

A Simple Tree Pattern Matching Algorithm for Code Generator

Tzer-Shyong Chen, Feipei Lai, and Rung-Ji Shang
Dept. of Electrical Engineering
National Taiwan University, Taipei, Taiwan, R.O.C.
E-mail: flai@cc.ee.ntu.edu.tw

Abstract

This paper describes a simple tree pattern matching algorithm for the code generator of compilers. The intermediate code (Register Transfer Language) is matched with the tree-rewriting rules of the instruction description which describe the target architecture to generate the assembly code. The hashing function is used in our system to transform a tree pattern matching problem into a simple number comparison. Compared with GNU C compiler (gcc), the tree pattern matching time can be reduced by 69% and the compiler time by 6%, and the space of the instruction descriptions can be reduced by 4.10 times on DLX and 2.14 on SPARC. The size of table, which is necessary for code generator, is quite small in our method.

1 Introduction

Code selection can be done by the tree pattern matching. Through instruction patterns, target instructions are first described, then the tree pattern matcher searches for a cover [5] of the input tree. However, if there are several possible covers in a given input tree, this process usually becomes indetermined. The cost of the instruction patterns indicates the quality of code such as execution time or code size. By choosing the code according to the cost, the ambiguity of code selection is resolved.

There are several methods to select the cover of the minimum cost. Graham and Glanville [12] propose a concept about the use of *LR* parsing. In this method, instruction patterns are written in prefix order and interpreted as a context free grammar. Moreover, through the modified *LALR(1)* parser which is constructed by the above grammar, the cover can be found by parsing the input tree. Because the essence of the grammars is ambiguous, some heuristics and simplifications are offered to resolve the ambiguity. Using general tree pattern matching method to avoid ambi-

guities is guaranteed to select the cover of the minimum cost. For instance, the general methods which use a dynamic programming algorithm are used in *TWIG* [1].

Our experience with the tree-rewriting rules has shown that such a method is easy to use and the specification of the instruction patterns is independent from the actual tree pattern matching algorithm. However, at the early time of the architecture development, the architecture will be changed sometimes. Thus, a flexible compiler is necessary for the architect designing a new architecture. For the need of an architect, how to retarget a compiler to different machines is an important issue.

The goal of our research, *Arden* (Architecture development environment) compiler, is design a flexible compiler to help the architect retarget the compiler to a new instruction set. *Arden* compiler uses a simple and efficient tree pattern matching method to produce an efficient code generator. By traversing a template of tree-rewriting rule from the bottom up, it can be hashed to a *characteristic number* which can represent the rule. Then, the tree pattern matcher can generate the cover sets by comparing the *characteristic number* of a subtree in an *RTL* tree with those of tree-rewriting rules from the bottom up. In order to output efficient codes, the action phase will choose a cover set of a minimum cost instructions for each *RTL* tree. Thus, the tree pattern matching problem will become a simple number comparison. By changing the instruction descriptions, we can retarget the code generator to different instruction sets more easily.

In the next section, we illustrate the flowchart of *Arden* compiler. Instruction description is discussed in section III. Section IV describes a simple tree pattern matching algorithm. Section V will show the experimental results, and conclusions are given in the final section.

2 Flowchart of Arden Compiler

The compiler of Arden consists of the gcc front-end, the tree pattern matcher, the instruction description, and the action phase. The front-end of *gcc* takes *c* program as input, and outputs the intermediate code (*RTL* tree). A template of tree-rewriting rule in instruction description which is used to describe the target instructions can be hashed to a *characteristic number*, and this *characteristic number* is used to represent the tree-rewriting rule which includes this template. The cover sets of an *RTL* tree are generated by the tree pattern matcher, which compares the *characteristic number* of a subtree in an *RTL* tree with those of the tree-rewriting rules from the bottom up. The subtree, matched with a tree-rewriting rule, will be replaced by the corresponding replacement node; the replacing process will continue until the root of the *RTL* tree is encountered. The action phase outputs the assembly code which has a minimum cost among the cover sets for the target machine.

