Code-generator generators: generating the instruction selector

Clément Vasseur <clement.vasseur@lrde.epita.fr>

Technical Report n^o0403, 1.5, 06 2004

The main part of a compiler back-end is the instruction selection. Its goal is to generate the assembly code for the target CPU from an intermediate representation of the program.

There are several methods for performing an efficient instruction selection in an automatic way. So it is possible to generate the code-generator, similar to the way parsers are generated. That is, the code generator is generated from a specification that gives the links between the trees of intermediate representation and the instructions that should be emitted.

First, we will see how the selection of instructions works, and what algorithms are involved. Then we will explain how they can be automated and generated.

Keywords
compilation, instruction selection, code-generator, burg

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
lrde@epita.fr – http://www.lrde.epita.fr

Copying this document

Copyright © 2004 LRDE.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being just "Copying this document", no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is provided in the file COPYING.DOC.

Contents

1 Introduction **5**

2 Instruction selection **6**
2.1 Inside the compiler . . . 6
2.1.1 Intermediate representation . . . 6
2.1.2 From trees to assembly instructions . . . 8
2.2 Maximal-Munch . . . 9
2.2.1 The algorithm . . . 9
2.2.2 Discussion . . . 9
2.3 Dynamic programming . . . 10
2.3.1 Principle of dynamic programming . . . 10
2.3.2 The algorithm . . . 10
2.3.3 Discussion . . . 11

3 Generating the code-generator **12**
3.1 History of code-generator generators . . . 12
3.1.1 *Graham* and *Glanville* - 1980 . . . 12
3.1.2 *Davidson* and *Fraser* - 1984 . . . 12
3.1.3 *TWIG* - *Aho*, *Ganapathi* and *Tjiang* - 1989 . . . 12
3.1.4 *BURG* - *Fraser*, *Henry* and *Proebsting* - 1992 . . . 12
3.2 *BURG* tree-grammar specification . . . 13
3.3 Generator implementation . . . 13
3.4 The `label` function . . . 13
3.4.1 The bottom-up traversal . . . 14
3.4.2 Arrays initialization . . . 14
3.4.3 Performing the actual work . . . 14
3.4.4 Handling the `ADDRLP` node . . . 15
3.4.5 Handling the `CVCI` node . . . 15

4 Other approaches **16**
4.1 BURG . . . 16
4.2 MBURG . . . 16
4.3 WBURG . . . 17
4.4 GBURG . . . 17

5 Implementation for the Tiger compiler **18**
5.1 The intermediate representation 18
5.2 Monoburg 18
5.3 Further work 18

6 Conclusion **20**

7 Bibliography **21**

Chapter 1

Introduction

The code generation, or instruction selection, in the context of a compiler, is the part that deals with the generation of the assembly code. Given an input program, the compiler generally translates it in assembly language, so that it can be assembled to an executable for a given processor.

This process, which takes place in the back-end of the compiler, exhibits the following characteristics:

- First, it is a **critical** part of the compiler. Indeed, if there is an error somewhere in the code generation, the resulting binary won't run properly.
- Then, it is quite **difficult** to implement, because the CPU instruction set must be known. The programmer must be aware of the various mnemonics and addressing modes for this particular instruction set.
- Moreover, the code selector is **hard to debug**, because its output is a lot of assembly code, and there is no simple automatic way to verify that the generated binary behaves exactly like specified by the input source code.
- Finally, the code-generator itself is **specific** for each architecture supported by the compiler. It must be rewritten each time a new CPU needs to be supported.

All these characteristics make one wonder if one could not generate the code generator instead of implementing it by hand. Similarly, the lexer and the parser are already generated most of the time. So, it seems quite natural to look for a way to derive the code-generator from a concise specification.

This technical report consists of three major parts. The first part covers the instruction selection pass, introducing the algorithms involved in this process, that one generally implements by hand. The second part deals with the automation of the code generation, in this part the instruction selector is generated. The third part covers the related work, the other well-known methods that can achieve the same goal.

