A Code Generator Generator for Superscalar Architectures

Mariza A. S. Bigonha[1]
José Lucas M. Rangel Netto[2]
Roberto S. Bigonha [3]

Abstract

Modern computer architectures have motivated research for more efficient compiler techniques. These new architectures, however, delegate the solution of the most complicated problems in code generation to the compilers. This paper shows the design of a code generator system for superscalar architectures based on formal machine description. We also discuss about several problems related to code generation to these processors.

1 Introduction

The focus of the code generator system described in this paper is the superscalar processors. These processors are an evolution of the `RISC` (*Reduced Instruction Set*) architectures. Superscalar architectures include several common features. The most importants are: (a) the ability to execute more than one instruction per cycle; (b) the incorporation of multiple functional units operating in parallel; (c) the inclusion of pipeline mechanism. An important advantage of these features is the ability to exploit the instruction level parallelism by executing concurrently a number of operations in the various pipeline stages and in different functional units[6]. Independently of the mechanism used to extracted these concurrent operations from an essentially sequential instruction stream, the compiler must effectively take an advantage of these features in order to generate a quality code. Register allocation and instruction scheduling play a very important role in this process.

The global register allocation algorithm maps user variables and compiler-generated temporaries to machine registers over an entire procedure. The allocation is considered good if the user variables stays in registers its entire lifetime. Instruction scheduling is the process of moving instructions in order to allow them to be scheduled to the different units of the processor. This process minimize the total execution time and produces code that uses more efficiently the target's pipelines and functional units. The two most important points in the instruction scheduling are: (i) good utilization of the target architecture and

[1]DSc (PUC/RJ - 1994). Departament of Computer Science, Federal University of Minas Gerais, Belo Horizonte - MG - Brazil, E-mail:mariza@dcc.ufmg.br

[2]Ph.D. (Wisconsin, E.U.A). Mathematic Institute of UFRJ and DI/PUC/RJ. E-mail: rangel@inf.puc-rio.br

[3]PhD. (UCLA/USA 1981). Departament of Computer Science, Federal University of Minas Gerais, Belo Horizonte - MG - Brazil, E-mail: bigonha@dcc.ufmg.br.

(ii) preserving the semantics of the program, i.e., a valid scheduling must preserve the execution order described by the edges of graph which gives the instructions dependences.

The most important unresolved problem is to determine the degree of communication between register allocation and instruction scheduling that makes possible to generate a efficient scheduling. Questions like how much these two functions must be integrated to improve the generated code still remains without answer. Nevertheless, work has been done in these subject [3, 4, 2, 21], and [7] makes some suggestions it continues being a big issue. The machine description language issue is another problem in the sense that no languages completely cover the RISC class [7].

The goal of this work is to present the tool based on the ideas proposed by Bradlee [3, 4, 5], which helps the implementation of compilers in the superscalar machine environment. This work comprises: components: (a) the design and formal definition of the semantics and syntax of a machine description language (LDA) that allows specification of instruction scheduling requirements along with other code generation information necessary in superscalar architectures; (b) the project of a retargetable code generators (GGCO) whose objective is to generate automatically tables to directy the code generator kernel from the machine especification of an architecture. A retargetable code generator is one that can be changed automatically from a description of a new target machine, so that it can generate code for that new target; (c) the identification of the level of integration necessary between instruction schedules and register allocation.

2 Compiler Construction Methodology

In the last decade the instruction selection for *Complex Instruction Set Computers* (CISC) was the biggest issue handled in the code generators by compiler developers. Results of this can be seen in systems like PO [12] and successors, in CODEGEN [20]and AutoCode [11]. Since the CISC architectures implement the most common operations in many different ways, their code generators concentrated on machine especifications that allow instructions to be selected by pattern matching [1]. With the advent of the *Reduced Instruction Set* (RISC), the phase of instruction selection of the compiler for these architectures were simplified. In these new processors, all arithmetic, logical, or conditional instructions are register-based. All memory accesses are done with loads and stores, the functional units and pipeline cost are exposed to the code generator. Besides that, the RISC architecture implements most operations only one way. As a consequence, the compiler needs not choose anymore among instructions with multiple addressing modes. Therefore, the compiler's emphasis was shifted from code selection to instructions scheduling and register allocation. As the emphasis has been changed, problems related with the code generators for RISCs architectures become different from the ones related with CISC architectures. Now, to produce a efficient code generator the compiler should capture most scheduling information, like operation latencies and resource conflits. Taking in acount that the scheduler needs registers to overlap the execution of independent operations, it is very important the interaction between register allocation and instruction scheduling. Less attention can be devoted to the interaction between code selection and register allocation

given the relatively simplicity of code selection.

