GLOBAL OPTIMIZATIONS IN A PROLOG COMPILER FOR THE TOAM

NENG-FA ZHOU

▷ Backtracking in the WAM may be very expensive for some kinds of programs due to the WAM's simple scheme for indexing clauses. Several compilation algorithms have been proposed for constructing sophisticated indexing code. However, these algorithms have the drawback that they may generate exponential size code for some kinds of programs. This paper presents a Prolog compiler for the TOAM (matching Tree Oriented Abstract Machine). The compiler generates code for a program that is at most linear in the size of the program. It adopts a simple algorithm for detecting the exclusiveness among clauses and the determinacy of predicates. For those predicates that are detected to be determinate, the generated code does not create any choice point. The compiler also optimizes shallow backtracking in the sense that it treats a failure caused by a test as a jump. In addition, the compiler classifies cuts into commit, shallow, and deep cuts, and optimizes the code for commit and shallow cuts. Empirical results show that these optimizations can improve the performance of compiled code significantly. ◁

1. INTRODUCTION

Backtracking in the WAM [22] may be very expensive [18] for some kinds of programs due to the WAM's simple scheme for indexing clauses. In the WAM, the decision of which clauses should be tried is made solely on the basis of the type and sometimes the main functor of the first argument of a call. This may result in choice points being created unnecessarily and common instructions of the clauses in a predicate being reexecuted on backtracking.

The TOAM [23] is designed with the aim of eliminating the deficiency of the WAM mentioned above. It provides a group of new instructions for encoding matching trees and new data areas for interpreting matching trees. A matching

Address correspondence to Neng-Fa Zhou, Faculty of Computer Science and Systems Engineering, Kyushu Institute of Technology, 680-4 Kawazu, Iizuka, Fukuoka, Japan.

Received March 1991; accepted June 1992.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/93/$6.00

tree consists of a root, test nodes, and leaves. The root corresponds to the beginning of the definition of a predicate, test nodes represent conditions for clauses in the predicate to be applicable, and leaves correspond to the remaining code in the bodies of the clauses in the predicate. A call is evaluated by interpreting the nodes in the corresponding matching tree in preorder. When a test node fails, we jump to its neighboring node. When a leaf fails, we restore the computation state using the current choice point, and backtrack to the alternative node. Hence, the common nodes having been executed need not be reexecuted on backtracking.

Leaves in a matching tree are classified into cut leaves and noncut leaves. A leaf in a matching tree is called a *cut leaf* if, after it is reached, all the leaves to the right of it need not be visited for any call. A matching tree is said to be *determinate* if all its leaves are cut leaves. A choice point is created for a call after test nodes succeed and a noncut leaf is reached in the corresponding matching tree. The choice point is discarded when a cut leaf is visited or there is no node remaining to be visited in the matching tree.

The idea of transforming a predicate into a tree is an extension of the idea of attaching indexing code to a predicate [21]. Different definitions of trees with different names, e.g., selection tree [20], switching tree [9], and decision tree [11], have been proposed. In these trees, a node corresponds to a set of clauses, and an edge is a test which acts as a filter. The clauses in a child node are those in the clauses of the parent node that can pass the test along the edge from the parent to the child node. When a leaf contains one clause only, we know there is only one clause in the predicate satisfying the conditions along the path from the root to the leaf. When a leaf contains more than one clause, a try-block must be generated. As a clause may appear in more than one leaf in a tree, duplicate code for such a clause must be generated.

The definition of a matching tree is different from those of previous trees. A leaf in a matching tree corresponds to only one clause in the corresponding predicate. As will be shown later, very compact code can be generated for matching trees.

In this paper, we present the TOAM architecture and a Prolog compiler for the TOAM. The TOAM architecture described here is a renewed version of that originally presented in [23]. The compiler has the following main features: 1) it generates code for a program that is at most linear in the size of the program, 2) it detects the determinacy of predicates and generates code for those predicates detected to be determinate such that no choice point will be created, 3) for nondeterminate predicates, it detects the exclusiveness among the clauses in them and uses the results to generate efficient code, 4) it also optimizes shallow backtracking in the sense that it treats a failure caused by a test node as a jump to the neighboring node of the test node, and 5) it classifies cuts into commit, shallow, and deep cuts, and optimizes the code for commit and shallow cuts.

Familiarity with the WAM is assumed. The reader may refer to [1, 8, 13, 22] for a detailed description of the WAM. In Section 2, we describe some preliminary concepts. In Section 3, we present the TOAM architecture. In Section 4, we describe the compilation phases. In Section 5, we give examples illustrating the compilation phases. In Section 6, we show the experimental results. In Section 7, we discuss both the advantages and disadvantages of the compiler, and in Section 8, we compare our work with related work.

2. PRELIMINARIES

2.1. The WAM

The WAM has three storage areas: the *program code*, the *registers*, and three stacks: the *local stack*, the *heap*, and the *trail stack*.

The WAM has a group of specific registers for representing the machine state. These registers indicate the current program point (P), the continuation program point (CP), the current environment (E), the current choice point (B), the top of the trail stack (T), and the top of the heap (H). The WAM also has a group of registers for passing arguments and temporary variables.

The heap contains terms created during execution. The trail stack is used to unbind variables on backtracking. The local stack contains two kinds of records: *environments* and *choice points*. An environment is a record with the following fields:

Parent:	Parent environment
CP:	Continuation program point
$Y_1, \ldots, Y_m$:	Permanent variables

A choice point is a record with the following fields:

P:	Alternative program point
CP:	Continuation program point
E:	Current environment
B:	Current choice point
T:	Top of the trail stack
H:	Top of the heap
$r_1, \ldots, r_n$:	Argument registers

The WAM has an abstract processor that realizes an instruction set. The instructions can be classified into *get*, *put*, *unify*, *procedural*, *nondeterminism*, and *indexing* instructions. The get instructions correspond to the arguments at the head of a clause and are responsible for unifying the arguments with those of a given call. The put instructions correspond to the arguments of a call in the body of a clause and are responsible for loading the arguments into the appropriate registers. The unify instructions correspond to the arguments of a compound term. The procedural instructions (*call*, *execute*, *proceed*, *allocate*, *deallocate*, etc.) are responsible for control transfer and environment allocation associated with procedure calling. The nondeterminism instructions (*try*, *retry*, *trust*, etc.) are responsible for choice point allocation and state restoration. The indexing instructions are responsible for linking together the code for different clauses of a procedure.

