Parameter Passing and Control Stack Management in Prolog Implementation Revisited

NENG-FA ZHOU
Kyushu Institute of Technology

Parameter passing and control stack management are two of the crucial issues in Prolog implementation. In the Warren Abstract Machine (WAM), the most widely used abstract machine for Prolog implementation, arguments are passed through argument registers, and the information associated with procedure calls is stored in possibly two frames. Although accessing registers is faster than accessing memory, this scheme requires the argument registers to be saved and restored for backtracking and makes it difficult to implement full tail recursion elimination. These disadvantages may far outweigh the advantage in emulator-based implementations because registers are actually simulated by using memory. In this article, we reconsider the two crucial issues and describe a new abstract machine called ATOAM (yet Another Tree-Oriented Abstract Machine). The ATOAM differs from the WAM mainly in that (1) arguments are passed directly into stack frames, (2) only one frame is used for each procedure call, and (3) procedures are translated into matching trees if possible, and clauses in each procedure are indexed on all input arguments. The above-mentioned inefficiencies of the WAM do not exist in the ATOAM because backtracking requires less bookkeeping operations, and tail recursion can be handled in most cases like a loop statement in procedural languages. An ATOAM-emulator-based Prolog system called B-Prolog has been implemented, which is available through anonymous ftp from *ftp.kyutech.ac.jp (131.206.1.101)* in the directory *pub/Language/prolog*. B-Prolog is comparable in performance with and can sometimes be significantly faster than emulated SICStus-Prolog. By measuring the numbers of memory and register references made in both systems, we found that passing arguments in stack is no worse than passing arguments in registers even if accessing memory is four times as expensive as accessing registers.

Categories and Subject Descriptors: D.1.6 [**Programming Techniques**]: Logic Programming; D.3.4 [**Programming Languages**]: Processors—*compilers*

General Terms: Experimentation, Languages

Additional Key Words and Phrases: Abstract machine, Prolog

1. INTRODUCTION

Prolog, unlike procedural programming languages, lacks explicit control constructs for specifying loops and facilities for accessing global variables. Loops are usually specified as tail-recursive procedures, and each procedure has to take enough arguments through which inputs are given and outputs are returned. For these reasons,

Author's address: Faculty of Computer Science and Systems Engineering, Kyushu Institute of Technology, 680-4 Kawazu, Iizuka, Fukuoka 820, Japan; email: zhou@mse.kyutech.ac.jp.

it is imperative that Prolog compilers generate good code for procedure calls and returns and handle tail recursion efficiently.

In the WAM, arguments of a procedure call are passed from a calling procedure (caller) to a called procedure (callee) through argument registers. They are first transferred from the caller's frame or some other argument registers to appropriate argument registers, and then, when necessary, they are transferred into the callee's frame. The WAM uses two kinds of frames for storing information associated with procedure calls. One is called *environment*, which stores the continuation program point and local variables, and the other is called *choice point*, which stores the machine status registers and the argument registers that need be restored upon backtracking. Warren [1983] made these decisions based probably on the following considerations. First, passing arguments in registers would be faster than passing arguments in stack frames because manipulating registers is faster than manipulating frame slots. Even in an emulator-based implementation where argument registers are actually simulated by using memory, manipulating registers is faster than manipulating frame slots because of the so-called direct addressing, i.e., computing addresses of "registers" at load time. Second, storing information about a call into two frames would be space efficient, because in some cases only an environment frame, and in some other cases only a choice point frame, is adequate.

However, the WAM is awkwardly inefficient for some kinds of programs. Consider the following motivating example:

```
intersect([],S,[]).
intersect([X|Xs],S2,[X|Ys]):-
      membchk(X,S2),!,
      intersect(Xs,S2,Ys).
intersect([X|Xs],S2,S3):-
      intersect(Xs,S2,S3).
```

This procedure computes the intersection of two sets represented as lists. It specifies a loop where backtracking is only a conditional jump. Ideally, the code for the procedure should create a frame only once and reuse it repeatedly in the loop. Unfortunately, due to the parameter passing and control stack management schemes adopted in the WAM, both a choice point and an environment have to be created and discarded each time the loop is executed, and the argument register storing `S2` has to be saved in both frames. By using the factoring [Van Roy 1990] and flattening techniques used in KL1 programming [Ueda and Chikayama 1990], the procedure can be translated into:

```
intersect([],S,[]).
intersect([X|Xs],S2,S3):-
      membchk_aux(X,S2,Ok),
      intersect_aux(Xs,S2,S3,X,Ok).
intersect_aux(S1,S2,S3,X,Ok):-
      Ok=:=1,!,
      S3=[X|S4],
      intersect(S1,S2,S4).
intersect_aux(S1,S2,S3,X,Ok):-
      intersect(S1,S2,S3).
```

The procedure call `membchk_aux(X,S2,Ok)` is similar to `membchk(X,S2)`, but takes a new argument `Ok` that will be bound to 1 or 0 depending on whether `membchk(X,S2)` succeeds or not. Although no choice point manipulation is necessary for the translated program, an environment still needs be created and discarded repeatedly in the WAM.

The example program is a common pattern of most Prolog programs where loops are specified as tail-recursive procedures and where the condition for choosing clauses is not inline. To eliminate the above-mentioned inefficiencies of the WAM, we designed and implemented a new abstract machine, called ATOAM, that differs from the WAM mainly in that (1) arguments are passed directly into stack frames, (2) only one frame is used for each procedure call, and (3) procedures are translated into matching trees if possible, and clauses in each procedure are indexed on all input arguments. The ATOAM has the following advantages over the WAM: (1) backtracking becomes simpler because no argument registers need be saved and restored for backtracking, and some procedures with deep guards, i.e., noninline tests, can be treated as determinate; and (2) tail recursion is converted into iteration, and thus it becomes unnecessary for tail-recursive procedures to create and discard frames repeatedly in many cases. The disadvantage of the ATOAM is that it lost some merits available in the WAM, such as the frame-trimming technique.

The readers are assumed to be familiar with compiler design for procedural languages and Prolog programming. The knowledge about the WAM is not mandatory, albeit helpful, for understanding the ATOAM architecture. In the next section, we give a brief and informal introduction to the WAM. In Section 3, we define two intermediate representation forms of programs called *canonical-form Prolog* and *matching tree*, respectively. In Section 4, we describe the architecture of the ATOAM. In Section 5, we present the storage allocation algorithm, and in Section 6 we focus on the tail recursion elimination technique. In Section 7, we report the experimental results. In Section 8, we compare thoroughly the ATOAM with the WAM and some other variants, and in Section 9, we point out some further improvements.

2. THE WAM: AN EXCURSION

The WAM is an abstract machine that consists of a memory architecture and an instruction set for encoding Prolog programs. We briefly survey the WAM, answering questions about how terms are represented, how unifications are compiled, how arguments are passed, and how the control stack is managed. The reader is recommended to refer to Ait-Kaci [1991] and Van Roy [1994] for the details.

2.1 Term Representation

There is a data area called *code area* that contains, besides instructions compiled from programs, a symbol table that stores information about the atom, function, and procedure symbols appearing in the programs. There is one record for each symbol in the symbol table storing such information as the *name*, *arity*, *type*, and *entry point* if the symbol is defined.

A term is represented by a word containing a value and a tag. The tag distinguishes the type of the term. It may be `REF` denoting a reference, `INT` denoting an integer, `ATM` denoting an atom, `STR` denoting a structure, or `LST` denoting a list.

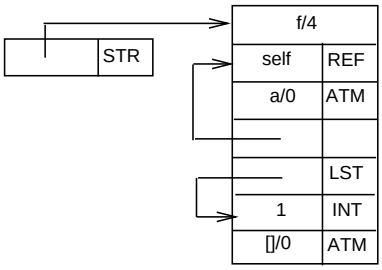

Fig. 1. Internal representation of `f(X,a,X,[1])`.