Practically there does not exists retargetable code generators systems especifically designed for RISC architecture. Even less for superscalar machines. The Gnu [24] and Marion systems [3] are the only ones found in the literature. Up today, Marion [3] is the only system that includes a machine description language, but it can not model complicated features found in some superscalar architectures, like the SPARC's register windows, instruction side effects, such as setting the condition code, general multiple instruction issue, and the 88000's resource contention priority scheme.

Until recently, the interpreted machine description present in the GNU system did not contained scheduling information. Actually, there exists at least two GNU versions that include a method to schedule instructions. One of them uses the Gibbons (et al) algorithm [16], in which the register allocation is made before the instruction scheduling and there is no communication between these two phases. The other one includes a algorithm developed by Tiemann [25], with the target-dependency latency and resource information encapsulated.

3 The GGCO System

The GGCO is the code generator generator we have designed. The GGCO design was based on the work of Bradlee [3, 4, 5]. Its architecture, illustrated in Figure 1, comprises the following parts: (a) `MD.c`, which is a file with a set of automatically generated tables and routines. (b) `gen-mdc`, a module containning the semantics of a description in LDA (see Section 5); (c) `MD.h`, a module with the definitions of the most important data structures and types used in the file `MD.c`; (d) The *front-end* modules correspond to the LCC file written by Fraser and Hanson [15]; (e) The *back-end* module [3] contains the instruction scheduling and register allocation strategies. The GGCO system receives as input a processor specification in the machine description language LDA, (see Section 4) and provides automatically a set of tables and functions that represents the result of executing the most important directives of LDA. The `MD.c`, the *front-end* and *back-end* files of Figure 1, are processed by MAKE to produce the *mcc* compiler for the desired architecture. The *mcc* compiler receives a C input file and generates an intermediate language and gives control to code generator that produce an object code semantically equivalent to the input file.

The compiler's front end accepts ANSI C and generates an intermediate language of directed acyclic graph (DAGs). This language provides the initial configuration for the code DAG. The DAG edges represent all possible operators present in the instruction set of the architecture under analysis. The front end transforms all control flow operators (for, while, if, etc.) into low level compare and branch operations. The C language operators with side effects are changed into explicit arithmetic and branch operations.

Each code generator is produced from a machine description of a specifically machine

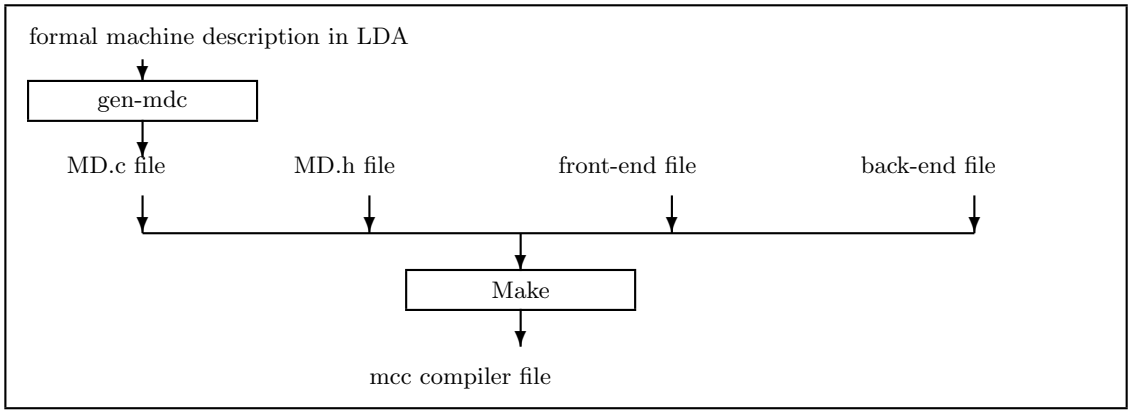

Figure 1: Code Generator Generator Architecture–GGCO

architecture and performs code selection by pattern matching and then moves control to the code generation strategy. The code generation strategy is responsable for: (1) the activation of the instruction scheduling and global register allocation; (2) the degree of communication between these two functions; (3) by the inclusion of the scheduling algorithm.

