2007

TR-2007024: A Register-free Abstract Prolog Machine with Jumbo Instructions

Neng-Fa Zhou

A Register-free Abstract Prolog Machine with Jumbo Instructions

Neng-Fa Zhou

CUNY Brooklyn College & Graduate Center
zhou@sci.brooklyn.cuny.edu

Abstract. Almost all current Prolog systems are based on the Warren Abstract Machine (WAM), in which registers are used to pass procedure arguments and store temporary variables. In this paper, we present a stack machine for Prolog, named *TOAM Jr.*, which departs from the TOAM adopted in early versions of B-Prolog in that it employs no registers for arguments or temporary variables, and offers variable-size instructions for encoding procedure calls. TOAM Jr. is suitable for fast bytecode interpretation: the use of coarse-grained instructions results in more compact code and execution of fewer instructions than the use of fine-grained instructions; and the omission of registers facilitates interpretation of tagged operands and instruction merging. TOAM Jr. has been employed in B-Prolog since Version 7.0. Benchmarking shows that TOAM Jr. significantly enhances the speed of B-Prolog.

1 Introduction

Almost all current Prolog systems are based on the Warren Abstract Machine (WAM) [1, 16] and its variants [5, 13]. The WAM was originally designed for both software and hardware implementations. Procedure arguments are passed through argument registers so that hardware registers can be exploited in native compilers and hardware implementations. There are implementations that compile programs into C or native code through a WAM-like intermediate language [6, 14], but most popular and successful implementations only compile programs into WAM bytecode which is then executed by an emulator. In an emulator-based implementation, passing arguments through registers loses its advantage since registers are simulated. In [18], Zhou proposed an abstract machine, called Tree-Oriented Abstract Machine (TOAM), in which arguments are passed in an old-fashioned way through stack frames. This scheme of passing arguments through the stack, now common in virtual machines such as JVM [8] and MSIL [7], had been adopted by Prolog interpreters [9, 17]. In the TOAM, registers are retained to store temporary variables.

The WAM's instruction set is very fine-grained in the sense that roughly each symbol in the source program is mapped to one instruction. This fine-grainedness is a big obstacle to fast interpretation due to the high dispatching cost commonly seen in abstract machine emulators. Instruction specialization and merging are

two well-known techniques for reducing the overhead of interpretation for Prolog [2, 5, 11]. In [10], a tool is proposed to support this endeavor.

This paper proposes an abstract machine, called *TOAM Jr.*, which is based on the TOAM but departs from it in that no argument registers are provided. This means that even temporary variables are stored in stack frames. The omission of registers reduces the number of types of operands and thus makes the interpretation of tagged operands less expensive. In addition, the elimination of registers facilitates instruction specialization and merging. Another difference is that TOAM Jr. provides coarse-grained and variable-size instructions. A flattened call with no structured argument is normally encoded in only one instruction. In this way, a program requires fewer instructions to encode and execute, and hence the dispatching cost can be reduced significantly. TOAM Jr. has been employed in B-Prolog since Version 7.0. Benchmarking indicates that TOAM Jr. helps enhance the speed by 59% on Windows and 89% on Linux.

The reader is assumed to know the basics of Prolog implementation as described in [9]. The rest of the paper is structured as follows: Section 2 defines a language called *canonical-form Prolog* which can be compiled straightforwardly into TOAM Jr.. Section 3 introduces the TOAM memory architecture. Second 4 gives the definition of the base instruction set. Section 5 discusses instruction specialization and merging. Section 6 shows the experimental results, and Section 7 concludes the paper.

2 Canonical-form Prolog

A *canonical-form clause* takes the following form:

Head :- Guard ChoiceOperator Body

where *ChoiceOperator* is either ':' (called a *determinate choice operator*) or '?' (called a *nondeterminate choice operator*), and the terms in the clause are flattened to certain levels. The head is flattened such that all its arguments are distinct variables. The *Guard* consists of only flattened *inline tests*. A *matching test* $t_o = t_p$ in the guard is flattened such that t_o is a variable that has occurred before in the clause and t_p is a variable, a constant, or a structure or a cons whose arguments are distinct new variables. In particular, a matching test is called an *identity test* if both t_p and t_o are variables. The body consists of flattened unifications and calls. A unification in the body is flattened such that one operand is a variable and the other is a variable, a constant, a list with no compound elements, or a non-list structure with no compound arguments. A call is flattened such that no argument is compound.