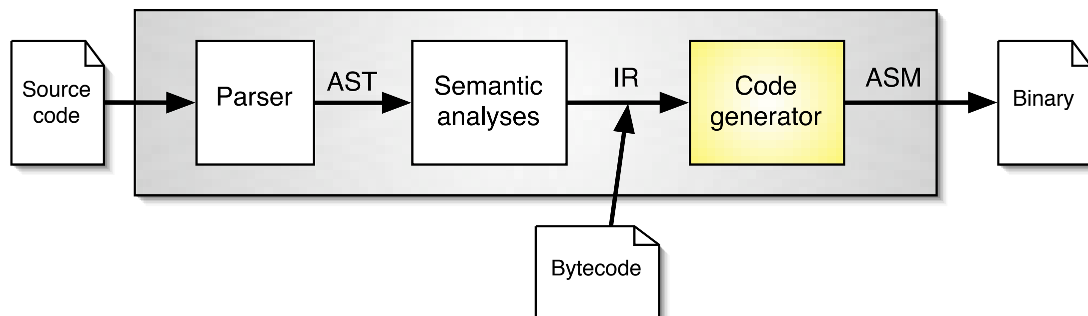
Chapter 2

Instruction selection

This chapter is about the instruction selection pass in a compiler. We will explain what it is and how it can be implemented by hand using a simple algorithm which is not optimal, then we will see the optimal one.

2.1 Inside the compiler

Let's consider a schematic view of a compiler. Instruction selection is the part that converts the intermediate representation of the program to assembly instructions.



AST: Abstract Syntax Tree.

IR: Intermediate Representation.

ASM: Assembly instructions.

2.1.1 Intermediate representation

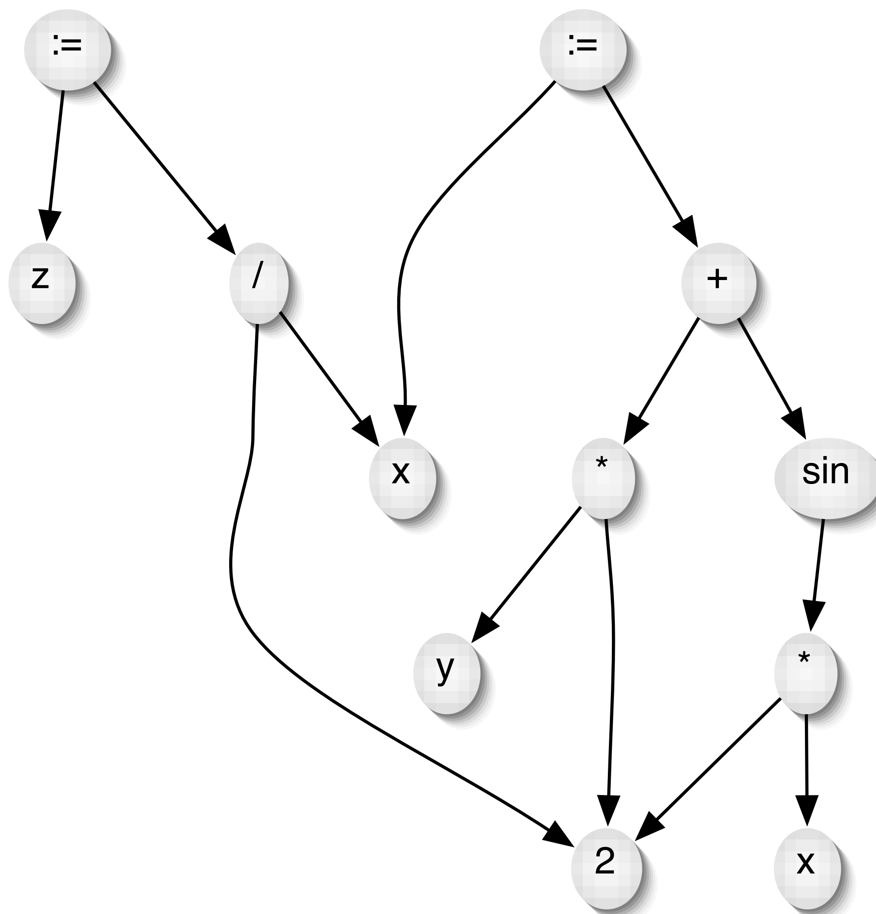
In a compiler, several intermediate representations are possible. First, the intermediate representation can be of different levels:

- **Higher level** than the instruction set: the nodes express high-level concepts like source-level variables, FOR and WHILE loops. This representation is close to the source language. Thus, it is not convenient for code selection because there is a big difference between the nodes and the instructions that should be generated.