When an instruction fails, the WAM simply branches to the location stored in the P field of the current choice point. That location will be the address of a *retry*(*retry_me_else*) or *trust* (*trust_me_else_fail*) instruction that is the beginning of the next clause to try.

2.2. Modes

The mode of a predicate p indicates how the arguments of any call to p are instantiated just before the call is evaluated. The mode of a predicate p of n

arguments is declared as

:-mode p($M_1, \ldots, M_n$),

where $M_i (i = 1, \ldots, n)$ is c (closed), f (free), nv (nonvariable), or d (don't-know). We assume that modes are declared for all predicates and all the modes are correct.

3. THE TOAM

The TOAM has a new register, called *PB*, which is used to store the parent choice point. It is set to hold the value of the *B* register just after a call to a predicate whose corresponding matching tree is nondeterminate is invoked. Another new register, called CSIZE, holds the number of argument registers stored in a choice point. A choice point is modified to contain the contents of these two new registers.

PB: Parent choice point
CSIZE: Number of argument registers

A choice point is created for the current call if a noncut leaf is visited and no choice point has been created for the current call; it is discarded when a cut leaf is visited or there is no node remaining to be visited in the matching tree. The *PB* register is used to determine whether or not a choice point has been created for the current call. If the value of the *PB* register is different from that of the *B* register, then we know that a choice point has been created for the current call; otherwise, if *PB* and *B* have the same contents, then we know that no choice point has been created for the current call.

The TOAM provides four groups of new instructions: *conditional jump*, *nondeterminism*, *get-argument*, and *hash* instructions. A conditional jump instruction corresponds to a test node in a matching tree. It jumps to the address specified in the instruction when the test fails. For example, the *jump_on_not_var r*, *L* instruction jumps to *L* if the value of *r* is not a variable.

The nondeterminism instructions are defined as follows:

```
root N:
    CSIZE ← N;
    PB ← B;
cut_leaf:
    if B ≠ PB then
        B ← B(B); %B(B) denotes the B field of the current choice point
noncut_leaf L:
    if B ≠ PB then
        P(B) ← L
    else
        create a choice point;
create L:
    create a choice point;
reset L:
    P(B) ← L;
discard:
    B ← B(B);
```

These instructions are responsible for creating and discarding choice points. The *root N* instruction is the start instruction in the code for a nondeterminate matching tree. It sets the register CSIZE to hold the value N and the *PB* register to hold the content of B. The *cut_leaf* instruction is the start instruction in the code for a cut leaf. It first determines whether or not a choice point has been created for the current call. If a choice point has been created, then it discards the choice point; otherwise, it does nothing. The *noncut_leaf L* instruction is the start instruction in the code for a noncut leaf. It first determines whether or not a choice point has been created for the current call. If a choice point has been created, it resets $P(B)$ with L; otherwise, it pushes a choice point on top of the local stack. *create L* and *reset L* are two specific instructions of the *noncut_leaf L* instruction and *discard* is a specific instruction of the *cut_leaf* instruction.

A get-argument instruction assigns a component of a structure or list into a variable or a register. For example, *get_t_arg* r_1, r_2, *1* assigns the first component of the structure in r_2 to r_1, and *get_t_car* r_1, r_2 assigns the head of the list in r_2 to r_1. A hash instruction jumps to a child node directly, which has the form

hash r, $((v_1, l_1), \ldots, (v_n, l_n), l_{n+1})$.

It moves control to l_i $(1 \leq l_i \leq n)$ when the value in r is equal to v_i; otherwise, it moves control to l_{n+1}.

The *fail* instruction in the TOAM is defined differently from that in the WAM. It restores the computation state using the current choice point[1] and moves control to the address stored in the P register.

4. COMPILATION PHASES

4.1. Program Specialization

This phase specializes every clause based on the mode of the head predicate into an equivalent clause in the following form:

```
H:-T, B.
```

where T is a sequence of in-line tests, i.e., calls to built-in predicates that do not bind any variable. For any call G, if there exists a unifier θ such that $(G = H\theta) \wedge T\theta$, then it preserves the original program semantics to reduce G into $B\theta$.

For a clause

```
H:-B.
```

we first separate the in-line tests in front of B from other calls in B and obtain

```
H:-T, Cut, B'.
```

where Cut is either "!" or *true*, depending on whether or not there is a cut following T in the original clause. We then move output unifications in the head to the right of T or Cut as follows. For each f-argument A (whose corresponding mode is f) in the head that is not a variable or a variable that occurs in the head more than once, we replace A with a new variable Y and insert $Y = A$ **after** Cut. For each d-argument A in the head that is not a variable or a variable that occurs

[1] Which is done by *retry* and *trust* instructions in the WAM.

in the head more than once, we replace A with a new variable Y and insert $Y = A$ **before** Cut. For each *nv*-argument in the head, if it is a variable, then we treat it as a *d*-argument; otherwise, we treat all of its components as *d*-arguments. Note that all the variables in T must appear in the head after the transformation.

Consider, for example, the max/3 predicate which computes the greater one of two given numbers.

```
:-mode max(c, c, f).
max(X, Y, X):-X ≥ Y.
max(X, Y, Y):-X < Y.
```

After specialization, the program is transformed to

```
max(X, Y, Z):-X ≥ Y, Z = X.
max(X, Y, Z):-X < Y, Z = Y.
```

where output unifications are put to the right of tests.