The GGCO code generation strategy is the same proposed by Bradlee in the Marion system [3]. It consists of two parts: strategy-independent portion, and strategy-dependent portion. The strategy-independent part of the back end posseses three components: the constructor of the code DAG, the global register allocator and the scheduling support. The code DAG builder is responsible for the construction of a DAG from the machine instructions for each basic block. A basic block is a sequence of code that have no internal branching. Scheduling support handles low-level scheduling details, for instance, it controls the list of instructions that can be scheduled without causing a delay, verify the resource conflits. The strategy-dependent part includes the scheduling algorithm, tables and functions generated from the machine specification. Its modular structure permits quick reconfiguration of new strategies of instruction scheduling and register allocation. Based in this modularity, we have incorporated this new strategy in GGCO.

3.1 Instruction Scheduling

The most important data structure in the scheduling process is the scheduling graph, the code DAG. It is represented in this data structure the basic block instructions in which the program was divided. In the code, DAG nodes represent instructions, and directed labeled edges represent dependences between instructions. As the scheduling considered in this project is inside basic block, the precedence restrictions considered are: restrictions based on data dependence and control dependences. Control dependence exists only between basic blocks and their corresponding edges are derived from the control flow graph of the program. The dependence data between instruction can be a true-dependence or a false-dependence. A true-dependence, also called flow dependence, is an edge from a definition to a use. A false-dependence is classified in output dependence and an anti-dependence.

An anti-dependence is an edge from a use to a definition. An output-dependence is an edge between two definitions [27].

The approach used for instruction scheduling is list scheduling [14], [18], [19], [13], [16], [3], [26], [17]. It works as follow: given a code DAG, the scheduler mantains a list of instructions that are ready to be scheduled without causing a delay. On each iteraction it selects the highest priority node in the ready list to be scheduled using a heuristc and then updates the list. According to [3] this approach, in general, has worst-case running time of *O(e)*, where *e* is the number of edges in the DAG, but the heuristic can increase the complexity. All list scheduling algorithms found in the literature use heuristics to assigning priority to nodes in the ready list. The difference among their systems is the order they apply the heuristic. A frequently used heuristic for assigning priority is called *maximum distance*. This heuristic is defined by the length of the longest path through the code DAG from the instruction node to a leaf node. The length of a path is the sum of all edge labels along the path. The idea behind this heuristic is that the node farthest from completion is the most critical, so the others nodes can be scheduled later. Another heuristic gives higher priority to nodes with more successors. The point here is that scheduling a node with several successors creates more opportunities for the scheduler in the next cycles because it permits more nodes to become ready sooner. As third choice there is a heuristic which chooses a node with greater operation latency to have higher priority. than one of its successors. The philosophy is that scheduling such a node first, there will be more opportunities to overlap the latency with other instructions. The GGCO's code generation strategies use list scheduling algorithms with the *maximum distance* as the primary heuristic as in [3].

3.2 Register Allocation

Register allocation problem has been treated in terms of coloring a graph. In this approach, nodes in the graph represent variables and the edges represent interference. Therefore, we connected two variables in the graph if there is a interference between them, i.e., if they cannot simultaneously use the same register at some point in the program. The objective of the register allocator algorithm is to assign a register (*color*) to every variable such that each one has a different color from any of its neighbors [10]. With the advent of the new architecures such as, superscalar, parallel, allowing the parallelism between instructions, an optimal coloring of the interference graph algorithms does not necessarily results in a good machine utilization. It happens because in these architectures it is also necessary to take into account the reordering of the instructions performed by the instruction scheduler algorithm. When instruction reordering is done after register allocation the selection of registers may limit the possibilities to reorder instructions due to false dependencies that are introduced with the reuse of registers. On the other hand, when instructions reordering is made before register allocation the number of live registers is increased, implying longer register lifetimes thus more registers are needed and more spills may be introduced. In addition, in some cases register allocation must precede instruction scheduling since the exact register assignment is needed by the scheduler [19].