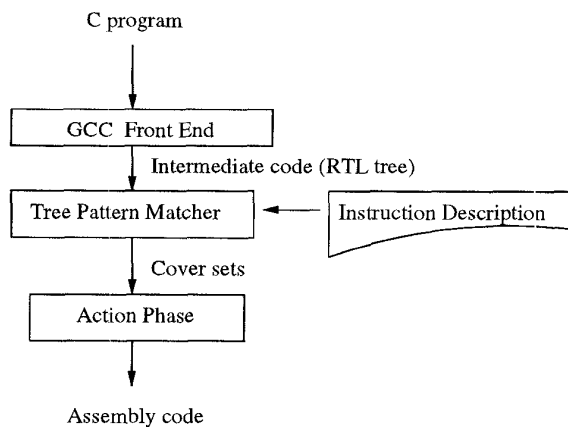


Figure 1: Flowchart of Arden compiler

3 Instruction Description

3.1 Instruction Description

In order to generate the machine assembly code, the instruction description of a target machine can be represented by the tree-rewriting rules which contain macro expression, a replacement node, a template, and sets of condition expressions, cost, and action. A tree template, composed of a replacement node and a template, represents a computation which is performed by one or more machine instructions. A set

of condition expression is used to select a proper action. After a template has been matched with a subtree in an *RTL* tree, the condition expression must be checked. The syntax of a tree-rewriting rule in an instruction description is described as the following:

```

%define_insn
  @ Macro expression @
  { Replacement ← Template }
  @ Condition expression1 @
  @ Cost1 @
  { Action1 }
  @ Condition expression2 @
  @ Cost2 @
  { Action2 }
  ⋮
  @ Condition expressionN @
  @ CostN @
  { ActionN }
%
  
```

The entry between two @s is optional.

- *Macro expression* defines the macro strings which will be expanded in template, condition expression, or action.
- *Replacement* is a replacement node, and *template* is the representation of an *RTL* tree.
- *Condition expression* will settle some constraints for the operands in the template and will be checked by the tree pattern matcher when the template is matched with a subtree in an *RTL* tree.
- *Cost* is the execution cycle time of the action code.
- *Action* returns the assembly code for the rule.

For example, *add* and *sub* instructions in *SPARC* architecture can be defined in one rule by the following macro expression.

```

%define_insn
  @ VAR macro_operator = {"plus","minus"}
  & macro_opcode = {"add","sub"} @
  { (r SI 0) ← (macro_operator:SI (r SI 1)(I SI 2 )) }
  @ %i2 < 4096 and %i2 ≥ -4096 @
  @cost=1@
  { macro_opcode %r1, %i2, %r0 }
  @ %i2 ≥ 4096 or %i2 < -4096@
  @cost=3@
  { "sethi hi(%i2), g1; or lo(%i2), g1, g1;
    macro_opcode %r1, g1, %r0;" }
%
  
```

The above *action* is just a piece of *SPARC* assembly

code. In this *macro expression*, *macro_operator* is either plus or minus, and *macro_opcode* is either add or sub. The applicability of the rule will be settled by a set of condition expression. The operands contain strings like *%rn* and *%in* where *n* is the order of the operands in the tree template. For example, a target register operand is represented by the string *%r0*, and the immediate value of the second operand by *%i2*. When the template is matched with a subtree in an *RTL* tree, the tree pattern matcher will check the second operand *%i2*. If the value of *%i2* is between -4096 and 4095, the code of *action1* will be outputted; otherwise, the code of *action2* will be output. The action of the tree-rewriting rule is outputted in the action phase and consists of statements which are assembly codes or assembler modules. For example, if *macro_operator* is replaced by *plus* in the template, the *macro_opcode* will be replaced by *add*. This rule indicates that the target register is equal to the result of the first source register plus the immediate value. For example, if the *RTL* tree is "reg 3 ← reg4 + 30", "(r SI 0)←(plus :SI (r SI 1)(I SI 2))" can be matched in the tree-rewriting rules, and thus the instruction "add r4, 30, r3" is outputted.