- **Same level** as the instruction set: each node in the intermediate representation tree can be expressed using one or several assembly instructions.
- **Lower level** than the instruction set: an assembly instruction can cover several nodes, because the intermediate representation is only made of atomic operations.

Then, there are several different forms of intermediate representation:

- **Directed Acyclic Graph (DAG)**: in this representation, a node that represents a value is shared between it's definition and its uses. This intermediate representation keeps the memory footprint low, but it is quite complicated to work with. More specifically, general code selection for a DAG is NP-complete Ertl (1999).

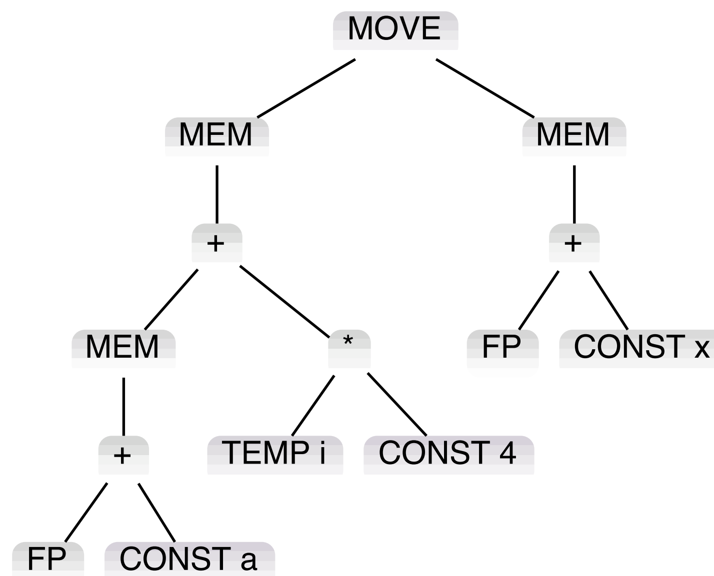



```
x := 2 * y + sin(2 * x)
z := x / 2
```

- **Three-address code**: this representation resembles assembly instructions, with mnemonics and operands. Most statements look like `x := y op z`. It can be in SSA (Static Single Assignment) form Cytron et al. (1991), where each temporary is assigned exactly once. This can be very useful for performing code improvement.

```
L1: i := 2
    t1 := i + 1
    t2 := t1 > 0
    if t2 goto L1
```

- **Trees**: in this representation, the program is a forest. Each node represents an operation, with its children being the arguments, thus making the temporary results locations implicit.



In this report, we work with a low-level tree-based intermediate representation. Effectively, the input program is translated into a list of trees, and each of these trees will generate an amount of assembly instructions.

2.1.2 From trees to assembly instructions

In order to make this generation possible, each machine instruction is represented as a tree pattern. Then, the goal is to tile the intermediate representation tree with the instruction patterns. Of course, the tiling needs to be as efficient as possible, with respect to a cost. The cost is chosen arbitrarily. It can be good to optimize for code size or code speed, or something else.