Several compilers use different graph models to implement register allocators and instruction schedulers functions [17, 26, 3]. Since the meanings of the nodes and edges in those graphs are different, a simple combination of the graphs is impossible. Nevertheless, the strategy used in GGCO for register allocation and instruction scheduling uses a simple common graph, common framework, *the parallel interference graph*, for representing the input program for both tasks. In this framework the emphasis is on register allocation, and the method used to allocate register is based in the Chaitin work [9]. This strategy was originally proposed by Pinter [23].

Pinter's algorithm works as follows: to generate a parallel interference graph, first we introduce all the scheduling constraints explicitly in the schedule graph. In this approach as much more edges are present in the graph the better the results will be; that happens because what we really going to use is the edges that are *in the complement* of the constructed graph. The edges in the complement graph present the parallelism available in the machine for the given program. The next stage of his algorithm is to integrate those edges with the interference graph. With this new graph the register allocation algorithm can take the available paralellism into account. The scheduling is done after the register allocation.

Since the minimum coloring problem is NP-complete, in general the number of registers is smaller than the number of colors. Thus, in practice a spilling stage is carried out. With this in mind, the problem of register allocation algorithm, for superscalar machine is to find a optimal register mapping with minimum number of register, minimized cost of spilling and whose scheduling graph does not have a false-dependence. To get that it is necessary to apply on the *parallel interference graph* the same heuristics used during register allocation or scheduling. One type of heuristic could eliminate edges from the graph, but to do that it is necessary the knowlege of which edge may be eliminated. It involves consideration of both the scheduler and the allocator. For instance, if it is considered removing edges that prevent false-dependences some parallelization options are lost because of register pressure. On the other hand, it is possible to remove interference edges which may lead to spill and preserve some edges that yield good parallelization.

Chaitin [9] and Pinter [23] algorithms do not deal with register pairs problem. Pairs of registers are often needed in processors to represent half registers in a double precision loads, stores and instructions moves. In their algorithms, a pseudo-register can be removed from the graph if it is garanteed to exist one physical register for it during the coloring phase, which we call unconstrained node. This means that its degree, i.e., the number of its neighbors in the interference graph, is fewer than allocable physical registers. Register pairs change the definition of an unconstrained node. A node now is considered unconstrained if the sum of the physical register requirements of its neighbors plus the number of physical registers required by itself is less than the number of allocable registers. With the introduction of this new definition it is possible to get a good coloring of the *parallel interference graph* even though the demand of the neighbors of a node is greater than the number of allocable registers. The reutilization of colors in neighbors whose edges do not constrain can generate a coloring of this graph preserving the objetive proposed by Pinter, that is, "find a optimal register allocation whose scheduling graph does not have a false dependence".

4 LDA, The Machine Description Language

The GGCO's machine description language, LDA, possesses three main facilities: (a) resources declaration, (b) compiler writer's virtual machine description, (c) instructions definitions. In the declaration section are specified the registers, machine resource, functional units, constants, memory size and other features of the architecture. In the compiler writer's virtual machine are described a runtime model. It offers directives to specify general purpose registers, the pipeline stages, memory, etc. The instruction section introduces each machine instruction, its functions and scheduling requirements. Besides that, it includes tree transformations necessarily to match intermediate language patterns with machine language patterns. To get a perspective of the aplicability of the LDA features and GGCO system as well, we list in the sequel common architectural features not yet treated by other compiler systems. LDA possesses, besides all features of Marion [3], the following resources: (1) facilities to supports register windows. It is possible to specify parameters and arguments separately, to model the register renaming; (2) the machine description language, LDA, establishes the resource necessary for each instructions. With this information LDA constructs a resource vector for each instruction. Each element of the resource vector contains all resources needed on a particular cycle. In the Motorola 88010 processor [22] the priority is defined as follows: integer instructions have the highest priority, then floating point instructions, and lastly load instructions. To solve the structured hazard like a priority scheme to regulate the use of the register write-back-bus of Motorola 88010, GGCO adopts the following scheme suggested by Bradlee [3]: (a) allow a priority range to be associated with a resource declaration in the machine description; (b) allow an element of a resource vector associated with an instruction to indicate its priority; (c) examine priorities when checking for structural hazards and allow schedules to contaim structural hazards, if they are caused by higher priority resources. (3) To avoid control hazards, the LDA specifies the number of delay slots into the instruction directive. To avoid fills branch delay slots with no-ops as in Marion [3], we proposed a Gross and Hennessy [19] algorithm implementation in GGCO, including it as a separated intra-procedural pass after instruction scheduling. This algorithm attempts to fill delay slots with instructions that are before the branch, with instructions that follow the branch target, and with instructions that follow the branch. Gross and Hennessy found that, on a machine whose branches have one delay slot that is always executed, their algorithm filled, counted statically, 90% of the delay slots.