4.2. *Detection of Determinacy*

For a predicate with n specialized clauses $c_1, c_2, \ldots,$ and c_n,

c_1: H_1:-T_1, B_1.
c_2: H_2:-T_2, B_2.
...
c_n: H_n:-T_n, B_n.

we say that c_j is an *exclusive clause* of c_i if: 1) the heads of c_i and c_j are not unifiable, or 2) after T_i is satisfied, T_j cannot be satisfied simultaneously for any call, or 3) T_i is followed by a cut. A clause c_i is called a *cut clause* if for any j $(j > i)$, c_j is an exclusive clause of c_i; otherwise, the clause is called a *noncut clause*. The predicate p is said to be *determinate* if every clause in it is a cut clause. The predicate p is said to be *globally determinate* if it is determinate and all the predicates used in any B_i $(1 \le i \le n)$ are globally determinate.

The problem of determining whether or not a predicate is determinate at compilation time is, in general, unsolvable [7, 9, 16]. The problem lies in condition 2), since it is impossible, in general, to check whether or not two conditions can be simultaneously satisfied. However, there are many cases where the problem can be solved. For example, it is easy to detect that the test $X \ge Y$ is mutually exclusive with $X < Y$.

We denote a specialized clause as

```
⟨H, T⟩:-B.
```

where the left-hand side (LHS) indicates the conditions on a call for the clause to be applicable to the call.

4.3. *Classification and Transformation of Cuts*

Consider the following clause with a cut in its body,

```
⟨H, T⟩:-L, !, R.
```

where L and R are sequences of literals. If L is an empty sequence of literals, then the cut is called a *commit cut*. If all the predicates in L are globally determinate, then the cut is called a *shallow cut*; otherwise, the cut is called a *deep cut*.

A commit cut in a clause just tells a compiler that the clauses below in the same predicate are exclusive clauses of the clause. What a compiler needs to do is just to save this information and use it when generating code for the clause. No extra code needs to be generated for a commit cut. A shallow cut needs only to detect whether or not a choice point has been created for the call unified with the head, and discards it if one is created. A shallow cut in a noncut clause discards the current choice point, and is thus replaced by a *discard* instruction. A shallow cut in a cut clause does nothing, and is thus removed from the clause. A deep cut is implemented with two internal predicates, *savecp/1* and *cutto/1*, as in the SB-Prolog system [7].

Note that if there is more than one shallow cut in a noncut clause, then these cuts are not treated in the same way. The first shallow cut is replaced by a *discard* instruction, and the remaining shallow cuts are removed.

4.4. Flattening Terms

In this phase, we flatten the terms on the LHS of a specialized clause. For the LHS, $\langle H, T\rangle$, of a clause, we first process those variables that occur more than once in H. For each such a variable X, we replace one of its occurrences with a new variable Y and insert $Y = X$ in front of T. We repeat this step until all variables in H are singletons.

We then traverse the arguments in H from right to left and flatten them as follows. When we encounter a compound term t, we first obtain a new term t' by flattening all its arguments from right to left, and then replace t' with a new variable Y and insert $Y = t'$ in front of T. When we reach a constant c, we simply replace it with a new variable Y, and insert $Y = c$ in front of T; otherwise, we do nothing. For example, the LHS

$$\langle p(X, [f(Y, Z)|L]), X \geq Y\rangle$$

is flattened to

$$\langle p(X, W_0), W_0 = [W_1|L], W_1 = f(Y, Z), X \geq Y\rangle.$$

This phase is very simple. We can improve it by using those heuristics adopted in WAM compilers [10, 19].

4.5. Variable Localization

This phase changes those permanent variables that occur as arguments of tests in the LHS of a clause into temporary variables. For each permanent variable X that occurs as an argument of a test, we simply replace all occurrences of X on the right-hand side (RHS) of the clause with a new variable Y and insert a new call $X = Y$ in front of the RHS. This phase makes it possible to delay the allocation of an environment until the LHS succeeds.

4.6. Register Allocation

This phase allocates registers for those variables that occur as arguments of tests in the LHS. The register allocation algorithm for the WAM [5] is used for allocating registers to other temporary variables.

Let N be the arity of the head in the LHS, and let Ready be the next register to be allocated. Ready gets $N+1$ as its initial value. This phase scans the variables that occur as arguments of tests on the LHS from left to right and allocates registers to them. When scanning a variable whose first occurrence appears as the ith argument of the head, we allocate register i to the variable. When scanning a variable that occurs as a component of a list or structure, we allocate register Ready to the variable and increment Ready by one.

For example, for the LHS

$$\langle p(X, W_0), W_0 = [W_1|L]), W_1 = f(Y, Z), X \geq Y \rangle.$$

X, W_0, W_1, and Y are variables that occur as arguments of tests. They are allocated registers 1, 2, 3, and 4, respectively.

4.7. Matching Tree Construction

This phase constructs a matching tree for every predicate. For each clause in a predicate, we encode the LHS into a list of test nodes, each of which corresponds to a test on the LHS. The order of the test nodes in the list is the same as that of the tests on the LHS. We associate each test node with a sequence of get-argument instructions that get values for those registers used in the test node.

A matching tree for a predicate is constructed by merging the corresponding node lists of the clauses in the predicate. Initially, the matching tree consists of only a root node. For every node list, we set the root node to be the parent node to which the next node is to be attached, and attach the nodes from left to right as follows. When attaching a test node t with the get-argument instruction sequence g to a parent node p, if p has no child, then we add t as a child of p and associate the edge between p and t with g; otherwise, if p has children, suppose its rightmost child is t' and the edge from p to t' is associated with g'; then we first check whether or not t is equal to t' and g is equal to g'. If it is true, we then merge these two nodes, set t' to be the new parent node, and continue to attach remaining nodes; otherwise, if it is false, we then add t to the right of t', connect p and t, associate the edge with g, set t to be the new parent node, and continue to attach remaining nodes. After all the nodes in a list are processed, we attach a leaf that is a sequence of instructions encoded from the RHS to the parent node. For example, Figure 1 shows a matching tree constructed for the following predicate.

```
:-mode exclusive(nv, nv).
exclusive(X=:=Y, W=\=Z):-
    X==W,
    number(Y),
    number(Z),
    Y=:=Z.
exclusive(X=:=Y, W=\=Z):-
    X==W,
    Y==Z.
```

We assume that leaves in a matching tree are numbered 1, 2, and so on from left to right. A leaf l is called an *exclusive leaf* of another leaf l' if the corresponding clause of l is an exclusive clause of the corresponding clause of l'. A leaf is said to be a *cut leaf* if its corresponding clause is a cut clause. A matching tree for a predicate is said to be *determinate* if the predicate is determinate.