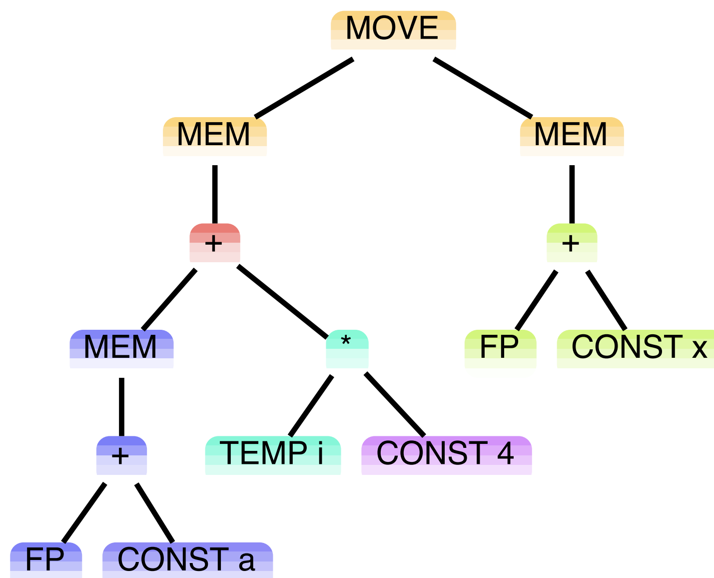
Also, the difference between an optimal and an optimum tiling must be clear. The tiling is optimal if for two successive nodes there can't be another tiling that yields a smaller cost. It is optimum if for all the nodes there can't be another tiling that yield a smaller cost. An optimum tiling is optimal, the opposite is not true.

2.2 Maximal-Munch

The Maximal-Munch instruction selection is a greedy algorithm that covers the intermediate representation tree in a single pass.

2.2.1 The algorithm

The algorithm consists in a top-down pattern matching. The intermediate representation tree is walked recursively, starting from the root node. At each step, the largest instruction pattern that fits the node is chosen, and the corresponding instruction is generated. Then the algorithm recurses with the nodes matching the leaves of the tree pattern corresponding to the chosen instruction.



- $r_1 \leftarrow M[fp + a]$
- $r_2 \leftarrow r_0 + 4$
- $r_2 \leftarrow r_i \times r_2$
- $r_1 \leftarrow r_1 + r_2$
- $r_2 \leftarrow fp + x$
- $M[r_1] \leftarrow M[r_2]$

2.2.2 Discussion

First, it is noticeable that the machine instructions are generated in reverse order. Indeed, the instruction corresponding to the root node needs the results of the subtrees in order to perform its calculation. Thus, the instruction pattern that fits the root node is the first one which is selected, but the last one to be executed.

Moreover, it can be proved that this algorithm produces an optimal tiling in linear time. Effectively choosing the largest tile that fits the root node ensures local optimality. Indeed, if there

was a larger tile that matched the same as two smaller tiles, it would have been selected in the first place.

However, the Maximal-Munch instruction selection does not necessarily find the global minimum. That is, it does not guarantee an optimum tiling of the intermediate representation tree. Indeed, the algorithm selects the first instruction pattern that fits the root node, without taking into account its subtrees. So, a really big pattern can be missed below the top node.

Finally, the Maximal-Munch algorithm is very efficient, and performs a good code selection for RISC architectures.

2.3 Dynamic programming

In this section, we will explain an algorithm that performs an optimum tiling of the intermediate representation tree, using the well-known approach of dynamic programming.

2.3.1 Principle of dynamic programming

The general principle of dynamic programming allows to solve an optimization problem by caching sub-problems solutions rather than recomputing them.

Let's take the general approach of "Divide-and-Conquer" problem solving. This approach is a top-down technique which starts from the initial problem as a whole, and progresses recursively to the sub-problems. The disadvantage of this method is that recursively solving the sub-problems may perform the same computations several times because several sub-problems may be identical.

When using dynamic programming, this problem does not appear because the approach is bottom-up and the intermediate results are memorized, thus avoiding to recompute identical sub-problems.

2.3.2 The algorithm

The algorithm consists in two steps.

- First, a **bottom-up** traversal is applied in order to assign a cost to each node. This cost represents the total cost of the instructions emitted for the whole tree starting at the current node. For example, if we chose to optimize for code size, the cost of a node would be the total size of the instructions emitted for the tree rooted at this node.