5 Philosophy of the LDA Formal Definition

The formal definition of LDA follows the denotational method for specifying the semantics of programming languages, which defines a set of mapping from the syntactic domains[4] of description to the corresponding code generators. Figure 2 shows the most important mapping of a machine description architecture to its corresponding code generator. This mapping is defined by function `gen-mdc` which specifies the semantics of a description in

[4]Domains can be seen as type being used to model syntax or semantics properties in a given programming language.

LDA. The final result of this mapping is a piece of code in C language, which corresponds to a machine dependent portion of the code generator for the described processor.

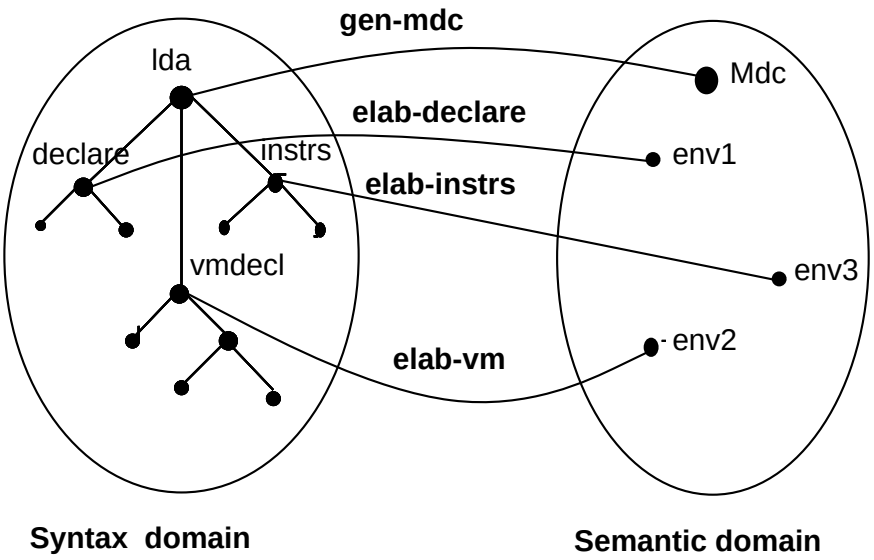

Figure 2: Mapping of LDA in Mdc.

The `gen-mdc` function receives as input a file with the LDA specification in the form of abstract syntax tree, ativates the functions: `elab-declare`, `elab-vmdecl`, `elab-instr` and `elab-tables` to elaborate the several parts of a machine description and generate the machine dependent code of the code generator. This formal definition can be seen as a code generator generator system produced from a machine description. The piece of code generated is stored in `MD.c` file. The function `elab-declare` elaborates the LDA declarations; the `elab-vmdecl` function elaborates the virtual machine information; the function `elab-instr` elaborates the instructions of LDA, and the function `elab-tables` elaborates the final tables, producing the C code. The complete formal especification of LDA can be found in [7].

In the specification of this mapping the most important semantic domain is *env*, which defines the environment where the formal definition are estabilished, i.e., defines a tuple composed of the identification of all generated auxiliary entities to the final tables elaboration. This domain comprises a set of tables and lists constructed from each LDA specification. It is also part of a signature of most semantic functions.