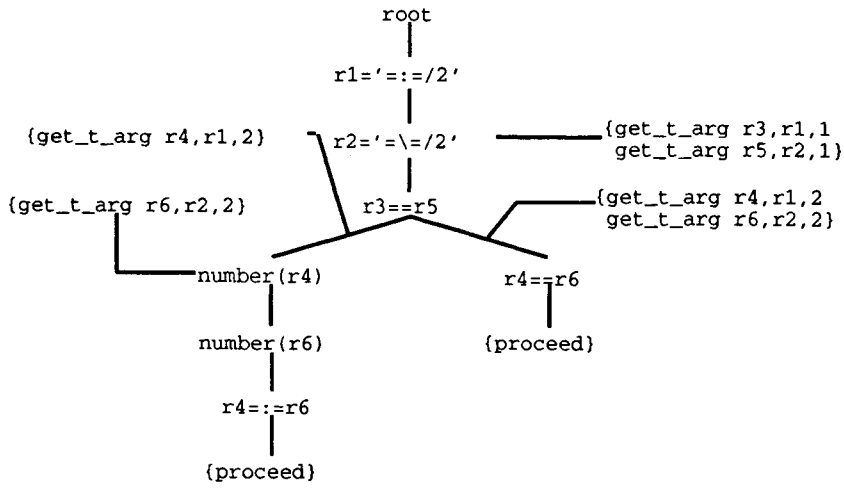

FIGURE 1. The matching tree for exclusive/2.

A leaf in a matching tree has two fields: *exclusive_leaves* and *cut_leaf*. The field *exclusive_leaves* indicates the set of exclusive leaves of the leaf. The field *cut_leaf* indicates whether or not the leaf is a cut leaf.

A test node has four fields: *children*, *gets*, *leaves*, and *exclusive_leaves*. For a test node t with n children $c_1, c_2, \ldots, c_n$, these fields have the following meaning. The field *children* links the test node with its children. If all the links to the children of the test node have the same sequences of get-argument instructions, then these sequences of instructions are moved up and stored in the *gets* field of the test node. For example, in Figure 1, the *gets* field of the node $r2 = ' = \backslash = /2'$ contains two instructions, while the *gets* field of the node $r3 = = r5$ is empty since the two edges going to its children have different values. The field *leaves* indicates the set of leaves in the subtree below the test node. The field *exclusive_leaves* indicates the set of exclusive leaves of the test node, which is defined to be

$$\text{exclusive\_leaves}(t) = \bigcap_{i=1}^{n} \text{exclusive\_leaves}(c_i).$$

The root of a matching tree has two fields: *children* and *gets*, which have the same meaning as above.

4.8. Code Generation

This phase traverses a matching tree in preorder, and generates executable code for the matching tree. We define two terms before describing how to generate code for a matching tree.

The *neighboring node* of a test node is the node to visit after the test node fails. It is the sibling node to the right of the test node if there is such a sibling; otherwise, it is the neighboring node of the parent of the test node. The neighboring node of the root of a determinate matching tree is

```
F: fail
```

and the neighboring node of the root of a nondeterminate matching tree is

```
CF: cut_leaf
    fail
```

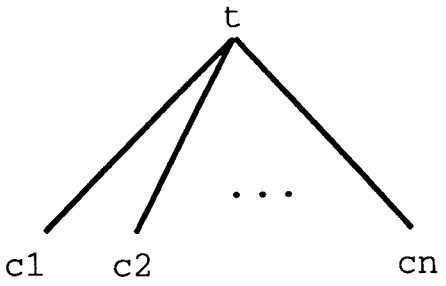

FIGURE 2. A subtree.

The *alternative node* of a tree is the node to visit when all the paths in the tree fails. The alternative node of a determinate matching tree is F, and the alternative node of a nondeterminate matching tree is CF. For a subtree whose root is t and t has n children $c_1, c_2, \ldots, c_n$ (see Figure 2), suppose the alternative node of the subtree is *altnode*; then the alternative nodes of the subtrees below c_i's are given as follows. If for any two children c_i and c_j $(i < j)$,

$$\text{leaves}(c_j) \subseteq \text{exclusive\_leaves}(c_i)$$

or the node t has only one child, then the alternative node of the subtree below any c_i is *altnode*. Otherwise, we classify the n children into k clusters G_1, $G_2, \ldots, G_k$, and transform the original subtree into that shown in Figure 3, such that for any two nodes c_i and c_j $(j > i)$ in the same cluster,

$$\text{leaves}(c_j) \subseteq \text{exclusive\_leaves}(c_i).$$

Any cluster is made to contain as many children satisfying the above condition as possible. The alternative node of the subtree below $G_i (i \leq k-1)$ is G_{i+1}, and the alternative node of the subtree below G_k is *altnode*.

Consider, for example, the following predicate,

```
:-mode p(c, c).
p(a, -).
p(b, -).
p(-, c).
p(-, d).
p(A, B):-s(A, B)
```

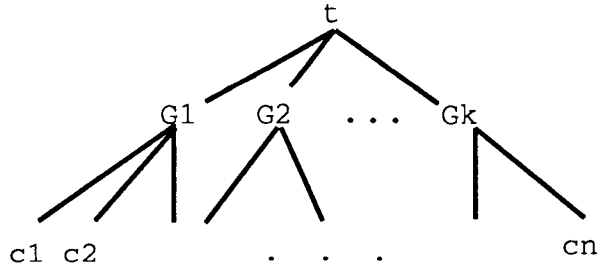

FIGURE 3. A transformed subtree.