  The cost assignment proceeds like this:

  - for each tile t of cost c that matches at node n
  - c_i = cost of each subtree corresponding to the leaves of t
  - cost of $n = c + \sum c_i$

- Then a **top-down** traversal selects the minimum cost tiling. Starting from the root node, the best instruction pattern, as determined by the previous traversal, is selected (its corresponding instructions are generated). This process is applied recursively to the leaves of the current pattern.

2.3.3 Discussion

This algorithm produces an optimum tiling of the tree. At each stage, it uses the best results of the subtrees in order to select the best instruction pattern for the current node. It is also quite efficient, since it caches the intermediate results for all the nodes.

Chapter 3

Generating the code-generator

In this chapter, the generation of the code-generator is covered. We choose to explain the approach developed in *IBURG* Fraser et al. (1992a), because it is very simple yet quite powerful since it produces optimal code-generators.

3.1 History of code-generator generators

3.1.1 *Graham* and *Glanville* - 1980

One of the very first code-generator generation method that proved usable is the *Graham-Glanville* method Glanville and Graham (1978). In this method, linearized tree patterns are recognized using a bottom-up shift-reduce parser. A $LR(0)$ grammar was automatically derived from the machine instruction set. The biggest problem with this approach is that grammars for *Graham-Glanville* code-generators are highly ambiguous. Thus, the disambiguation of the grammar could introduce blocking states in the parser.

3.1.2 *Davidson* and *Fraser* - 1984

Another method for code selection based on code optimization Davidson and Fraser (1984) was devised. This method does not try to select good code from the tree directly. First it locally macro-expands the tree into naive machine instructions. Then, the resulting code is incrementally improved using a set of declarative optimizations, like a peephole optimizer. This method is still in use today, in GCC (the GNU Compiler Collection) FSF (1987).

3.1.3 *TWIG* - *Aho*, *Ganapathi* and *Tjiang* - 1989

TWIG Aho et al. (1989) introduced the principles of tree pattern matching and dynamic programming. This approach is very similar to the one explained in this report. However, *TWIG* uses a table-driven pattern matching, which is more complicated than raw code generation.

3.1.4 *BURG* - *Fraser*, *Henry* and *Proebsting* - 1992

The *BURG* Fraser et al. (1992b) code-generators generator is based on *BURS* (Bottom-Up Rewrite System) theory Nymeyer and Katoen (1997), in order to move the dynamic programming to compile-compile time. The generator is more complicated, but it is very efficient. *IBURG* is a

variant of *BURG* that does the dynamic programming at compile time, using a straight-forward method.

3.2 *BURG* tree-grammar specification

The *BURG* tree-grammar specification is considered a standard for code-generators generators. It consists in a cost-augmented tree-grammar, with an action corresponding to each rule. The action is used to effectively emit the machine instructions. In the following example, costs are noted between parenthesis at the end of the line. When there is none, the default cost is 0. Actions are specified after the '=' symbol.

```
stmt: ASGNI(disp,reg) = 4 (1);
stmt: reg = 5;
reg: ADDI(reg,rc) = 6 (1);
reg: CVCI(INDIRC(disp)) = 7 (1);
reg: IOI = 8;
reg: disp = 9 (1);
disp: ADDI(reg,con) = 10;
disp: ADDRLP = 11;
rc: con = 12;
rc: reg = 13;
con: CNSTI = 14;
con: IOI = 15;
```

3.3 Generator implementation

The code-generators generator is a software component that takes as input a specification (a *BURG* tree-grammar in the case of *IBURG*), and generates a software module that plugs into the compiler. This module reads an intermediate representation tree and generates machine code as a result.

We have seen that an *IBURG*-generated code-generator performs two passes over the intermediate representation tree. The first pass, `label`, assigns a cost corresponding to the best instruction pattern, to each node, in a bottom-up traversal. The second pass, `reduce`, executes the action associated with the best instruction pattern selected for each node, in a top-down traversal. This second pass is generic and does not depend on the grammar. Most of the time, the compiler writer builds this part himself. So we will focus on the first pass, performed by the `label` function.

As can be seen in the grammar snippet, several non-terminal symbols are used in order to specify how instruction patterns fit together. This introduces a complexity which prevents us from using the simple form of dynamic programming explained earlier, with one cost per node. The next section gives a solution for this problem.