The present formal definition is written in $\mathcal{SCRIPT}$ [8], which is a functional language that offers a simple notation to describe the denotational semantics of a programming language in modular and legible style. The abstract syntax is defined in $\mathcal{SCRIPT}$ as a production list of a context free grammar. Non-terminals represent syntactic domains and the tokens are "quotations" domains. The formal definition of all semantics domains are grouped in modules, which control the visibility of their denotations and provide the services associated with each domain. For each of the most important semantic domains used in the definition of mapping `gen-mdc` there exist a module $\mathcal{SCRIPT}$, which encapsulates their denotations and provides the associated services.

The most high level function of the LDA formal definition, the function `gen-mdc`, maps LDA description into portions of C code, which comprise the machine dependent part of the code generator. This portion of program stored in a `MD.c` file includes the data structure and types definitions declared in the previous `MD.h` file (see Section 3).

The most important generated tables are: table of resources and productions. The resource table describes the resources used by instructions and is one of the most important information to scheduler. Its contents is used essentially in the basic routines of instruction scheduling to verify existents conflicts and to group instructions. The production table is an array that contains instructions information. Each array element corresponds to an instruction directive given in the description and contains: (a) a pattern tree and a replacement symbol derived from the expression given in the directive; (b) an array that indicates, for each instruction, its operand kind and their location within the pattern and subject trees; (c) an index into an array of resource vectors; and (d) cost, latency and delay slot data. Information in this table are used by the code generator when performing instruction scheduling, register allocation and code selection. Other generated tables contain declarations, information about the virtual machine, classes and elements, auxiliary latencies, patterns, etc.

The two most important target-dependent functions are the one that returns the general purpose register set for a given type, and the one that returns register overlapping information and register sizes. These informations are used during the creation of pseudo-registers and register allocation.

6 Conclusions

It was presented in this paper the facilities provided by a code generator generator, GGCO. The most important contribution is the formal specification of the processor dependent part of the code generator by means of a special purpose language. Its formal definition may be seen as a code generator system which provides from the machine description the mapping of the processor to its code generator, and whose final result is piece of C program, which corresponds to the dependent machine part of the code generator for the architecture described.

The system prototype is not yet completely implemented. For instance, the facilities to support register windows of sun station SPARC permit only one window. Particularly: (a) additional studies must be done before the incorporation of the register allocation algorithm presented in Section 3.2 in the system in order to evaluate and compare it with the algorithm proposed by Bradlee for scheduling and register allocation; (b) implementation of Gross and Henessy algorithm is needed to fill the *delay slots* with valid instructions during scheduling; (c) implementation of the proposed solution to attend the priority schema present in some superscalar architecture is still yet to be accomplished. On the other hand, all above facilities are already available in LDA.

References

[1] Aigrain, P. and others, *Experience with a Graham–Glanville Code Generator*, Proceedings of the Sigplan'84 Symposium on Compiler Construction, pages: 13-24, ACM Sigplan Notices 19(6), june/1984.

[2] Benitez, Manuel E. and Davidson, Jack W., *A Portable Global Optimizer and Linker*, ACM Sigplan Notices, 23, 7, July/1988.

[3] Bradlee, David G. et al., *The Marion System for Retargetable Instruction Scheduling*, ACM Sigplan Conference on Programming Language Design and Implementation, ACM Sigplan Notices 26(7), July/1991.

[4] Bradlee, David G. et al., *Integrating Register Allocation and Instruction Scheduling for RISCs*, ASPLOS-IV Proceedings - Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, April/1991.

[5] Bradlee, David G., *Retargetable Instruction Scheduling for Pipelined Processors*, University of Washington, 1991, Departament of Computer Science and Engineering,FR-35.

[6] Benitez, Manuel E. and Davidson, Jack W., *Code Generation for Streaming: an Access/Execute Mechanism*, ASPLOS-IV Proceedings - Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 1991, Santa Clara California, April.

[7] Bigonha, Mariza A. Silva, *Otimização de Código em Máquinas Superescalares*, Tese de Doutorado, DI-PUC/RJ, Abril/1994.