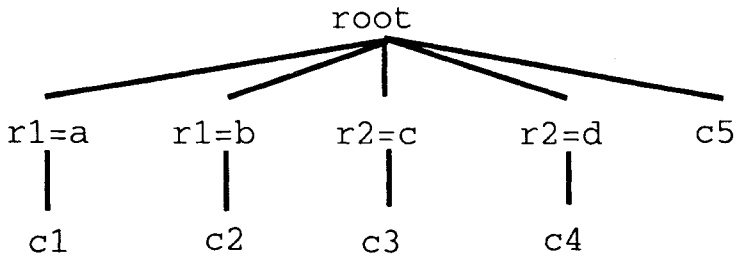

FIGURE 4. The original matching tree for $p/2$.

The original and the transformed matching trees for the predicate are shown in Figures 4 and 5, respectively. The alternative node of C_1 is G_2 and that of C_3 is C_5.

The code for the root of a matching tree is as follows. If the matching tree is nondeterminate, then we generate a *root CSIZE* instruction for the root, where CSIZE is the largest register number shared by more than one path in the matching tree, and output the instructions in the *gets* field of the root. Otherwise, we just output the instructions in the *gets* field of the root.

The code for a test node is a conditional jump instruction followed by the instructions in the *gets* field of the test node. The conditional jump instruction jumps to the neighboring node of the test node after the test fails. For a subtree whose root is t and t has n children $c_1, c_2, \ldots, c_n$ (see Figure 2), if all of these children are tests on the same register, then a hash instruction is generated.

The code for a noncut leaf is a *noncut_leaf L* instruction followed by instructions at the leaf, where L is the address of the alternative node of the leaf. The code for a cut leaf of a matching tree is as follows: if the matching tree is determinate or the matching tree is nondeterminate but we have not yet generated a *noncut_leaf L* instruction for the matching tree, then we just output the instructions at the leaf; otherwise, we generate a *cut_leaf* instruction and output the instructions at the leaf.

For example, the following shows the generated code for the matching tree shown in Figure 1.

```
exclusive/2 :
    root 5
    jump_on_not_eqrs r1, =:= /2, CF
    jump_on_not_eqrs r2, = \= /2, CF
    get_t_arg r3, r1, 1
    get_t_arg r5, r2, 1
    jump_on_not_equalrr r3, r5, CF
    get_t_arg r4, r1, 2
    jump_on_not_number r4, L
    get_t_arg r6, r2, 2
    jump_on_not_number r6, L
    jump_on_not_eqlrr r4, r6, L
    noncut_leaf L
    proceed
```

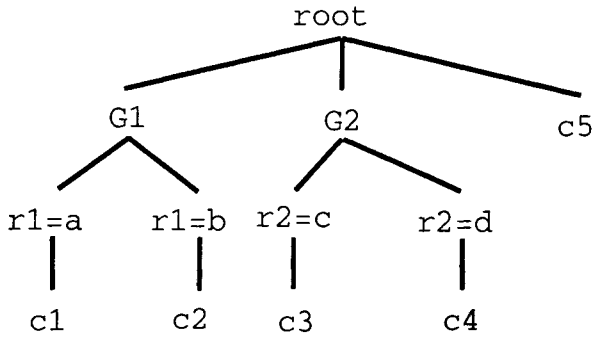

FIGURE 5. The transformed matching tree for $p/2$.

```
L :
    get_t_arg r4, r1, 2
    get_t_arg r6,r2,2
    jump_on_not_equalrr r4, r6, CF
    cut_leaf
    proceed
```

The root 5 instruction at the beginning indicates that the contents of five registers from the first to the fifth are to be stored in a choice point. In fact, only $r1$ and $r2$ need to be saved because $r4$ and $r6$ used in the second leaf are computed from $r1$ and $r2$, respectively. Therefore, the *root 5* instruction can be safely replaced by *root 2*. The code can be improved further in many other aspects. The first three labels of CF can be replaced by F since no *noncut_leaf* instruction appears before them. The *noncut_leaf L* instruction can be replaced by a *create L* instruction. In addition, the *get_t_arg r4, r1, 2* instruction below the label *L* can be moved up and merged with another *get_t_arg r4, r1, 2* instruction above.

5. EXAMPLES

In this section, we give examples illustrating the compilation phases described in the previous section.

5.1. Example 1: Compiling a Determinate Predicate

Consider the following predicate which checks whether or not an element is a member in a list,

```
:-mode member(c, c).
member(X, [X|_]):-!.
member(X, [_|T]):-member(X, T).
```

The cut in the first clause is classified as a commit cut, and the predicate is detected to be determinate. Based on this information, the compiler generates the following code for the predicate.

```
member/2 :
    jump_on_not_list r2, F
    get_t_car r3, r2
    jump_on_not_eqrr r1, r3, L
    proceed
```