3.4 The `label` function

The `label` function is the function that does the labeling pass. All the code generation smartness goes in it. This function takes a node of the intermediate representation as an argument, and its goal is to assign a cost to this node, based on the selection of the best instruction pattern.

In order to cope with the complication of using several non-terminal symbols in the *BURG* grammar, the best cost of the node must be kept for each non-terminal symbol. Thus, an array of costs should be included in the data structure used for the nodes. Moreover, the selected instruction patterns must be memorized as well, in order for the reduction pass to execute the appropriate action. This leads to the inclusion of another array in the node data structure. This array keeps for each non-terminal symbol the action number associated with the best instruction pattern.

3.4.1 The bottom-up traversal

```
function label(node p)
  foreach child q of node p
    label(q)
```

The first step, as we are doing a bottom-up traversal, is to recurse into the children of the current node, in order to assign the costs for all the trees at the leaves of the instruction patterns that will be tested on the current node.

3.4.2 Arrays initialization

```
  foreach i in 0..N
    p.rule[i] = 0
    p.cost[i] = MAX_COST
```

Then, the `rule` and `cost` arrays have to be initialized. They respectively keep the actions and the costs associated with the best instruction pattern for the current node. `rule` is initialized to 0 because no instruction pattern have been selected yet, and `cost` is initialized to `MAX_COST` in order to select the next instruction pattern whatever its cost. `N` represents the number of non-terminal symbols in the grammar, it is also the size of the two arrays.

3.4.3 Performing the actual work

```
  switch (p.node_type)
```

The body of the function, actually doing all the work, is a dispatch depending on the type of the current node. Then, code must be generated for each possible type of node, in order to fill the `rule` and `cost` arrays using the best instruction pattern. In this report, we will explain two examples of node types corresponding to the *BURG* grammar given earlier: `ADDRLP` and `CVCI`.

First, let's introduce an helper procedure that will be useful for dealing with the nodes in the `switch` statement: `record`.

```
function record(node p, integer nt, integer cost, integer rule)
  if cost < p.cost[nt] then
    p.rule[nt] = rule
    p.cost[nt] = cost
  endif
```

This is the `record` procedure used in the rest of the code. It is quite trivial, it just records the `rule` and `cost` for non-terminal `nt` if the cost is lower than the one already registered.

3.4.4 Handling the **ADDRLP** node

Now we can move on to the ADDRLP node handling. Remember the rule for ADDRLP:

```
disp: ADDRLP = 11;
```

ADDRLP node gives a disp non-terminal symbol, with cost 0, associated with action 11.

```
case ADDRLP:
  c := 0
  record(p, disp, c, 11)
  record(p, reg, c + 1, 9)
  record(p, rc, c + 1 + 0, 13)
  record(p, stmt, c + 1 + 0, 5)
```

The first call to record precisely registers what can be seen in the grammar rule. As for the other calls, we should recall a few other rules of the grammar to understand them:

```
stmt: reg = 5;
reg: disp = 9 (1);
rc: reg = 13;
```

Those rules are injections of non-terminals, those chain-rules simply derive one non-terminal from another. Since an ADDRLP node gives a disp non-terminal, we must also register every non-terminal symbol that can be derived from disp. These rules show that reg, rc and stmt can lead to disp, which explains the calls to record in the above code.

3.4.5 Handling the **CVCI** node

Now let's see how to handle a CVCI node. This one is interesting because the instruction pattern matches several nodes. First recall the grammar:

```
reg: CVCI(INDIRC(disp)) = 7 (1);
```

The code for this node must recognize the CVCI(INDIRC(disp)) pattern:

```
case CVCI:
  if p.child[0].node_type = INDIRC
    and p.child[0].child[0].rule[disp] != 0
  then
    c := p.child[0].child[0].cost[disp] + 1;
    record(p, reg, c, 7)
    record(p, rc, c + 0, 13)
    record(p, stmt, c + 0, 5)
  endif
```

The cost for this node is based on the cost of the tree corresponding to the disp non-terminal. Then, there are the usual injections coming from the reg non-terminal.