3.2 Tree-rewriting Rules

The front-end of *Arden* translates source programs into an intermediate representation (*RTL*). The *RTL* program is a series of expression trees which are then transformed into postfix order for the bottom-up comparison. Fig. 2 shows the *RTL* of an assignment *a:=b+1* in which both *a* and *b* are local variables; one is stored at offset 4, and the other at offset 8 for the stack pointer which is stored in register *sp*. The *mem* operator will return the content of a memory location. In translating an *RTL* tree, there are two steps:(1)

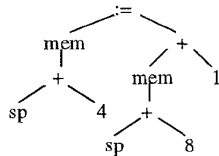


Figure 2: Intermediate representation of *a:=b+1*

traversing the tree in postfix order and (2) producing code for each individual node. Each nonterminal node in the template represents an intermediate result calculated and will be replaced by a replacement node in tree-rewriting rules. Fig. 3 shows the tree-rewriting rules needed to translate the *RTL* tree of Fig. 2. The

instruction *add* of *rule1* in Fig. 3 is to add the content of a memory location (addressed by the sum of stack pointer and an offset) and a register, and returns the result into a register. By repeatedly searching, a

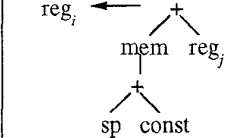
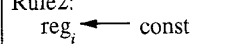
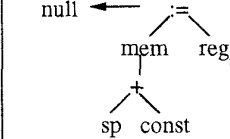
Rewrite rule	Cost	Action
Rule1: 	3	ld [sp+const], rt add rt, rj, ri
Rule2: 	1	mov const, ri
Rule3: 	2	st ri, [sp+const]

Figure 3: Tree-rewriting rules for *a:=b+1*

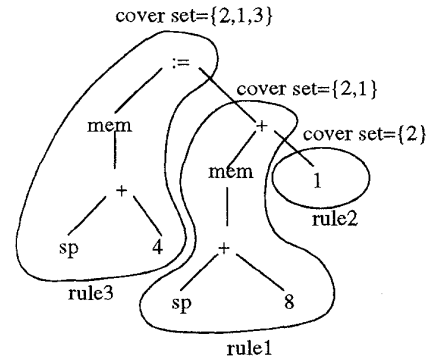


Figure 4: The cover tree for *a:=b+1*

subtree in an *RTL* tree can be reduced to a replacement node, and the *RTL* tree can be rewritten by the tree-rewriting rules. After tree pattern matching, the nodes of an *RTL* tree will be marked with cover sets which include all the possible matching combinations of the replacement rules, and the subtrees will be replaced by replacement nodes. The labeled tree is called a cover tree [5]. Fig. 4 shows how to transform an *RTL* tree into a cover tree.

4 A Simple Tree Pattern Matching Algorithm

The target assembly code is generated by tree pattern matching in which an *RTL* tree is reduced into a replacement node by repeatedly searching for the subtree in the *RTL* tree. The subtrees, matched with template, will be replaced by the corresponding replacement node. By using the hashing function from the bottom up, each node of the template can get its own *characteristic number*, then the tree-rewriting rule can be marked with the *characteristic number* of the root in the template. The cover sets of an *RTL* tree are generated by the tree pattern matcher which compares the *characteristic number* of a subtree in an *RTL* tree with those of the tree-rewriting rules from the bottom up. Then, the action phase will choose a minimum cost instructions for the output assembly code. The hashing function is defined as $F(\text{root}, \text{left}, \text{right}) = (\text{root} + \text{left} * \text{right}) \bmod \text{prime}$ - the *root* is the root of the tree, *left* is the left subtree of the root, *right* is the right subtree of the root, and the *prime* number is 17041. Hence, we will represent each operator and terminal node with a different number. Below is an example of operator/terminal node representation.