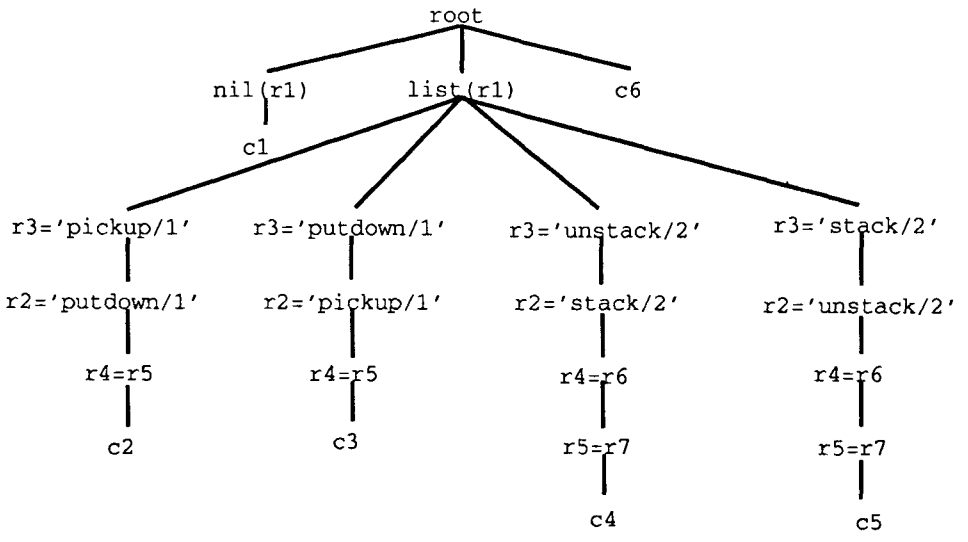

FIGURE 6. The matching tree for edge_consistent/2.

```
L :
    get_t_cdr r2, r2
    execute member/2
```

The code is much more efficient than that for the WAM due to that, among other things, 1) no choice point will be created, 2) the *jump_on_not_list r2, F* instruction will be executed only once for a call, and 3) the code does not contain instructions for the anonymous variables.

5.2. Example 2: Compiling Another Determinate Predicate

The following predicate is a part of a program for solving the blocks world problem. It checks whether or not a selected operator is consistent with the previous one.

```
:-mode edge_consistent(c, c).
edge_consistent([], _):-!.
edge_consistent([pickup(X)|_], putdown(X)):-!, fail.
edge_consistent([putdown(X)|_], pickup(X)):-!, fail.
edge_consistent([unstack(X, Y)|_], stack(X, Y)):-!, fail.
edge_consistent([stack(X, Y)|_], unstack(X, Y)):-!, fail.
edge_consistent(_, _).
```

The matching tree constructed for this predicate is shown in Figure 6. For the sake of simplicity, get-argument instructions are not shown in the figure. From the matching tree, the compiler generates the following code:

```
edge_consistent/2 :
    jump_on_not_nil r1, L1
    proceed
```

```
L1 :
        jump_on_not_list r1, L6
        get_t_car r3, r1
        hash r3,
             ((pickup/1, L2),
             (putdown/1, L3),
             (unstack/2, L4),
             (stack/2, L5),
             L6)
L2 :
        jump_on_not_eqrs r3, pickup/1, L6
        jump_on_not_eqrs r2, putdown/1, L6
        get_t_arg r4, r3, 1
        get_t_arg r5, r2, 1
        jump_on_not_eqrr r4, r5, L6
        fail
L3 :
                  ⋮
L4 :
        jump_on_not_eqrs r3, unstack/2, L6
        jump_on_not_eqrs r2, stack/2, L6
        get_t_arg r4, r3, 1
        get_t_arg r6, r2, 1
        jump_on_not_eqrr r4, r6, L6
        get_t_arg r5, r3, 2
        get_t_arg r7, r2, 2
        jump_on_not_eqrr r5, r7, r6
        fail
L5 :
                  ⋮
L6 :
        proceed
```

The hash instruction moves control to one of the four children of *list*($r1$) directly, or to $L6$ if no one of the children is reachable. Note that the jumping address for any test node below *list*($r1$) is $L6$, the address of the neighboring node of *list*($r1$).

5.3. *Example 3: Compiling a Nondeterminate Predicate*

Consider the following nondeterminate predicate,

```
:-mode t(c, c)
t(_, [0, 1]).
t(_, [0, 0]).
t(D, T):-D > 0, s(D, T).
```

The predicate is nondeterminate since the third clause is not an exclusive clause. The matching tree for this predicate is shown in Figure 7. The code for the

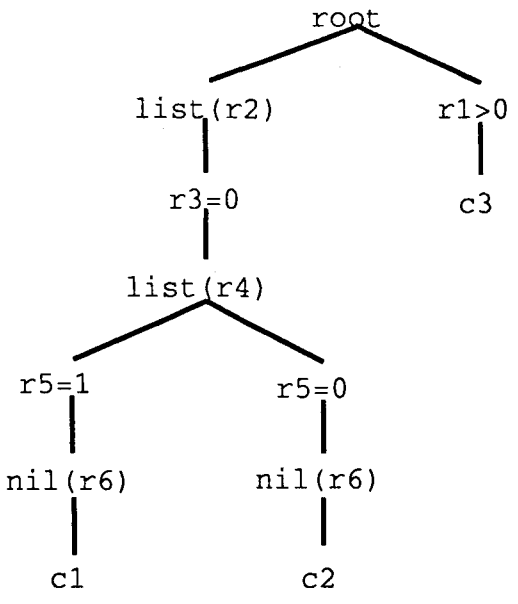

FIGURE 7. The matching tree for $t/2$.

matching tree is as follows:

```
t/2 :
        root 5
        jump_on_not_list r2, L3
        get_t_car r3, r2
        jump_on_not_eqlrn r3, 0, L3
        get_t_cdr r4, r2
        jump_on_not_list r4, L3
        get_t_car r5, r4
        hash r5, ((1, L1), (0, L2), L3)
L1 :
        jump_on_not_eqlrn r5, 1, L3
        get_t_cdr r6, r4
        jump_on_not_nil r6, L3
        noncut_leaf L3
        proceed
L2 :
        jump_on_not_eqlrn r5, 0, L3
        get_t_cdr r6, r4
        jump_on_not_nil r6, L3
        noncut_leaf L3
        proceed
L3 :
        jump_on_not_gtrn r1, 0, CF
        cut_leaf
        execute s/2
```

Note that the address of the alternative node of $C1$ is $L3$, not $L2$.