The value of a term is an address except when the term is an integer (in this case, the value represents the integer itself). The address points to a different place depending on the type of the term. The address in a reference points to the referenced term. An unbound variable is represented by a self-referencing pointer. The operation that looks for the value at the end of a reference chain is called *dereference*. The address in an atom points to the record for the atom symbol in the symbol table. The address in a structure $f(t_1,\ldots,t_n)$ points to a block of $n+1$ consecutive words in an area called *heap* where the first word points to the record for the functor f/n in the symbol table, and the remaining n words store the n components of the structure. The address in a list $[H|T]$ points to a block of two consecutive words in the heap where the first word stores the car H, and the second word stores the cdr T.

For example, Figure 1 shows a possible representation of the term `f(X,a,X,[1])`, where self is a self-referencing pointer.

2.2 Compiling Unification

A general unification procedure is costly, since it has to check the types of the terms to be unified. In the WAM, a unification call is encoded into a sequence of specific unification instructions if either operand is known at compilation time. For example, the code for the call `X=a` performs the unification as follows. If `X` is a variable, then bind `X` to `a`; if `X` is an atom, then test whether it is equal to `a`; otherwise, it is fail. The compiled code does no test of the type of the second operand.

2.3 Parameter Passing and Stack Management

In the WAM, clauses in each procedure are compiled separately. The code for the head of each clause performs unification between a call and the head, and the code for the body passes arguments and control to the callees and handles control returns. Arguments are passed from the caller to the callee through argument registers: the first one is placed in the first argument register; the second one is placed in the second argument register, and so on.

The argument registers need be saved when there are multiple candidate clauses to be tried. Some of them also need be saved when the callee is not binary. Consider the second clause in our motivating example:

```
intersect([X|Xs],S2,[X|Ys]):-
      membchk(X,S2),!,
      intersect(Xs,S2,Ys).
```

Before the call `membchk(X,S2)` is executed, the second argument register storing `S2` has to be saved because its content may be destroyed in the execution of `membchk`. In the WAM, an environment frame of the following structure is pushed onto the local stack before a nonbinary clause is executed:

```
Parent:     Parent environment
CP:         Continuation program point
Y1,...,Ym:  Permanent variables
```

where `Y1,...,Ym` are variables, called *permanent variables*, that can survive longer than a call. In the clause, the variables `Xs`, `S2`, and `Ys` are permanent.

The code for separate clauses in a procedure is connected by indexing and backtracking instructions. The indexing code divides the clauses into several groups, based on the main functors of the first arguments of the heads. For each group that has multiple candidates, backtracking instructions are generated to ensure that when one clause fails the next one will be tried. Consider again our example procedure. For a call whose first argument is a list, the indexing code will classify the second and the third clauses into the same group, and thus backtracking instructions connecting the code for these two clauses will be generated.

Backtracking instructions create and manipulate *choice points* which have the following fields:

```
P:          Alternative program point
CP:         Continuation program point
E:          Current environment frame
B:          Current choice point frame
T:          Top of the trail stack
H:          Top of the heap
X1,...,Xn:  Argument registers
```

For our example procedure, before the second clause is entered, a choice point is created, whose fields have the following meaning: `P` points to the code of the third clause such that control can be moved there when the second clause fails; `CP` stores the continuation program point to go to when the second clause succeeds; `E` points to the latest environment frame which will become the current one after the second clause succeeds; `B` points to the parent choice point frame; `T` points to the top of another stack called trail stack where the updates done to the arguments by the second clause are stored; `H` points to the top of the heap with which the space occupied by the terms created by the second clause can be released before execution backtracks to the third clause.

3. INTERMEDIATE REPRESENTATION

Figure 2 depicts the compilation phases in the B-Prolog compiler. The *specialization* phase translates a given program into canonical-form Prolog where input and output unifications are separated and where determinism of clauses is denoted

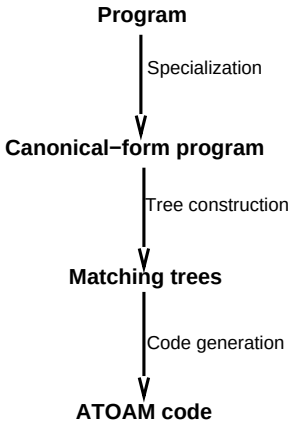

Fig. 2. Compilation phases.

explicitly. The *tree construction* phase constructs a tree, called *matching tree*, for every procedure in the canonical-form program. The *code generation* phase computes the properties of the matching trees and generates executable code based on them.

3.1 Canonical-Form Prolog

In canonical-form Prolog, each procedure is defined by a sequence of *matching clauses* in the form of

```
H:-G : B. (1)
```

or

```
H:-G ? B. (2)
```

H is an atomic formula; G and B are two sequences of atomic formulas; the colon (:) is called a *determinate choice operator*; and the question mark (?) is called a *nondeterminate choice operator*. H is called the head; G is called the guard; and B is called the body of the clause.

One-directional matching rather than full unification is used to choose clauses for a selected goal. A clause is applicable to a goal `C` if `C` matches the head, i.e., the head becomes identical to the goal after a substitution is performed to it ($H\theta = C$), and the guard succeeds ($G\theta$). The choice operator : indicates that the goal is committed to the clause, and thus the remaining clauses will not be tried when `B` fails; whereas, the ? indicates that when B fails, the remaining clauses will be tried.

A procedure is said to be *determinate* when all the choice operators in its definition are determinate; otherwise, it is said to be *nondeterminate* when the ? choice operator is used at least once in its definition. A procedure is said to be *globally determinate* if it is determinate and if no procedure called directly or indirectly by the calls in the bodies is nondeterminate. A call is said to be *inline* if its evaluation does not invoke any procedure. A procedure is said to be *binary* if the guard of any clause in it consists only of inline calls and if the body consists of at most one

noninline call to the left of which there may exist some inline calls. A procedure is said to be *flat* if the guard of any clause in it consists only of inline calls. A procedure is said to be *nonflat* if the guard of at least one clause is *deep*, i.e., contains noninline calls. The guard G must be a sequence of calls to globally determinate procedures. No call in it can bind any variable in the head unless it succeeds and unless the : is guaranteed to follow it.

Recall the `intersect/3` procedure. Suppose the first two arguments are inputs and that the third argument is output; then the procedure can be translated into

```
intersect([],S2,S3):-true : S3:=[].
intersect([X|Xs],S2,S3):-
    membchk(X,S2) :
    S3:=[X|Ys],
    intersect(Xs,S2,Ys).
intersect([X|Xs],S2,S3):-true :
    intersect(Xs,S2,S3).
```

where `:=/2` denotes assignment. This procedure is determinate because only determinate choice operators are used, and it is nonflat because the call `membchk(X,S2)` is not inline.

3.2 Specialization Algorithm

We can translate a Prolog program into canonical-form Prolog by moving all unifications in the head into the body for every clause. For example, the procedure

```
p(a).
p(b).
```

can be translated into:

```
p(X):-true ? X=a.
p(X):-true : X=b.
```

However, the resulting program would be terribly inefficient because procedures in it tend to be nondeterminate.

The specialization algorithm adopted in the B-Prolog compiler is as follows. For each procedure, if there are two consecutive clauses whose heads have nonvariable terms in the same argument position, then specialize the procedure into two: one taking care of the input case and the other taking care of the output case of the argument. The specialization is done only for one argument position. For example, the example procedure shown above is specialized into

```
p(X):-var(X) : p_aux(X).
p(a):-true : true.
p(b):-true : true.
p_aux(X):-true ? X:=a.
p_aux(X):-true : X:=b.
```

This algorithm, although simple, is far from satisfactory because (1) it duplicates the code for procedures, (2) it does not do global program analysis and thus cannot identify deep guards, and (3) the resulting procedures cannot be indexed on more than one argument.