operator/terminal	different number
reg	203
mem	204
+	302
null subtree	1

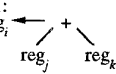
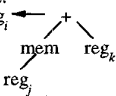
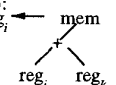
Rewrite rule	Cost	Action	Characteristic number
Rule1: 	1	add rj, rk, ri	7429
Rule2: 	3	ld m[rj], rt add rt, rk, ri	14759
Rule3: 	2	ld m[rj+rk], ri	7633

Figure 5: Tree-rewriting rules represent some instructions of a target machine

After applying the hashing function F into tree-rewriting rules, we can get the *characteristic number* for each tree-rewriting rule in Fig. 5. The processes of *characteristic number* calculation are depicted as the following:

The first rule: $F(+, \text{reg}_j, \text{reg}_k) = (302 + 203 * 203) \bmod 17041 = 7429$
 The second rule: $F(\text{mem}, \text{reg}_j, \text{null}) = (204 + 203 * 1) \bmod 17041 = 407$

$F(+, \text{mem}[\text{reg}_j], \text{reg}_k) = (302 + 407 * 203) \bmod 17041 = 14759$
 The third rule: $F(+, \text{reg}_j, \text{reg}_k) = (302 + 203 * 203) \bmod 17041 = 7429$
 $F(\text{mem}, \text{reg}_j + \text{reg}_k, \text{null}) = (204 + 7429 * 1) \bmod 17041 = 7633$

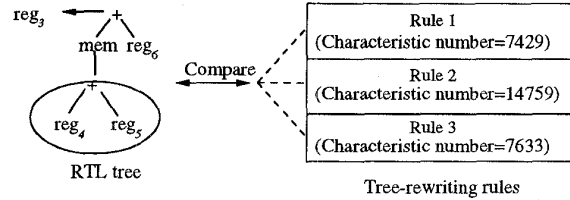


Figure 6: Simple tree pattern matching by the hashing function

The tree-rewriting rules in Fig. 5 represent some instructions of a target machine. Each rule is marked with a *characteristic number* computed by the hashing function. In tree pattern matching, the assembly instruction of the target machine will be generated by the action of a tree-rewriting rule. To illustrate, let us traverse the *RTL* tree by tree pattern matching. The process is shown in Fig. 6. After the *characteristic numbers* of the three rules have been generated, the tree pattern matcher will traverse the *RTL* tree from the bottom up. The tree pattern matcher will compute the *characteristic number* of a subtree in the *RTL* tree - $F(+, \text{reg}_4, \text{reg}_5) = 7429$. Then, Comparing the *characteristic number* 7429 with those of the tree-rewriting rules, we find that *rule*₁ matches with the subtree of the *RTL* tree. The template of the first rule in which $j=4$ and $k=5$ matches with the leftmost leaf of the *RTL* tree. If we use this rule, the subtree "reg₄ + reg₅" of the *RTL* tree will be replaced by reg₇, and the instruction "add r₄, r₅, r₇" will be generated. The second rule in which $i=3$, $j=7$, and $k=6$ matches with the root of the *RTL* tree. If we choose the second rule, the *RTL* tree will be replaced by a single node reg₃; then both instructions "ld m[r₇], r₈" and "add r₈, r₆, r₃" will be generated. The code which generated by the tree pattern matcher to translate the *RTL* tree are shown as the following:

```
add r4, r5, r7
ld m[r7], r8
add r8, r6, r3
```

The tree pattern matching algorithm includes two phases: 1. the preprocessing phase. 2. the tree pattern matching phase. The preprocessing phase parses the instruction description, and calculates the *characteristic number* for each tree-rewriting rule. For a specific target instruction set, the preprocessing phase only needs to be done once. As for the tree pattern

matching phase, an *RTL* tree can be parsed by consulting the *characteristic numbers* of the tree-rewriting rules to produce the cover sets.

