][j] = ( inst[i]->IsDep(inst[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;
      }
      // Is this instruction already scheduled ?
      if ( (instr[i]→getCycle()) < 0 ) {
        // Are all the predecessors scheduled ?
        if (verify_predecessors(i, dep, instr)) {
          // Wait for data dependencies to be resolved
          cycle_min = earliest_cycle(i, dep, instr);
          // Now wait for resources to be available...
          offset = 0;
          while (res_table→IsConflict(instr[i],cycle_min+offset)) offset++;
          // Mark resources occupancy into reservation table
          res_table → markRes(instr[i], cycle_min + offset);
          if (cycle_min + offset > last_cycle) last_cycle = cycle_min+offset;
          // Specify the cycle
          instr[i] → setCycle(cycle_min + offset);
          to_be_scheduled--;
          break;
        }
      }
    }
  }
  // The case of the branch instruction
  if (branch_seen) {
    instr[brindex] → setCycle(last_cycle + 1);
    to_be_scheduled--;
  }
  // The delay slot instruction, if any
  if (to_be_scheduled) instr[nasm-1] → setCycle(last_cycle + 2);

  // Reorder according to values specified by setCycle()
  bb → orderAccordingToCycles();
  bb → addNecessaryNops();
  delete res_table;
  delete instr;
}
```