6. PERFORMANCE EVALUATION

We have implemented the compiler described above by modifying the SB-Prolog compiler [6]. In this section, we compare the efficiency of the code for the TOAM with that of the code for the WAM in terms of code size, number of choice point operations, and execution time. The code for the WAM is generated by the SB-Prolog compiler. The following seven representative benchmark programs are tested.

queens: finding all solutions for the 8-queens problem
nrev: reversing a list of size 30
qsort: sorting a list of size 50
diff: differentiating four expressions
mutest: a program containing multiple functional predicates
mutest1: a specialized program of mutest.
The multiple functional predicates are specialized.
BOYER: proving a theorem in propositional logic

Table 1 shows the byte code size. The size of the code for the TOAM is smaller than that for the WAM except for the code of *diff*.

Table 2 shows the number of choice point operations, i.e., the sum of the number of choice point creations and machine state restorations performed during execution.

Table 3 shows the execution time. Our system improved the time performance for all programs except for *nrev* and *mutest*. The slow-down for *nrev* is mainly due to the fact that get-argument instructions have to specify which component of which term are to be fetched. The slow-down for *mutest* is because the WAM compiler generates indexing code for the first argument, even though its mode is *d*, and the indexing code does reduce the number of clauses to be tried when the first argument of a call happens to be instantiated. By comparing the results for *mutest* and *mutest1*, we know that it is necessary to specialize multiple functional predicates before transforming them into matching trees.

7. DISCUSSION

The execution speed of compiled code depends on many factors, among which the number of instructions executed and the times of memory accesses made are the most important ones. In this section, we discuss the advantages and disadvantages of our compiler in terms of these two factors.

TABLE 1. Code size (byte)

Program	WAM	TOAM	TOAM/WAM
queens	1384	759	0.55
nrev	1000	816	0.82
qsort	1310	1228	0.93
diff	1985	2048	1.03
mutest	1486	1127	0.76
mutest1	2095	1544	0.74
BOYER	15669	14054	0.90

TABLE 2. Choice point operations

Program	WAM	TOAM	TOAM/WAM
queens	79482	22696	0.29
nrev	0	0	—
qsort	1745	0	0
diff	460	0	0
mutest	7985	12865	1.61
mutest1	7985	3805	0.48
BOYER	779542	75832	0.10

7.1. Advantages

1) *Optimization of determinate predicates.* The code for those predicates detected to be determinate does not create any choice points.
2) *Pruning exclusive clauses.* The definition of a neighboring node is different from that of an alternative node. When a leaf is visited, those nodes lying between the leaf and its alternative node will be pruned.
3) *Delay of the creation of choice points and the allocation of environments.* The creation of a choice point for a nondeterminate predicate is delayed until tests succeed and a noncut leaf is visited. The localization of permanent variables in tests makes it possible to allocate an environment after tests succeed.
4) *Avoidance of the reexecution of common instructions.* Compilers for the WAM compile clauses in a predicate separately. However, our compiler compiles a predicate as a whole into a matching tree. Once instructions shared by several paths in a matching tree are executed, they need not be reexecuted when backtracking occurs.
5) *Specialization of unification to matching.* The head H of a specialized clause is compiled into specific and efficient code as any call G will b matched against the head ($G = H\theta$) instead of being unified with the head ($G\theta = H\theta$). It is especially the case when the head contains many anonymous variables.
6) *Reordering of get-argument instructions.* In the code for the WAM, a *get_structure* or *get_list* instruction is followed by unify instructions that correspond to the arguments of a structure or list. A get-argument instruction can be considered as a unify instruction in read mode. However, get-argument instructions are placed back if possible in the code for a clause. Therefore, the get-argument instructions following test instructions need not be executed if these tests fail.

TABLE 3. Execution time (milliseconds, SUN-3/80)

Program	WAM	TOAM	TOAM/WAM
queens	8800	4260	0.48
nrev	28	32	1.14
qsort	70	34	0.49
diff	12	8	0.67
mutest	170	228	1.34
mutest1	170	148	0.87
BOYER	120120	68490	0.57

7) *Optimization of cuts.* By classifying cuts into commit, shallow, and deep ones, the compiler is able to delete all commit and some shallow cuts, and replace other shallow cuts with *discard* instructions. The investigation of real Prolog programs shows that large portions of cuts are commit or shallow ones, and the optimization of them is nontrivial.

7.2. Disadvantages

1) *Manipulations of two more fields in a choice point.*[2] The operation of the CSIZE and PB fields in a choice point affects the execution speed. Suppose the average number of argument registers stored in a choice point is 4; then it takes more than about 1/6 time to create a TOAM choice point than to create a WAM choice point.
2) *Comparison of the contents of B and PB.* The *noncut_leaf L* and *cut_leaf* instructions need to check whether or not a choice point has been created for the current call by comparing the contents of the B register with that of the PB register.
3) *Fetch of the operands of get-argument instructions.* There are operands in a get-argument instruction specifying which component of which term is to be fetched. On the contrary, in the WAM, the S register points to the next component to be unified.

8. RELATED WORK

There are many approaches for eliminating undesirable operations of choice points. Mellish [15] argued that large portions of Prolog programs are actually determinate. He proposed a technique for detecting the determinacy of predicates, and optimizing the code of determinate predicates for the POPLOG VM machine. Debray and Warren [7] considered the problem of inferring functionality, a more general definition of determinacy, of predicates and literals, and optimizing functional predicates and literals by inserting cuts into programs automatically. However, the code for a predicate may create choice points, even though it is detected to be determinate.

Butler, Loganatharaj, and Olson [2] considered the transformation of mutually exclusive clauses into *if_then_else*. They also proposed to generate efficient code for *if_then_else* in the case when the condition is an in-line test. However, two clauses can be transformed into *if_then_else* only if their heads subsume each other and they are mutually exclusive. Even some determinate predicates cannot be transformed.

Van Roy, Demoen, and Willems [20] proposed an approach for constructing sophisticated indexing code for predicates. In this approach, the compiled code is comprised of selection code, try-blocks, and clause code. The selection code tests the main functors of the arguments of a call and moves control to the appropriate try-block. Hickey and Mudambi [9] improved this approach. They designed a group

[2] One of the referees pointed out that instead of saving PB, we can get its correct value by resetting PB as PB = $B(B)$ each time a failure occurs. This method is the same as that with the B0 register of the WAM described in [1, p. 76].

of switching instructions with which unification instructions are allowed to be combined in the selection code. However, duplicate code may be generated for a clause. Recently, Kliger and Shapiro improved their decision tree compilation algorithm for FCP, a concurrent programming language that does not support deep backtracking, and proposed a decision graph compilation algorithm [12]. Their new algorithm also generates linear size code for all programs.