4.1 Preprocessing Phase

The use of macro strings is for the reduction of tree-rewriting rules which are used to describe the target instructions. Then, in preprocessing phase, the macro strings will be expanded by the tree pattern matcher for each tree-rewriting rule. The tree pattern matcher first expands macro strings for each tree-rewriting rule. Next, the tree pattern matcher computes a *characteristic number* for each template, and then sorts the templates again according to their *characteristic numbers*.

4.2 Tree Pattern Matching Phase

There are two steps to traverse an *RTL* tree in the tree pattern matching phase. The first step is to traverse the *RTL* tree in postfix order. Then, each node in an *RTL* tree will get a *characteristic number*, and the tree pattern matcher will compare this *characteristic number* with those of the tree-rewriting rules. If the *characteristic number* of a subtree in an *RTL* tree is equal to that of a tree-rewriting rule and one of the condition expressions in this rule can be satisfied, the tree pattern matcher will record the information of match node in the match parsing stack. The match node is the root of the subtree in an *RTL* tree which can be replaced by a replacement node. The second step is to replace each match node by a replacement node in the match parsing stack until the root is encountered. After that, the tree pattern matcher outputs the cover sets. If there exist multiple cover sets, the action phase will choose a one of minimum cost. The cover sets which are generated from the traversal of an *RTL* tree are shown in Fig. 7.

For the match node “mem”, this subtree can be matched by the template of *rule*₃, then we can rewrite this subtree as a single replacement node *reg*₇. Next, “*reg*₇+*reg*₆” can be matched by the template of *rule*₁, and the cover set[1]={3, 1} will be output. As for the match node “+”, this subtree can be matched by the template of *rule*₁ and this subtree can be rewritten as a single replacement node *reg*₇. Then, the rest of the *RTL* tree can be matched by the template of *rule*₂. At last, the cover set[2]={1, 2} will be output. There are several different combinations of rules which are matched into an *RTL* tree. If several different cover sets are matched into the root of an *RTL* tree, the one of the minimum cost will be selected.

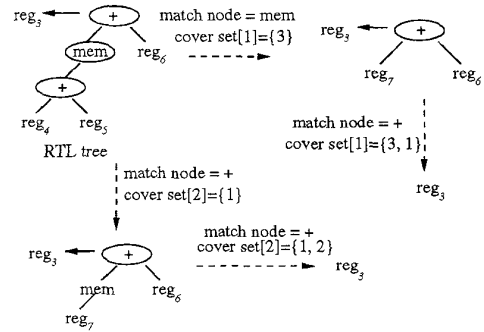


Figure 7: The cover sets of an *RTL* tree

5 Experimental Results

We have implemented two code generators for *DLX* and *SPARC*. The comparison of number of rules and size of instruction descriptions between *gcc* and *Arden* is shown in Table 1. Compared with *gcc*, *Arden* use fewer rules to describe a target architecture. In addition, the preprocessing phase takes 0.2 second for *DLX* and 0.6 second for *SPARC*. The tables generated by preprocessing phase occupy 29 KB for *DLX* and 65 KB for *SPARC*. The program size of the tree pattern matching phase is 87 KB. Table 3 summarizes the tree pattern matching time of the *SPEC* [13] benchmarks compiled by *gcc* and *Arden*. Compared with *gcc*, the average matching time can be reduced by 69%. In Table 4, only the compiler time is included in the three comparisons with the manufacturer’s C compiler (*cc*), *gcc*, and *Arden*. The compiler time in *Arden* is less than in *gcc* and *cc*. *Arden* runs 1.06 times faster than *gcc* on average. All the above measurements are carried out on a *SPARC* 10 workstation.