[8] Bigonha, Roberto S. , $\mathcal{SCRIPT}$ An Object Oriented Language for Denotational Semantics (User's Manual and Reference), DCC - UFMG, RT 03/94.

[9] Chaitin, Gregory J., *Register Allocation and spilling via graph coloring*, ACM Sigplan Notices, 17, 6, ACM Sigplan Symposium on Compiler Construction, June/1982.

[10] Callahan, David and Koblenz, Brian, *Register Allocation via Hierarchical Graph Coloring*, ACM Proceeding of the ACM Sigplan'91 Conference on Programming Language Design and Implementation. Toronto, Ontario, Canada, 26-28, 1991. 192-203

[11] Costa, Paulo S. S., *Um Gerador Automático de Geradores de Código*, Tese de Mestrado, PUC-RJ, Fevereiro/1990.

[12] Davidson, Jack W., *Simplifying Code Generation Through Peephole Optimization*, DCSc - The University of Arizona, Tucson Arizona, December/1981.

[13] Fisher, Joseph A. et al., *Parallel Processing: A smart compiler and a dump machine*, ACM Sigplan Notices, 19, 6, Proceedings of the ACM Sigplan, Symposium on Compiler Construction, June/1984,

[14] Fisher, Joseph A., *Trace Scheduling: A Technique for Global Microcode Compactation*, IEEE Transactions on Computers, 30, 7, July/1981.

[15] Fraser, Christopher W. and Hanson, David R., *A Code Generation Interface for ANSI C*, Department of Computer Science, Princeton University, 1992, Research Report, CS-TR-270-90, September, Last Revised September 1992.

[16] Gibbons, Phillip B. and and Muchnick, Steven S., *Efficient Instruction Scheduling for a Pipelined Architecture*, Proceedings of the ACM Sigplan'86 - Symposium on Compiler Construction, ACM Sigplan Notices 21(7), July/1986.

[17] Goodman, James R. and Wei-Chung-Hsu, *Code Scheduling and Register Allocation in Large Basic Blocks*, International Conference on Supercomputing - Conference ACM-PRESS Proceedings, St. Malo France, July/1988.

[18] Hennessy, John and Gross, Thomas, *Code Generation and Reorganization in the Presence of Pipeline Constraints*, Conference Record of the 9^{th} Annual ACM Symposium on Principals of Programming Languages, 128-133, Albuquerque New Mexico, January/1982.

[19] Hennessy, John and Gross, Thomas, *Pospass Code Optimization of Pipeline Constrains*, ACM Transactions on Programming Languages, 1983, 5(3).

[20] Henry, Robert R., *Code Generation by Table Lookup*, Computer Science Department, University of Washington, Technical Report, # 87-07-07, FR-35 Seattle, WA 89195 USA, July/1987.

[21] Kerns, Daniel R., *Balanced Scheduling: Instruction Scheduling when Memory Latency is Uncertain*, Proceedings of the ACM Sigplan'93 Conference on Programming Language Design and Implementation, Albuquerque NM, ACM Sigplan Notices - Vol. 28 number 6 june-1993.

[22] Motorola, Inc., *MC88100 RISC Microprocessor User's Manual*, Prentice-Hall, Englewood Cliffs, second edition, New Jersey/1990.

[23] Pinter, Shlomit S., *Register Allocation with Instruction Scheduling: a New Approach* , Proceedings of the ACM Sigplan'93 Conference on Programming Language Design and Implementation, Albuquerque NM, June/1993, ACM Sigplan Notices - Vol. 28 number 6 june-1993.

[24] Stallman, Richard M., *Using and Porting GNU C*, Free Software Foundation Incorporation, Cambridge Massachusetts, 1989

[25] Tiemann, Michael D., *The GNU Instruction Scheduler*, Stanford University, Class Report, CS 343, June/1989.

[26] Warren Jr, H. S., *Instruction Scheduling for the IBM RISC System/6000 Processor*, IBM Journal of Research and Development, 34, 1, January/1990.

[27] Fernandes, Edil T and Santos Anna Dolejsi, *Arquiteturas Super Escalares: Detecção e Exploração do Paralelismo de Baixo Nível*, /vIII Escola de Computação, 3 a 12 de agosto, Gramado -RS, 1992.