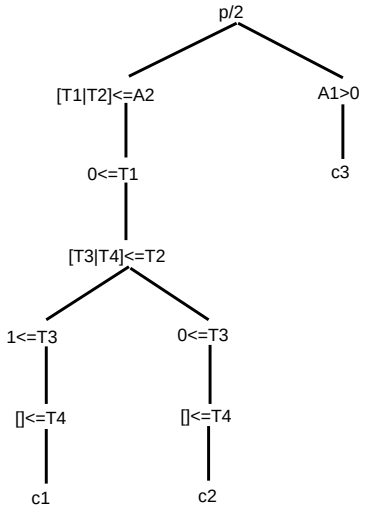

Fig. 3. The matching tree for p/2.

3.3 Matching Tree

Procedures in a canonical-form program are translated into matching trees one to one. A matching tree consists of a root, test nodes, and leaves. Each path from the root to a leaf corresponds to a clause in the procedure. The test nodes in the path represent the head and the guard, and the leaf represents the body of the clause. The purpose of this translation is to merge the tests shared by multiple clauses into one and eliminate redundant evaluation of shared tests [Zhou 1993].

Consider, for example, the following canonical-form procedure:

```
p(A1,[0,1]):-true ? true.
p(A1,[0,0]):-true ? true.
p(A1,A2):-A1>0 : s(A1,A2).
```

The matching tree constructed for the procedure is shown in Figure 3, where `<=/2` denotes one-directional matching. This tree is not optimal in terms of the number of nodes. If `[]<=T4` is attached to the tree before `1<=T3` and `0<=T3`, then the resulting tree will contain one less node.

A leaf is said to be *determinate* if its corresponding clause is determinate. A matching tree is said to be *determinate* if its corresponding procedure is determinate. The *neighboring node* of a node u is the sibling node to the right of u if there is such a sibling; otherwise, it is the neighboring node of the parent of u. The neighboring node of the root is *fail*. The *alternative node* of a node u is the node to go to after u fails. It is computed as follows: if u is a determinate leaf, then it is *fail*; otherwise, it is the nearest neighboring node of u that is not mutually exclusive with any ancestor of u. The alternative and neighboring nodes of a node may not be the same.

For example, in Figure 3, the alternative node of `1<=T3` is `0<=T3`, its neighboring

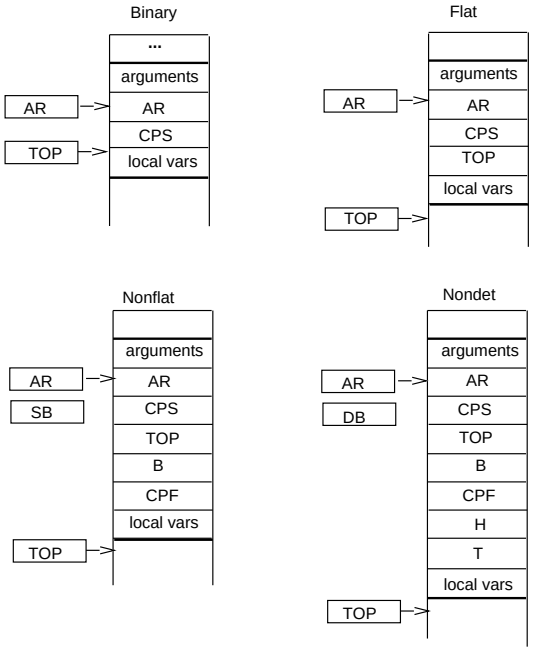

Fig. 4. The structures of frames.

node, but the alternative node of `[]<=T4` below `1<=T3` is `A1>0` because its neighboring node `0<=T3` is mutually exclusive with `1<=T3`.

A call to a procedure is evaluated by visiting the nodes in the corresponding matching tree in preorder. When a node succeeds, its leftmost child is visited; otherwise, its alternative node is visited.

4. THE ATOAM

4.1 Data Areas

The ATOAM uses all the data areas used by the WAM. The *heap* stores terms created during execution. The *trail* stack stores those variables that must be unbound upon backtracking. The *control* stack stores frames associated with procedure calls. Each time when a procedure is invoked by a call, a frame (see Figure 4) is placed on top of the control stack unless the frame currently on top of the control stack can be reused.

The structures of frames differ for different types of procedures. For binary procedures that are determinate, a frame contains the arguments, the AR slot pointing to the parent frame, the CPS slot storing the continuation program point to go to on success, and some local variables.

For flat procedures that are determinate, a frame contains, in addition to those in a frame for a binary procedure, a slot called TOP that points to the top of the control stack.

For procedures that are nonflat but determinate, a frame contains, in addition

to the slots in a frame for a flat procedure, two other slots: the B slot pointing to the frame of the latest predecessor of the call that has alternative program points to be tried and the CPF slot storing the alternative program point to go to after a branch if the procedure fails.

For procedures that are nondeterminate, a frame contains, besides the slots in a frame for a nonflat procedure, two other slots: the H slot pointing to the top of the heap and the T slot pointing to the top of the trail stack.

In the following, we will denote arguments as A_1, A_2, and so on, and local variables as Y_1, Y_2, and so on. In fact, arguments and local variables are identified internally by different offsets with respect to the AR slot. We will call the frame of a nondeterminate procedure a *deep choice point*, and the frame of a nonflat procedure a *shallow choice point*.

4.2 Registers

The current machine status is indicated by the following group of registers:

P:	Current program pointer
TOP:	Top of the control stack
AR:	Current frame
H:	Top of the heap
T:	Top of the trail stack
DB:	Latest deep choice point
SB:	Latest shallow choice point
HB:	H slot of the latest deep choice point

The last three registers need further explanation. The DB register points to the latest deep choice point. The SB register points to the latest shallow choice point. When a failure occurs, the contents of DB and SB are compared. If DB is younger than SB (younger(DB,SB)), then deep backtracking is invoked; otherwise, shallow backtracking is invoked. DB and SB never have the same contents. The differences between shallow and deep backtracking will become clear later.

The HB register is used in checking whether or not a variable needs to be trailed. It always holds the content of the H slot in the latest deep choice point. When a variable is bound, if it is older than HB or DB, then it is trailed.[1]

The two registers named S and RW of the WAM are also used. The S register points to the next component of a compound term to be processed, and the RW denotes the mode of unification: *read* or *write*. In addition, some temporary registers denoted as X_1, X_2, and so on are used.

4.3 Instructions

The instructions are classified into basic and control instructions. The basic instructions consist of *jump*, *hash*, *fetch*, *move*, *build*, *unify*, *unify argument*, *assign*, and *parameter-passing* instructions. The control instructions consist of *allocate*, *call* and *return*, and *backtracking* instructions. Figure 5 shows the instruction set.