Sterling and Shapiro [17] introduced a distinction between shallow and deep backtracking. Carlsson [4] proposed an approach for optimizing shallow backtracking. His approach treats shallow and deep backtracking differently in the sense that it creates partial choice points for shallow backtracking and creates full choice points for deep backtracking. However, the creation and use of partial choice points are frequent and expensive.

Recall the $t/2$ predicate given in Section 5. Although the first two clauses are mutually exclusive, the predicate is neither determinate nor functional. This predicate cannot be transformed into *if_then_else*. The system by Hickey and Mudambi generates two try-blocks for this predicate. Carlsson's approach does not take the mutual exclusion into account, and therefore cannot generate efficient code for this predicate.

The cut operator is also an object that should be optimized. The SICStus-Prolog system distinguishes cuts that are not preceded by a *call* instruction from others [3]. Ait-Kaci [1] considered the classification of cuts into shallow and deep ones. By his definition, a shallow cut is one located in front of the body of a clause, and a deep cut is any other one. His scheme of optimizing shallow cuts uses a new register B0, which is similar to the PB register used here and the CUTB register used in [14], and requires the *call* and *execute* instructions to set B0 to hold the content of B.

This work was conducted while the author visited Kyushu University. Thanks to Professor Kazuo Ushijima, his supervisor there, and Associate Professor Toshihisa Takagi for their sincere encouragement and valuable discussion. The referees' comments contributed greatly to the improvement of this paper.

REFERENCES

1. Ait-Kaci, H., *Warren's Abstract Machine*, MIT Press, 1991.
2. Butler, R. M., Loganatharaj, R., and Olson, R., Notes on Prolog Program Transformations, Prolog Style, and Efficient Compilation to the Warren Abstract Machine, Tech. Rep., Argonne National Laboratory, 1988.
3. Carlsson, M., Freeze, Indexing, and other Implementation Issues in the WAM, in: *Proc. of the 4th International Conference on Logic Programming*, 1987, pp. 40–58.
4. Carlsson, M., On the Efficiency of Optimizing Shallow Backtracking in Compiled Prolog, in: *Proc. of the 6th International Conference on Logic Programming*, 1989, pp. 3–16.
5. Debray, S. K., Register Allocation in a Prolog Machine, in: *Proc. of 1986 IEEE Symposium on Logic Programming*, 1986, pp. 267–275.
6. Debray, S. K. (ed.), The SB-Prolog System Version 3.0, A User Manual, Dept. of Computer Science, University of Arizona, Tucson, 1988.
7. Debray, S. K. and Warren, D. S., Functional Computation in Logic Programs, *ACM Trans. on Programming Languages and Systems*, 11:451–481 (1989).

8. Gabriel, J., Lindholm, T., Lusk, E. L., and Overbeek, R. A., A Tutorial on the Warren Abstract Machine for Computational Logic, ANL-84-84, 1984.
9. Hickey, T. and Mudambi, S., Global Compilation of Prolog, *J. Logic Programming*, 7:193–230 (1989).
10. Janssens, G., Demoen, B., and Marien, A., Improving the Register Allocation in WAM by Reordering Unification, in: *Proc. of the 5th International Conference and Symposium on Logic Programming*, 1988, pp. 1388–1402.
11. Kliger, S. and Shapiro, E., A Decision Tree Compilation Algorithm for FCP(|,:,?), in: *Proc. of 5th International Conference and Symposium on Logic Programming*, 1988, pp. 1315–1336.
12. Kliger, S. and Shapiro, E., From Decision Trees to Decision Graphs, in: *Proc. of the North American Conference on Logic Programming*, 1990, pp. 97–110.
13. Maier, D. and Warren, D. S., Computing with Logic: Logic Programming with Prolog, Benjamin/Cummings Publishing, 1988.
14. Marien, A. and Demoen, B., On the Management of Choice Point and Environment Frames in the WAM, in: *Proc. of the North American Conference on Logic Programming*, 1989, pp. 1036–1047.
15. Mellish, C. S., Some Global Optimizations for a Prolog Compiler, *J. Logic Programming*, 2:43–66 (1985).
16. Sawamura, H., and Takeshima, T., Recursive Unsolvability of Determinacy, Solvable Cases of Determinacy and their Applications to Prolog Optimization, in: *Proc. of 1985 IEEE Symposium on Logic Programming*, 1985, pp. 200–207.
17. Sterling, L. and Shapiro, E., *The Art of Prolog*, MIT Press, 1986.
18. Touati, H. and Despain, A., An Empirical Study of the Warren Abstract Machine, in: *Proc. of 1987 IEEE Symposium on Logic Programming*, 1987, pp. 114–124.
19. Umrigar, Z., Finding Advantageous Orders for Argument Unification for the Prolog WAM, in: *Proc. of the North American Conference on Logic Programming*, 1990, pp. 79–95.
20. Van Roy, P., Demoen, B., and Willems, Y., Improving the Execution Speed of Compiled Prolog with Modes, Clause Selection, and Determinism, LNCS-250, pp. 111–125, 1987.
21. Warren, D. H. D., Implementing Prolog-Compiling Predicate Logic Programs, Tech. Rep. 39 and 40, Dept. of Artificial Intelligence, Univ. of Edinburgh, Scotland, 1977.
22. Warren, D. H. D., An Abstract Prolog Instruction Set, SRI International No. 309, 1983.
23. Zhou, N. F., Takagi, T., and Ushijima, K., A Matching Tree Oriented Abstract Machine for Prolog, in: *Proc. of the 7th International Conference on Logic Programming*, 1990, pp. 159–173.