Chapter 4

Other approaches

This chapter gives an overview of some related work. Indeed, *IBURG* is not the only way to generate a code-generator, some other methods have been created to fulfill other needs.

4.1 BURG

BURG Fraser et al. (1992b) was the first code-generators generator to build up on the *BURS* (Bottom-Up Rewrite System) formalism. It was developed by *Fraser*, *Henry* and *Proebsting*, in 1992. The major difference between *BURG* and *IBURG* is that the former performs the dynamic programming at compile-compile time (ie: during the generation of the code-generator), while the later does it at compile time.

Dynamic programming is achieved at compile-compile time by encoding the unbounded number of tree configurations into a finite set of equivalence classes. The detailed explanation of the technique used in *BURG* goes beyond the scope of this report, since we chose to explain *IBURG* which is simpler.

There are two disadvantages when comparing this method to *IBURG*. First the cost associated with each rule in the grammar must be static, because they are used in compile-compile time. In *IBURG* they can be dynamic, evaluated during the code generation. Then, the generation of a *BURG* matcher takes a long time. However, *Proebsting* explained several techniques to make the *BURG* generator go faster Proebsting (1992).

4.2 MBURG

MBURG Gough and Ledermann (1997) is very similar to *IBURG*. It can select optimal code on trees in two passes: first a labeling pass, then a reduction pass.

Although *MBURG* uses the same concepts as *IBURG*, the implementation is different. First, it produces its output in *ISO Modula-2*. Then, two major improvements are provided. First, the labelling is incremental, it is performed when the intermediate representation nodes are created, thus avoiding the recursion of a bottom-up traversal. Then there are forced reductions which are performed during the labelling pass, which can help significantly when register allocation takes place during instruction selection, by doing some adaptations in the tree during the first pass.

4.3 WBURG

WBURG Proebsting and Whaley (1995) generates code-generators that produce an optimal code in a single bottom-up pass. This is an advantage compared to the two passes required by an *IBURG*-style code-generator because it allows to get rid of the intermediate representation.

Indeed, the labeling pass being a bottom-up process, it can be performed when the intermediate representation is created. Since the code-generator produced by *WBURG* does the labelling and the reduction in a single pass, the need for a intermediate tree is eliminated. This results in space and time savings.

In order to make a one-pass traversal possible, a *WBURG*-generated matcher buffers a small fixed-size stack of previously seen operations for deferred matches. The drawback of the method is that it only supports a proper subset of the grammars handled by two-pass systems. However it can handle the most common instruction sets (SPARC, MIPS R3000, x86).

4.4 GBURG

The *GBURG* Fraser and Proebsting (1999) (Greedy Bottom-Up Rewrite Generator) code-generators generator was developed for a very specific purpose. This generator targets the field of *JIT* (Just-In-Time) compilation, in virtual machines. The requirements in this environment are different than for a standard compiler. The generation time is at least as important as the execution time, since code generation takes place during the execution.

In order to minimize the generation time, several ideas are used. First, the generator must be small. The smaller it is, the more it can fit in the CPU cache, thus considerably accelerating its execution. A code-generator generated by *GBURG* for x86 can be as small as 8 KB, allowing it to fit entirely in an 8 KB I-cache.

Another idea to speed up the code generation is to sacrifice the quality of the generated code in order to use a finite-state machine to emit instructions in one pass. This involves using a simplified grammar (regular expressions), working on a linearized postfix intermediate representation (the bytecode generally used in virtual machines has this property), and using a greedy pattern matching for selecting the instructions.

As a result, *GBURG* generates code-generators that emit x86 instructions at 3.6 MB/sec on a 266 Mhz P6 machine. In comparison, an *IBURG*-generated instruction selector, like the one explained in this report, emits code at 1.8 MB/sec on the same machine.

Chapter 5

Implementation for the Tiger compiler

All the bibliographic research exposed in this report takes place in the context of an implementation in a Tiger compiler. The Tiger language is specified by Andrew Appel in "Modern Compiler Implementation in C, Java, ML" Appel (1998). We wanted to replace the hand-written code-generators in our compiler with a generated one, using a simple technology that could be taught in compilation lectures.