[1]One register, say B, that points to the latest choice point, whether deep or shallow, would be sufficient if we let the code pointed to by the CPF slot handle failure. However, it is beneficial using two registers because the variables that are older than SB but younger than DB need not be trailed when being bound.

```
Jump and conditional jump:
   jump L
   jmpn_eq_struct Z,f/n,L
   jmpn_eq_atom Z,a,L
   jmpn_eq_int  Z,N,L
   jmpn_var Z,L
   jmp_var Z,L
   switch_on_list Z,L1,L2

Hash:
   hash Z,((v1,L1),...,(vn,Ln),L)

Fetch:
   fetch_var Z
   fetch_void N

Move:
   move_struct Z,f/n
   move_atom Z,a
   move_int Z,N
   move_list Z
   move_var Z
   move_value Z1,Z2

Assign:
   assign_struct Z,f/n
   assign_atom Z,a
   assign_int Z,N
   assign_list Z
   assign_value Z1,Z2

Build:
   build_struct f/n
   build_atom a
   build_int N
   build_list
   build_var Z
   build_value Z
   build_void N

Unify:
   unify_struct Z,f/a
   unify_atom Z,a
   unify_int Z,N
   unify_list Z
   unify_value Z1,Z2

Unify argument:
   unify_arg_struct f/n
   unify_arg_atom a
   unify_arg_int N
   unify_arg_list
   unify_arg_var Z
   unify_arg_value Z
   unify_arg_void N

Parameter passing:
   para_struct f/n
   para_atom a
   para_int N
   para_list
   para_var Z
   para_value Z
   para_void N

Control:
   call q
   execute q
   allocate_flat N
   allocate_nonflat N
   allocate_nondet N
   return
   fork L
   fail
   commit
   cut
   jump_on_det L
   save_ht_jump L
```

Fig. 5. The ATOAM instruction set.

4.3.1 *Basic Instructions.* Basic instructions perform data movement and different kinds of unifications. They must obey the following two rules:

(1) References on the heap must never point to the control stack.

(2) No variable can reference any variables that are younger than it.

These rules are necessary for preventing pending references, i.e., pointers pointing to a discarded data area, from occurring.

Conditional Jump. These instructions correspond to inline tests. A conditional jump instruction tests a condition and moves control to the destination if the condition is satisfied. For example, the instruction *jmpn_eq_atom A1,a,L* moves control to L if $A1$ is not equal to the atom a; otherwise, it moves control to the next instruction. The string *jmpn* is the abbreviation for *jump on not.*

Hash. A hash instruction takes the form of $hash\ Z, ((v_1, l_1), \ldots, (v_n, l_n), l_{n+1})$. It tests the term in Z and moves control to a destination obtained by applying a hash function to the term. All $v_i's (1 \leq l_i \leq n)$ are mutually distinctive atoms, integers, or functors each of which is associated with an address. l_{n+1} is the address to go to when the hashing value of Z is different from that of any v_i.

Fetch. There are one or more *fetch* instructions following *jmpn_eq_struct* and *jmpn_eq_list* that are responsible for fetching the components of the compound term just tested. The *jmpn_eq_struct* and *jmpn_eq_list* instructions set the S register to let it point to the first component when the tests succeed. After fetching a component, a fetch instruction will increment the S register and let it point to the next component.

Move. A move instruction moves an operand to a frame slot or a register, or it moves data between frame slots and registers. For a move instruction, if the source is a frame slot, then the source must be dereferenced. If the dereferenced value is an unbound variable in the current frame, then the instruction must globalize it by creating a new variable on the heap and letting the original variable point to the new variable, for example:

```
p(U,V):-q(V,U).
```

It is possible for the tail call `q(V,U)` to reuse the frame of `p(U,V)`. The two arguments must be swapped before `q(V,U)` is executed. The generated code is:

```
p/2: move_value X1,A1
     move_value A1,A2
     move_value A2,X1
     execute q/2
```

The first two instructions have to dereference `A1` and `A2` and globalize them if necessary, because `A1` and `A2` are frame slots. To understand this necessity, the reader is encouraged to check what will happen if $A1$ and $A2$ happen to be two different unbound variables in the current frame.

Build. These instructions are responsible for building components of a compound term on top of the heap. The *build_value Z* instruction must dereference Z if Z is a frame slot. If the dereferenced value of Z is an unbound variable on the stack,

then the unbound variable must be globalized.[2] The *build_struct* and *build_list* are used only to encode the last components of compound terms. There are similar instructions in some WAM variants [Carlsson 1990].

Unify and Unify-Argument. Unify instructions correspond to calls of the =. There are one or more unify-argument instructions following *unify_struct* and *unify_list* that are responsible for unifying components of the compound term. The *unify_struct Z,f/n* instruction will set the RW and S registers as follows. If Z is a compound term whose functor is equal to f/n, then set RW to *read* mode, and set S to let it point to the first component of the compound term; otherwise, if Z is an unbound variable, then set RW to *write* mode, but do not touch the S register. The *unify_list* instruction behaves similarly. Unify-argument instructions behave differently according to the content of RW. In write mode, they act like build instructions, whereas in read mode, they unify the next component pointed to by the S register with the operand.

Assign. Assign instructions correspond to calls of the :=. Unlike unify instructions, they need not set the RW and S registers.

Parameter Passing. A parameter-passing instruction passes an argument to the frame of the callee by placing it in the position pointed to by the TOP register. Each time after an argument is passed, the TOP register is decremented.

4.3.2 *Control Instructions.* A call is said to be *globally determinate* if it does not leave any choice points behind after it succeeds. The condition for globally determinate procedures is stronger than that for globally determinate calls. A call of a globally determinate procedure is globally determinate, but not vice versa. For example, consider the following procedure:

```
p(X):-true ? X=a,!.
p(X):-true : X=b.
```

This procedure is not determinate, but any call to it is globally determinate.

For a clause

```
P:- G OP Q,R.
```

`Q` and `R` are two noninline procedure calls, and `OP` is a choice operator. If `OP` is : and `Q` is determinate, then the generated code would have the following skeleton:

```
allocate a frame
... % code for P and G
... % code for ':'
pass arguments of Q
call Q
rearrange arguments of R
```

[2] In many recent systems such as Aquarius-Prolog [Van Roy 1990], BinWAM [Tarau 1991], and VAM [Krall and Berger 1995], unbound variables are stored on the heap. This new scheme makes it unnecessary to globalize unbound stack variables in the *move_value* and *build_value* instructions. In addition, it also makes a trailing check cheaper. We retain the WAM's scheme of storing unbound variables on the stack because the new scheme increases the lengths of reference chains and takes more heap space. In fact, we found that the WAM's scheme performs no worse on average than the new scheme.

```
execute R
```

If OP is : but Q is not known be to determinate, then the generated code looks like

```
  allocate a frame
  ... % code for P and G
  ... % code for ':'
  pass arguments of Q
  call Q
  jump_on_det L
  pass arguments of R
  call R
  return
L:rearrange arguments of Q
  execute R
```

Two streams of code are generated for the tail call, one reusing the current frame and the other not. The *jump_on_det* instruction determines which stream is to be executed.

```
jump_on_det L:
   if (younger(AR,DB)) % AR is younger than DB
      P = L;
```

If OP is ?, then the following skelton of code will be generated:

```
allocate a frame
... % code for P and G
fork instruction
pass arguments of Q
call Q
pass arguments of R
call R
return
```

The *fork* instruction sets the backtracking point to go to after the body fails. The tail call R is treated in the same way as Q, and the frame of the head procedure is not reused.

Allocate. There is one allocate instruction at the entry point of every procedure except when the procedure is binary as well as determinate. The allocate instruction is responsible for fixing the size of the frame and saving some bookkeeping registers. In case when the procedure is binary as well as determinate, the part of frame created by the caller is enough for executing the callee, and thus no allocate instruction is necessary.

—*allocate_flat N* is responsible for allocating a frame for those determinate procedures that are flat, but not binary.

```
allocate_flat N:
   TOP = AR-N;
   AR->TOP = TOP;
```

—*allocate_nonflat N* is responsible for allocating a frame for those determinate procedures that are nonflat.

```
allocate_nonflat N
   TOP = AR-N;
   AR->TOP = TOP;
   AR->B = SB;
   SB = AR;
```

—*allocate_nondet N* is responsible for allocating a frame for nondeterminate procedures.

```
allocate_nondet N:
   TOP = AR-N;
   AR->TOP = TOP;
   AR->B = DB;
   AR->H = H;
   AR->T = T;
   HB = H;
   DB = AR;
```

Call and Return. There are two instructions for noninline calls, one named *call* corresponding to an intermediate call and the other named *execute* corresponding to a tail call that can reuse the frame of the head procedure. The *return* instruction returns control to the caller.