6 Conclusions

In the paper, we have presented a simple number comparison method for tree pattern matching to produce a code generator. Our experiment shows that this method can reduce the tree pattern matching time by 69%, and the instruction descriptions size of *gcc* is 3.92 times more than *Arden* on average. Moreover, this method can get an optimal instructions for an *RTL* tree. Because table generated through preprocessing phase is very small, the space which code generator needs is greatly reduced. In other words, the tree pattern matching time and the complexity of space get a significant reduction. Furthermore, if we want to retarget the code generator to different ma-

	gcc		Arden	
	DLX	SPARC	DLX	SPARC
rules	125	179	49	87
size (Kbytes)	50	68	9.8	22

Table 1: Instruction descriptions size for gcc and Arden

benchmark	Tree pattern matching time		gcc / Arden
	gcc (sec)	Arden (sec)	
008.espresso	9.12	5.75	1.59
022.li	2.68	1.53	1.75
023.eqntott	1.56	0.96	1.63
026.compress	0.56	0.31	1.81
072.sc	3.94	2.31	1.71
085.gcc	29.9	18.12	1.65

Table 2: Tree pattern matching time for C SPEC Benchmarks in seconds

chines, what we need is only to change the instruction descriptions. Such a characteristic can help the architect to design a new architecture more easily.

References

- [1] Aho, A. V., Ganapathi, M., and Tjiang, S. W. K., "Code Generation Using Tree Matching and Dynamic Programming," ACM Trans. Program Lang. Syst., Vol. 2, No. 4, Oct. 1989, pp. 491-561.
- [2] Cattell, R. G. G., "Automatic Derivation of Code Generators from Machine Descriptions," ACM Trans. Program Lang. Syst., Vol. 2, No. 2, Apr. 1980, pp. 173-190.
- [3] Chase, David R., "An Improvement to Bottom Up Tree Pattern Matching," Proceedings of the 14th Annual Symposium on Principles of Programming Languages, 1987, pp. 168-177.
- [4] Davison, J. W., and Fraser, C. W., "The Design and Application of a Retargetable Peephole Optimizer," ACM Trans. Program Lang. Syst., Vol. 2, No. 2, Apr. 1980, pp. 173-190.
- [5] Emmelmann, H. S., and Landwehr, F. W., "BEG - A Generator for Efficient Back Ends," Proceeding of ACM Conference on programming Language Design and Implementation, Vol. 24, No. 7, June 1989, pp. 227-237.
- [6] Fraser, Christopher W., and Hanson, David R., "A Code Generation Interface for ANSI C," Software-Practice and Experience, Vol. 21, No. 9, Sep. 1991, pp. 963-988.
- [7] Hennessy, J. L., and Patterson, D. A., "Computer Architecture: A Quantitative Approach," Morgan Kaufmann Publishers Inc., San Mateo, 1990.
- [8] Hoffman, C. W., and O'Donnell, M. J., "Pattern Matching in Trees," Journal of the ACM, Vol. 29, No. 1, January 1982, pp. 68-95.
- [9] Lai, Feipei, Tsaur, F., and Shang, R., "ARDEN - ARchitecture Development ENvironment," IEEE TENCON 92, Nov., 1992, pp. 181-185.
- [10] Proebsting, Todd A., "Simple and Efficient BURS Table Generation," Proceeding of ACM SIGPLAN'92 Conference on Programming Language Design and Implementation, June 1992, pp. 331-340.
- [11] Stallman, R. M., "Using and Porting GNU CC (for version 2.2)," Free Software Foundation, Inc., Cambridge, Massachusetts, U.S.A, May 1992.
- [12] Glanville, R. S., "A Machine Independent Algorithm for Code Generation and its Use in Retargetable Compilers," Ph.D. Thesis, University of California, Berkeley, 1978.
- [13] Standards Performance Evaluation Corp. SPEC Benchmark Suite Release 2.0, Jan. 1992.

benchmark	cc (sec)	gcc (sec)	Arden (sec)	gcc / Arden
008.espresso	76.4	69.8	66.4	1.05
022.li	18.0	16.4	15.2	1.08
023.eqntott	12.2	11.8	11.2	1.05
026.compress	4.0	4.0	3.7	1.08
072.sc	31.8	31.3	29.6	1.06
085.gcc	251.2	230.5	218.7	1.05

Table 3: Compiler time for C SPEC Benchmarks in seconds