5.1 The intermediate representation

The intermediate representation we are working on is the LIR (Low level Intermediate Representation). An example of such representation was given in section 2.1.1.

Our goal is to produce MIPS assembly language. The MIPS CPU, being a RISC architecture, is really simple and orthogonal. The grammar for describing the mapping between LIR and MIPS instructions should not require a lot of work and must be accessible for the students. Moreover, register allocation is deferred completely after the instruction selection pass, using an unlimited set of registers. This considerably simplifies the process.

5.2 Monoburg

Monoburg is an implementation of *IBURG*, developed in the context of Mono Novell (2004), the open source effort sponsored by *Novell* to create a free implementation of the .NET Development Framework. It was created in order to generate the code-generator for the Mono Virtual Machine, which uses *JIT* (Just-In-Time) compilation.

Since this code-generators generator is simple, maintained, easily available and free, it seemed like a good candidate for our Tiger compiler.

5.3 Further work

A prototype grammar has already been written, but it is not yet integrated in the compiler. Since our Tiger compiler is written in C++, it makes use of object-oriented data structures. These data

structures must be modified to be able to host a *Monoburg*-generated code-generator, and this change must not be intrusive to the other parts of the compiler. The work on this part is on-going, and shouldn't take more than a few week to complete.

Chapter 6

Conclusion

To conclude, we have seen that code-generator generation is practical. Indeed, several techniques have been developed to fulfill different needs and different requirements. As a result, code-generator generators are widely used in a lot of production compilers today, as well as *JIT* compilers for virtual machines.

The generation of the instruction selection pass introduces several benefits. First, the quality of the code generated by the compiler is better, because the code-generator is known to perform on optimal code selection, when using the algorithm based on dynamic programming. Then, the maintenance of the compiler is easier, because the programmer manipulates a declarative grammar instead of lines of codes. Last but not least, using a code-generator generator eases the process of retargetting the compiler to a different CPU. Indeed, changing the grammar is much easier than writing a new code-generator in the back-end.

Anyway, the more we generate, the better it is.

Chapter 7

Bibliography

Aho, A. V., Ganapathi, M., and Tjiang, S. W. K. (1989). Code generation using tree matching and dynamic programming. *ACM Trans. on Programming Languages and Systems*, 11:491–516.

Appel, A. W. (1998). *Modern Compiler Implementation: In ML*. Cambridge University Press.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490.

Davidson, J. W. and Fraser, C. W. (1984). Code selection through object code optimization. *Transactions on Programming Languages and Systems*, 6(4):506–526.

Ertl, M. A. (1999). Optimal code selection in DAGs. In *Principles of Programming Languages (POPL '99)*.

Fraser, C. W., Hanson, D. R., and Proebsting, T. A. (1992a). Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226.

Fraser, C. W., Henry, R. R., and Proebsting, T. A. (1992b). Burg – fast optimal instruction selection and tree parsing. *ACM SIGPLAN Notices*, 27(4):68–76.

Fraser, C. W. and Proebsting, T. A. (1999). Finite-state code generation. *ACM SIGPLAN Notices*, 34(5):270–280.

FSF (1987). Gcc - gnu compiler collection.

Glanville, R. S. and Graham, S. L. (1978). A new method for compiler code generation. In *5th POPL conference record*.

Gough, K. J. and Ledermann, J. (1997). Optimal code-selection using MBURG. *Australian Computer Science Comm.: Proc. 20th Australasian Computer Science Conf., ACSC*, 19(1):441–450.

Novell (2004). Mono.

Nymeyer, A. and Katoen, J.-P. (1997). Code generation based on formal BURS theory and heuristic search. *Acta Informatica*, 34(8):597–635.

Proebsting, T. A. (1992). *Code Generation Techniques*. PhD thesis, University of Wisconsin - Madison.

Proebsting, T. A. and Whaley, B. R. (1995). One-pass, optimal tree parsing - with or without trees.