—*call q* fills in the AR and CPS slots of the frame for q and moves control to q.

```
call q:
   *TOP = AR;
   AR = TOP;
   AR->CPS = P;
   P = entrypoint(q);
```

—*execute q* just moves control to q.

```
execute q:
   P = entrypoint(q)
```

—*return*, before returning control to the caller, computes the new top of the control stack.

```
return:
   P = AR->CPS;
   AR = AR->AR;
   TOP = younger(AR,DB) ? AR->TOP : DB->TOP;
```

Backtracking. Backtracking instructions are responsible for treating failures.

—*fork L* lets the CPF slot of the current frame hold L.

```
fork L:
   AR->CPF = L;
```

—*fail* first determines the type of the failure. If SB is younger than DB, then it invokes shallow backtracking; otherwise, it invokes deep backtracking.

```
fail:
   if (younger(SB,DB)){
      AR=SB;    /* shallow backtracking */
      P = AR->CPF;
      TOP = AR->TOP;
   } else {     /* deep backtracking */
      AR = DB;
      H = HB;
      P = AR->CPF;
      TOP = AR->TOP;
      while (T > AR->T) unbind(--T);
   }
```

Notice that since the H register is not saved and restored for shallow backtracking, the data created by a guard become garbage after the guard fails. For this reason, the specialization algorithm should not classify those calls as guard calls that create data on the heap.

—*commit*, which corresponds to a : in a nonflat procedure, discards the latest shallow choice point.

```
commit:
   SB = AR->B;
```

—*cut*, which corresponds to a : in a nondeterminate procedure, discards the latest deep choice point.

```
cut:
   DB = AR->B;
   HB = DB->H;
```

4.3.3 *Example.* Recall our motivating example. The instructions generated for the canonical-form intersect procedure are listed in Figure 6. When the procedure is invoked, the caller should have allocated a frame for it with the arguments, the AR and CPS fields filled in. The *allocate_nonflat* instruction completes the job of allocating a frame for the procedure and lets the frame to be the latest shallow choice point. It also fills in the TOP slot such that the top of the control stack can be restored after membchk(X,S2) in the guard fails. The *switch_on_list A1,C2,COM_FAIL* is a conditional jump instruction that moves control to the next instruction if A1 is a nil, to C2 if A1 is a list, otherwise to COM_FAIL, the label of a *commit* instruction followed by a *fail* instruction. Each *commit* instruction corresponds to a : choice operator in the procedure. The two arguments [X|Xs] in the heads of the last two clauses are transformed into a test node in the matching tree, and thus X and Xs are fetched only once for each call. The *fork C3* instruction sets the alternative program pointer. After membchk(X,S2) fails, control will be moved to C3. The return instruction reclaims the current frame before moving control to the caller. The *execute* instructions correspond to the tail calls. The current frame is reused by the tail calls.

```
intersect/3:
   allocate_nonflat 6
C1:switch_on_list A1,C2,COM_FAIL
   commit              % intersect([],S2,S3):-true :
   assign_nil A3       % S3:=[].
   return
C2:fetch_var Y1        % intersect([X|Xs],S2,S3):-
   fetch_var A1
   fork C3
   para_value Y1
   para_value A2
   call membchk/2      % membchk(X,S2)
   commit              % :
   assign_list A3
   build_value Y1
   build_var A3        % S3:=[X|Ys],
   execute intersect/3 % intersect(Xs,S2,Ys).
C3:commit              % true :
   execute intersect/3 % intersect(Xs,S2,S3).
```

Fig. 6. ATOAM code for intersect.

5. STORAGE ALLOCATION

A variable is stored in either a register or a frame slot. This is decided by the following four rules:

(1) *Variables that are shared by more than one branch in a matching tree are stored in frame slots.* This rule is important for avoiding the components of shared compound arguments being fetched many times. For example, the variables `X` and `Xs` in the `intersect` procedure occur in the last two clauses and are thus stored in frame slots.

(2) *Variables occurring in more than one chunk are stored in frame slots.* A *chunk* is defined to be a noninline call in the guard or body of a clause to the left of which there may exist a sequence of inline calls. The head of a clause is counted into the first chunk. This rule may reduce data movement between registers and frame slots. If a variable that occurs in more than one chunk is stored in a register, then before the noninline call is executed, the register has to be saved into the frame because the content of the register may be destroyed during the execution of the noninline call.

(3) *Variables occurring in only one chunk in only one branch in a matching tree are stored in registers except when they occur as arguments in some tail calls.* This rule can be illustrated by using the following example:

```
p([a|V]):-p(V).
```

Because `V` must be eventually placed in the frame, storing it in a register would increase data movement.

(4) *Frame slots and registers allocated to variables are reclaimed as early as possible such that they can be reused to store other variables.* This rule is important for reducing data movement and for minimizing the size of frames and the

possibility of register overflow. A variable is said to be *inactive* if it cannot be accessible in both forward or backward execution. The storage allocated to a variable can be reclaimed immediately after the call in which the variable becomes inactive.

The task of detecting the inactiveness of variables becomes difficult due to aliasing. Consider, for example, the following clauses:

```
a:-b(U,V),c(U),d(V,W),e(W),f.
b(V,V).
c(_).
d(a,W).
e(W):-write(W).
```

Despite the calls `b(U,V)` and `c(U)` in the first clause being globally determinate, the variable `U` is still active even after `c(U)` because `V` points to `U`, and `V` is still active. If the frame slot allocated to `U` is reused to store `W`, then `V` and `W` would be erroneously treated as the same variable.

The aliasing problem never occurs if all unbound variables are stored in the heap. In B-Prolog, we modified the unification procedure to ensure that no stack variable can reference another stack variable.

6. TAIL RECURSION ELIMINATION

Like in any other programming languages, programs in Prolog typically spend much of their execution time in loops. In Prolog, loops are specified as recursive procedures, most of which are tail recursive or can be transformed into tail-recursive ones [Debray 1988]. The *tail recursion optimization* or *last-call optimization* technique in the WAM enables tail calls to reuse the frame for the parent call if the frame lies on top of the control stack.

Although, with tail recursion optimization, tail-recursive procedures consume constant stack space, the WAM code is significantly slower than the counterpart in procedural languages because frames have to be allocated and deallocated repeatly in case the procedures are not binary. Meier [1991] has noticed the problem and proposed a method for converting tail recursion into iteration in the WAM without any modification of the WAM. Nevertheless, Meier's method requires argument registers to be saved in an environment frame at the entry point of every tail-recursive clause. The method would become very complicated if we want to implement the tree compilation techniques that can merge test code shared by several clauses. Meier has never implemented his method, though he has reported possible gains obtainable by tail recursion elimination.

Recall the code shown in Figure 6. The code consumes constant stack space because the frame of the parent call is reused by the tail calls. It is, however, still unsatisfactory because the allocate instruction is executed repeatedly. The `execute` instruction moves control to the entry point of the procedure. The first instruction in the code for the procedure is an allocate instruction that allocates a new frame that is of the same size and contains almost the same information as the old one. Ideally, the frame should be allocated only once and be reused in the loop repeatedly. In this section, we show how to achieve this.

6.1 Flat Procedures

Let $H{:-}G : B_1, \ldots, B_n$ be a tail-recursive clause in a flat procedure. The *execute* instruction for B_n can be safely replaced by a *jump* instruction. When H is a binary determinate procedure, then the *jump* instruction jumps to the first instruction in the code for the procedure; otherwise, it jumps to the instruction just below the allocate instruction.

Consider the following procedure:

```
p([]):-true : true.
p([X|Xs]):-true : foo(X),p(Xs).
```

Suppose `foo(X)` is a globally determinate call. The generated code is

```
p/1:allocate_flat 3
C1: switch_on_list A1,C2,F
    return
C2: fetch_var X1
    fetch_var A1
    para_value X1
    call foo/1
    jump C1
```

6.2 Nonflat Procedures

Let $H{:-}G : B_1, \ldots, B_n$ be a tail-recursive clause in a nonflat procedure. There should be a *commit* instruction generated for the : . Just like for flat tail-recursive procedures, we replace the *execute* instruction for B_n with a *jump* instruction that jumps to the instruction just below the allocate instruction. If $B_1, \ldots, B_{n-1}$ never fail, then we also remove the *commit* instruction; otherwise, if some call in $B_1, \ldots, B_{n-1}$ may fail, then we replace the *commit* instruction with a *fork COM_FAIL* instruction where the *COM_FAIL* is the label of a *commit* instruction followed by a *fail* instruction.

It is in general impossible to decide whether or not a call may fail. However, many built-in procedures such as *write/1* do not fail. Using this information, the compiler is able to decide that some user-defined procedures never fail.[3]

Figure 7 shows the code for the intersect procedure where tail-recursive calls are converted into jumps. As there is no call to the left of the tail calls in the tail-recursive clauses that may fail, the two *commit* instructions are removed.

6.3 Nondeterminate Procedures

Eliminating tail-recursive calls from nondeterminate procedures is a little complicated. Consider a tail-recursive clause $H{:-}G : B_1, \ldots, B_n$ in a nondeterminate procedure. We eliminate the tail-recursive call as follows: replace the *execute* instruction for B_n with a *save_ht_jump* L instruction where L is the label of the instruction just below the allocate instruction. The *save_ht_jump* L saves the H and T registers into the current frame and then jumps to L.

```
save_ht_jump L:
```

[3]The B-Prolog compiler does not do such analysis currently.

```
intersect/3:
   allocate_nonflat 6
C1:test_on_list A1,C2,COM_FAIL
   commit
   assign_nil A3
   return
C2:fetch_var Y1
   fetch_var A1
   fork C3
   para_value Y1
   para_value A2
   call membchk/2
   assign_list A3
   build_value Y1
   build_var A3
   jump C1
C3:jump C1
```

Fig. 7. Optimized code for intersect.

```
AR->H = H;
HB = H;
AR->T = T;
DB = AR;
P = L;
```

This instruction is simpler than the *allocate_nondet* instruction. It does not need to set the TOP registers and the TOP slot in the frame. Nor does it need to do an overflow check for the control stack.

Consider the following procedure:

```
member(X,[Y|Ys]):-true ? X=Y.
member(X,[_|Ys]):-true : member(X,Ys).
```

The generated code is

```
member/2:
   allocate_nondet 8
C1:switch_on_list A2,L,CUT_FAIL
   jump CUT_FAIL  % A2=[]
L: fetch_var Y1   % A2=[_|_]
   fetch_var A2
   fork C2
   unify_value A1,Y1
   return
C2:cut
   save_ht_jump C1
```

where CUT_FAIL is the label of a *cut* instruction followed by a *fail* instruction. The tail recursion elimination technique for nondeterminate procedures is conservative albeit safe. In many cases, a much more specific instruction than *save_ht_jump* can be used. For example, in the code for member/2, the cut instruction can be

removed, and the `save_ht_jump` instruction can be replaced by a `jump` instruction. These improvements have not yet been realized in the B-Prolog system.

7. PERFORMANCE EVALUATION

An ATOAM-emulator-based Prolog system called B-Prolog has been implemented. Besides standard Prolog programs, the compiler can also compile canonical-form programs. The compiler itself is written in canonical-form Prolog. To reduce the overhead of interpretation by the emulator, we introduced some complex instructions each of which can encode a sequence of basic instructions. The implemented abstract machine has 220 instructions in total.

In this section, we compare B-Prolog (version 2.0) with SICStus-Prolog (version 3.0), a well-tuned WAM-based commercial system developed by the Swedish Institute of Computer Science, and we report the following three sets of performance data for the Aquarius benchmark suite [Van Roy 1995]: *number of memory and register references*, *CPU times*, and *stack space requirements*. In the following, we will refer to the two systems as BP and SP, respectively.

7.1 Number of Memory and Register References

To simplify the comparison, we concentrate on the references made to the control stack[4] and temporary registers. Furthermore, we ignore the references made to the stack by dereference, variable bind, and variable reset. Such a simplification is reasonable because the numbers of memory references made by these operations should be almost the same in both abstract machines.

For each instruction in the ATOAM, the number of stack references it makes is equal to the number of Y operands, and likely the number of register references is equal to the number of X operands, except when the instruction is a control or a parameter-passing instruction. For each parameter-passing instruction, an additional stack reference is counted in because it manipulates the frame slot pointed to by the `TOP` register. The numbers of memory references made by control instructions are shown in Table I. SP supports garbage collection, which requires local variables to be initialized. To make the comparison fair, we modified the *allocate* instructions such that local variables are initialized in the same way.

Table II shows different ratios, where X_{sp} (X_{bp}) and Y_{sp} (Y_{bp}) denote the numbers of register references and stack references made by SP (BP). BP is better than SP concerning the total number of stack and register references for all the programs except for `nreverse` and `flatten`. `nreverse` is a typical program to which the WAM is well suited, in which most of the execution time is spent on executing `concat`, a binary procedure for concatenating two lists. The results demonstrate that most programs do not have the same characteristics as `nreverse`. In emulator-based implementations where the cost of accessing registers and that of accessing memory is almost the same, the ATOAM is obviously better than the WAM. Furthermore, the ATOAM would be no worse than the WAM for native code compilation on a computer if accessing registers is less than four times as fast as accessing memory in the computer.

[4]SP uses two separate stacks, namely, the environment stack and the choice point stack, to represent the control stack.

Table I. Number of Memory References Made by Control Instructions

Instructions	No. of Memory References
call q	2
execute q	0
allocate_flat N	1+(N-3)
allocate_nonflat N	2+(N-5)
allocate_nondet N	4+(N-7)
return	3
fork L	1
fail	3
commit	1
cut	2
jump_on_det L	0
save_ht_jump L	2

7.2 CPU Time

Table III shows the ratios of the CPU times taken by emulated SP to those taken by BP. SP supports native code compilation and threaded code emulation, where a jump table rather than a switch statement is used to perform control dispatches for instruction interpretation. The compared system is installed with these extensions excluded. Each program was run at least 10 times, and the mean value was taken. The ratios depend to a large extent on the choices of computers and C compilers. On a SPARC-2 with 64MB RAM, BP is on average 56% faster than SP if the SUN CC compiler is used, and 20% faster than SP if the GCC compiler is used. On a SPARC-10 with 96MB RAM, however, the ratios become 43% and 34%, respectively.

Generally, the more a program refers to memory and registers, the more CPU time it will take. For some programs such as `tak` and `sendmore`, this relationship is clear. However, the factor of memory and register references can be overwhelmed by some other factors, such as term representation method, instruction complexity, memory management method, overflow-checking method, and programming style in the real implementations. For example, for `nreverse`, although BP makes much more memory references than SP, BP is even faster than SP for all the four runs. This may result from two special instructions adopted in BP for handling lists.

Table IV compares the average performance between BP and different installations of SP.

7.3 Stack Space Requirements

Table V shows the ratios between various stack spaces required by SP to those required by BP. The control stack space for *tak* is not given because SP consumes 6000 more stack space than BP, and thus including this data in the table would render the mean value meaningless. The values at other entries marked with – are not given either because BP consumes no space.

On average, BP consumes 30% more trail stack space than SP. The main reason is that BP trails address-value pairs rather than only addresses as is done in SP. The reason why BP consumes 14 times more space than SP for `boyer` is not clear.

Table II. Comparison of the Numbers of Memory and Register References

Program	$\frac{X_{sp}}{X_{bp}}$	$\frac{Y_{sp}}{Y_{bp}}$	$\frac{X_{sp}+Y_{sp}}{X_{bp}+Y_{bp}}$	$\frac{X_{sp}+2\times Y_{sp}}{X_{bp}+2\times Y_{bp}}$	$\frac{X_{sp}+3\times Y_{sp}}{X_{bp}+3\times Y_{bp}}$	$\frac{X_{sp}+4\times Y_{sp}}{X_{bp}+4\times Y_{bp}}$
boyer	5.71	0.97	1.76	1.40	1.27	1.20
browse	4.37	0.94	1.59	1.30	1.19	1.13
chat_parser	24.46	0.75	1.45	1.11	0.99	0.93
crypt	2.56	0.27	1.00	0.70	0.58	0.51
derive	3.22	0.96	1.48	1.25	1.16	1.12
fast_mu	10.97	1.05	1.92	1.51	1.36	1.28
flatten	5.90	0.39	0.90	0.66	0.57	0.53
meta_qsort	5.52	0.72	1.10	0.92	0.86	0.82
mu	20.83	0.84	1.53	1.19	1.08	1.02
nand	14.66	1.01	1.73	1.38	1.26	1.20
nreverse	2.66	0.16	0.85	0.56	0.44	0.38
poly_10	3.46	0.87	1.46	1.20	1.10	1.05
prover	5.16	0.78	1.32	1.07	0.98	0.93
qsort	10.98	0.78	1.44	1.12	1.01	0.95
queens_8	8.48	0.35	1.18	0.79	0.64	0.57
query	5.44	0.62	1.37	1.02	0.90	0.83
reducer	5.53	0.82	1.40	1.13	1.03	0.98
sdda	7.36	0.79	1.33	1.08	0.98	0.94
sendmore	27.57	1.03	2.52	1.80	1.55	1.42
serialize	6.68	0.96	1.61	1.31	1.20	1.14
simple_analyzer	6.73	0.42	1.02	0.73	0.63	0.58
tak	9.22	1.14	2.02	1.61	1.46	1.38
unify	9.59	0.85	1.56	1.22	1.10	1.04
zebra	8.84	1.70	2.51	2.13	2.00	1.92
(arithmetic mean)	9.00	0.80	1.50	1.18	1.07	0.99

For some programs, such as `poly_10`, BP uses less space than SP. This is due to an optimization technique adopted in BP for reducing the number of trailings by reordering unifications and cuts.

BP consumes 30% on average more control stack space than SP because of several factors. First, ATOAM treats every procedure as a whole, and thus it has to allocate a frame large enough for executing all the clauses in the procedure. Second, for programs that require a very small number of frames, the ratios favor the WAM because they do not take into account the space of the shadow frame stored in registers. Third, the current implementation classifies each procedure as either determinate or nondeterminate. It does not extract partial determinism, for instance,

```
p(a):-B1.
p(a):-B3.
p(b):-B2.
p(c):-B4.
```

B-Prolog treats this procedure as a nondeterminate procedure. The compiled code creates a big frame even for those calls of the procedure whose first arguments are `b` or `c`.

Table III. Comparison of CPU Times

Program	SPARC-2		SPARC-10	
	$\frac{SP}{BP}$ (cc -O)	$\frac{SP}{BP}$(gcc -O3)	$\frac{SP}{BP}$ (cc -O)	$\frac{SP}{BP}$ (gcc -O3)
boyer	1.61	1.31	1.39	1.30
browse	1.71	2.13	1.45	1.40
chat_parser	1.54	1.13	1.33	1.07
crypt	1.34	1.09	1.30	1.80
derive	1.07	0.97	1.35	0.93
fast_mu	1.43	1.13	1.29	1.29
flatten	1.41	0.94	1.05	1.29
meta_qsort	1.27	0.83	0.97	0.82
mu	1.97	1.22	1.50	1.53
nand	1.82	1.48	1.75	1.62
nreverse	1.41	1.20	1.45	1.27
poly_10	1.38	1.25	1.31	1.19
prover	1.16	0.91	1.35	1.03
qsort	1.71	1.14	1.42	1.40
queens_8	1.97	1.11	1.84	1.28
query	1.24	0.90	1.20	1.80
reducer	1.61	1.10	1.29	1.15
sdda	1.47	1.19	1.31	1.20
sendmore	2.13	1.44	1.64	1.32
serialize	1.66	1.35	1.94	1.40
simple_analyzer	1.21	0.86	1.01	0.96
tak	2.11	1.73	1.88	1.89
unify	1.98	1.42	1.70	1.96
zebra	1.31	1.06	1.56	1.15
(arithmetic mean)	1.56	1.20	1.43	1.34

Table IV. Comparison of CPU Times of BP and Different Installations of SP (SP/BP)

SPARC-2				SPARC-10			
emulator			native	emulator			native
switch-cc	switch-gcc	threaded	code	switch-cc	switch-gcc	threaded	code
1.56	1.20	1.06	0.44	1.43	1.34	1.27	0.39

7.4 Good Canonical Form

The specialization algorithm adopted in B-Prolog suffers from several inefficiencies as shown in Section 3.2. It can index clauses on only one argument, though not necessarily the first one. Thus, a matching tree usually has only one level of test nodes between the root and the leaves. If indexing is done only on the first argument as is done in SP, then no slow-down occurs on average at all. The reason is that the benchmark programs are written with the WAM's first argument-indexing scheme in mind.

To see what a good specialization algorithm can achieve, we translated several benchmark programs into canonical form by hand and tested both their time and space efficiencies. The results are very encouraging. For instance, for the *boyer*

Table V. Comparison of Stack Space Requirements

Program	$\frac{SP}{BP}$(Control)	$\frac{SP}{BP}$(Global)	$\frac{SP}{BP}$(Trail)
boyer	0.58	1.80	0.07
browse	0.86	1.00	0.80
chat_parser	0.97	1.00	0.51
crypt	0.84	1.00	2.00
derive	0.45	1.01	-
fast_mu	0.82	1.00	0.57
flatten	0.83	0.65	0.69
meta_qsort	0.46	1.13	0.84
mu	0.68	1.01	0.50
nand	0.85	0.95	0.49
nreverse	0.81	0.97	-
poly_10	0.54	0.82	3.57
prover	0.83	0.81	0.65
qsort	0.72	1.00	0.50
queens_8	0.67	1.00	0.50
query	0.80	1.12	0.50
reducer	0.20	1.00	0.37
sdda	0.82	0.97	0.76
sendmore	0.79	-	0.50
serialize	0.73	1.01	0.50
simple_analyzer	2.16	0.60	0.78
tak	-	-	-
unify	0.46	0.87	0.19
zebra	0.80	0.92	0.50
(arithmetic mean)	0.77	0.99	0.75

program, the hand-translated canonical form is 31% faster, takes 70% less code area, and consumes 170 % less control stack space than the compiler-translated canonical form. The hand-translated canonical form does not touch the trail stack because all the procedures are determinate.

8. RELATED WORK

The scheme of handling procedure calls described in this article is similar to that adopted for handling procedure calls in conventional compilers of procedural languages [Aho et al. 1986]. Similar schemes had been adopted in early Prolog implementations [Campbell 1984; Clark and Tarnlund 1982]. In the abstract machine [Warren 1977], only one frame that is big enough for holding information for both forward execution and backtracking was used for each procedure call. After Warren invented the WAM, people shifted their attention from this old scheme to the WAM.

Compared with the WAM, the ATOAM has the following advantages:

(1) Backtracking in the ATOAM is much simpler than that in the WAM because no argument registers need be saved and restored. In addition, shallow backtracking is treated more efficiently than deep backtracking because less bookkeeping registers need be saved and restored, and the trail stack need not be touched when a guard, whether inline or not, fails.

(2) In the ATOAM, the tail calls in a procedure can reuse both the space of and the information in the frame for the procedure if the frame can be reused. If the tail calls recursively call the procedure itself, the frame needs not be allocated and deallocated repeatedly. In contrary, in the WAM, although the first calls in the bodies can reuse some argument registers, no information in a control stack frame can be reused. ATOAM can surplus the WAM a lot in speed for tail-recursive procedures that are not binary, such as the `intersect` procedure.

(3) One advantage of the ATOAM, which has not yet been addressed by now, is that the delay mechanism can be implemented efficiently [Zhou 1996]. Just like for backtracking, data movements between the heap and registers can be eliminated for delay if arguments are passed in the stack.

(4) In an emulator-based implementation, the destination to which the next argument is to be passed is stored in the TOP register, and the emulator does not need to fetch and interpret it.

The ATOAM has the following disadvantages:

(1) The ATOAM is inefficient for executing binary determinate programs for which no memory access is required in the WAM. This problem is not so severe in an emulator-based system because the difference between the costs of accessing a register and accessing a frame slot is not so big. However, for a native compiler, this demerit may result in a big slow down compared with the WAM.

(2) The ATOAM may require a little more control stack space than the WAM. In the original WAM, clauses are assumed to be compiled separately. In contrast to that, a procedure is compiled as a whole in the ATOAM. Thus, a frame that is large enough for executing all the clauses in a procedure has to be allocated in the ATOAM. In addition, the frame-trimming technique in the WAM can trim frames dynamically without checking determinacy of procedures, but such a thing is impossible in the ATOAM. The ATOAM compiler does do a kind of frame trimming at compilation time, but it achieves almost nothing if no information about the determinacy of procedure calls is available. Let us consider the following clause:

```
a:-b(A),c(A,B),d(B,C),e(C),f(B).
```

If the procedure calls b(A) and c(A,B) are known to be determinate, then the frame slot taken by the variable A can be reallocated to C. However, if the compiler does not know any information about the determinacy, it has to allocate a new slot to C. Furthermore, the code for the tail call is duplicated.

Most of the previous research on Prolog implementation has been focused on inferring run-time information about variables and has used such information to specialize unification [Van Roy 1990], achieve multilevel clause indexing [Hickey and Mudambi 1989; Zhou 1993], optimize determinate procedures [Hickey and Mudambi 1989; Van Roy 1990; Zhou 1993], and remove redundant trailing checks and dereferences [Taylor 1989; Van Roy 1990]. Several variants of the WAM have been proposed. Tarau [1991] proposed a simplified WAM that is efficient for executing binary programs. The separate-stack WAM [Marien and Demoen 1990] splits the local stack of the WAM into two: one for holding environments and the other for

holding choice points. The VAM [Krall and Berger 1995], which aims at eliminating the argument-passing bottleneck of the WAM, combines the code for argument passing and that for head unification. Nevertheless, the inefficiencies of the WAM mentioned in the Introduction remain in these variants. In reality, no system based on these variants can significantly beat the fastest WAM-based systems without global program analysis.

9. FURTHER WORK

There are many alternatives to consider when deciding how arguments are passed and how frames are structured. The WAM lies at one extreme in the spectrum, where not only arguments but also the return address of every call are passed through registers. The ATOAM lies at the other extreme, in which anything is passed into the stack. An optimal choice might lie in the middle of the spectrum rather than at the extremes. Saumya K. Debray suggested to the author in 1993 a hybrid scheme in which some procedures have their arguments passed in the stack and in which other procedures have their arguments passed in argument registers. This decision should be made based on two factors, i.e., implementation environment and characteristics of programs. It is a further work to establish such a criterion.

Another further work, which is not unrelevant to the decision on argument passing, is to develop a good specialization algorithm that can translate Prolog programs into efficient canonical-form programs. Such an algorithm must be able to infer modes and detect determinacy of procedures. It must also be able to identify deep guards.

ACKNOWLEDGMENTS

Preliminary ideas and results of this article have been published in Zhou [1994]. I would like to thank Isao Nagasawa for his encouragement, Bart Demoen, Saumya K. Debray, Manuel Hermenegildo, and Paul Tarau for constructive comments, Mats Carlsson for suggesting to me to measure the numbers of memory references, and kindly answering my questions about how to get these data from SICStus-Prolog, and the anonymous referees for helpful comments on the presentation.

REFERENCES

Aho, A., Sethi, R., and Ullman, J. 1986. *Compilers – Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass.

Ait-Kaci, H. 1991. *Warren's Abstract Machine.* MIT Press, Cambridge, Mass.

Campbell, J. 1984. *Implementations of Prolog.* Ellis Horwood Chichester, U.K.

Carlsson, M. 1990. Design and implementation of an or-parallel prolog engine. Ph.D. thesis, S-16428, Swedish Institute of Computer Science, Kista, Sweden.

Clark, K. and Tarnlund, S. 1982. *Logic Programming.* Academic Press, New York.

Debray, S. 1988. Unfold/fold transformations and loop optimizations of logic programs. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation.* ACM Press, New York, 297–307.

Hickey, T. and Mudambi, S. 1989. Global compilation of prolog. *J. Logic Program. 7,* 3, 193–230.

Krall, A. and Berger, T. 1995. Incremental global compilation of prolog with the vienna abstract machine. In *Proceedings the 12th International Conference on Logic Programming.* MIT Press, Cambridge, Mass., 333–347.

MARIEN, A. AND DEMOEN, B. 1990. On the management of choice point and environment frames in the wam. In *Proceedings of the North American Conference on Logic Programming*. MIT Press, Cambridge, Mass., 1030–1047.

MEIER, M. 1991. Recursion vs. iteration in prolog. In *Proceedings of the 8th International Conference on Logic Programming*. MIT Press, Cambridge, Mass., 156–169.

TARAU, P. 1991. A simplified abstract machine for the execution of binary metaprograms. In *Proceedings of the Logic Programming Conference*. ICOT, Tokyo, Japan, 119–128.

TAYLOR, A. 1989. Removal of dereferencing and trailing in prolog compilation. In *Proceedings of the 6th International Conference on Logic Programming*. MIT Press, Cambridge, Mass., 48–60.

UEDA, K. AND CHIKAYAMA, T. 1990. Design of the kernel language for the parallel inference machine. *Comput. J. 33*, 494–500.

VAN ROY, P. 1990. Can logic programming executes as fast as imperative programming? Ph.D. thesis, Dept. of Computer Science, Univ. of California, Berkeley, Calif.

VAN ROY, P. 1994. 1983-1993: The wonder years of sequential prolog implementation. *J. Logic Program. 19*, 385–441.

VAN ROY, P. 1995. Aquarius benchmarks. *Available by anonymous ftp from gatekeeper.dec.com in pub/plan/prolog/AquariusBenchmarks.tar.Z*.

WARREN, D. 1977. Implementing prolog-compiling predicate logic programs. Ph.D. thesis, Dept. of Artificial Intelligence, Univ. of Edinburgh, Edinburgh, U.K.

WARREN, D. 1983. An abstract Prolog instruction set. Tech. Rep., SRI International, Menlo Park, Calif.

ZHOU, N. 1993. Global optimizations in a prolog compiler for the toam. *J. Logic Program. 15*, 265–294.

ZHOU, N. 1994. On the scheme of passing arguments in stack frames for prolog. In *Proceedings of the 11th International Conference on Logic Programming*. MIT Press, Cambridge, Mass., 159–174.

ZHOU, N. 1996. A novel implementation method for delay. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*. MIT Press, Cambridge, Mass., 97–111.

Received September 1995; revised February and May 1996; accepted June 1996