Consider the `append` procedure in Prolog:

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]):-append(Xs,Ys,Zs).
```

This program is translated equivalently into the following canonical-form program with no assumption of modes of arguments:

```
append(Xs,Ys,Zs):-var(Xs) :
   append_aux(Xs,Ys,Zs).
append(Xs,Ys,Zs):-Xs=[] :
   Ys=Zs.
append(Xs,Ys,Zs):-Xs=[X|Xs1] :
   Zs=[X|Zs1],
   append(Xs1,Ys,Zs1).
```

```
append_aux(Xs,Ys,Zs):-true ?
   Xs=[],
   Ys=Zs.
append_aux(Xs,Ys,Zs):-true :
   Xs=[X|Xs1],
   Zs=[X|Zs1],
   append(Xs1,Ys,Zs1).
```

In general, modes of arguments, either declared by the user as required in Mercury [14] or obtained by a program analyzer [3], can help generate more compact and efficient canonical-form programs.

3 The Memory Architecture

Except for changes made to accommodate event handling and garbage collection as to be detailed below, the basic architecture is the same as the TOAM [18] employed in early versions of B-Prolog. This architecture has been extended to support constraint propagation [19] and tabling [20].

3.1 Code and data areas

TOAM Jr. uses all the stacks and data areas used by the WAM. 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 in the programs. There is one record for each symbol in the symbol table which stores such information as the *name*, *arity*, *type*, and *entry point* if the symbol is defined.

The *control* stack stores frames associated with procedure calls. Procedure call arguments are passed through stack frames and only one frame is used for each procedure call. Each time a procedure is invoked by a call, a frame is placed on top of the control stack unless the frame currently at the top can be reused. Frames for different types of procedures have different structures. For standard Prolog, a frame is either *determinate* or *nondeterminate*. A nondeterminate frame is also called a *choice point* frame.

The *heap* stores terms created during execution. As in the original design of the WAM, a block of memory is used for the control stack and the heap where the stack grows downwards and the heap grows upwards. The *trail* stack stores updates that must be undone upon backtracking. The use of a trail stack to support backtracking is the major difference between Prolog abstract machines and abstract machines for other languages such as Pascal, Lisp, and Java.

3.2 Term representation

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 cons.[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. A *free* variable is represented by a self-referencing pointer. A free variable stored on the stack is called a *stack variable* and a free variable stored on the heap is called a *heap variable*. 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 the *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 arguments of the structure. The address in a cons $[H|T]$ points to a block of two consecutive words in the heap where the first word stores the head H, and the second word stores the tail T. As in Lisp implementations, this special representation for lists benefits applications where lists are heavily used.

3.3 Registers

The following registers are used to represent the current machine status:

`P:`	Current program pointer
`TOP:`	Top of the control stack
`AR:`	Current frame
`H:`	Top of the heap
`T:`	Top of the trail stack
`B:`	Latest choice point frame
`HB:`	`H` slot of the latest choice point frame, `B->H`

The `HB` register, which also exists in the WAM, is an alias for `B->H`. It is used in checking whether or not a variable need to be trailed. When a free variable is bound, if it is a heap variable older than `HB` or a stack variable older than `B`, then it is trailed.

3.4 Stack frame structures

Frames for different types of procedures have different structures. A determinate frame has the following structure:

`A1..An`:	Arguments
`AR`:	Pointer to the parent frame
`CP`:	Continuation program pointer
`BTM`:	Bottom of the frame
`TOP`:	Top of the frame
`Y1..Ym`:	Local variables

Where `BTM` points to the bottom of the frame, i.e., the slot for the first argument, and `TOP` points to the top of the frame, i.e., the slot just next to that for the last

[1] In B-Prolog and also our implementation, floats are represented as special structures.

local variable. The TOP register points to the next available slot on the stack. The BTM slot was not in the original design [18]. This slot was introduced for support of garbage collection and event-driven action rules which require a new type of frames called *suspension frames* [19]. The AR register points to the AR slot of the current frame. Arguments and local variables are accessed through offsets with respect to the AR slot. An argument or a local variable is denoted as y(I) where I is the offset. Arguments have positive offsets and local variables have negative offsets.

It is the caller's job to place the arguments and fill in the AR and CP slots. The callee fills in the BTM and TOP slots.

A choice point frame contains, besides the slots in a determinate frame, four slots located between the TOP slot and local variables:

CPF: Backtracking program pointer
H: Top of the heap
T: Top of the trail
B: Parent choice point

The CPF slot stores the program pointer to continue with when the current branch fails. The slot H points to the top of the heap and T points to the top of the trail stack when the frame was allocated. When a variable is bound, it must be trailed if it is older than B or HB. When execution backtracks to the latest choice point frame, the bound variables trailed on the trail stack between T and B->T are set back to free, the machine status registers H and T are restored, and the program pointer register P is set to B->CPF.

The original TOAM presented in [18] had another type of frame, called *non-flat*, for determinate programs that have non-flat or deep guards. This frame was abandoned since it is difficult for the compiler to extract non-flat guards to take advantage of this offering.

3.5 Assertions

The following assertions must always hold during execution:

1. No heap cell can reference a stack slot.
2. No older stack slot can reference a younger stack slot and no older heap variable can reference a younger heap variable.
3. No slot in a frame can reference another slot in the same frame.

Assertions 1 and 2 are also enforced by the WAM. The third assertion is needed to make dereference of the arguments of a last call unnecessary when the current frame is reused. To enforce this assertion, when two terms being unified are stack variables, the unification procedure *globalizes* them by creating a new heap variable and letting both stack variables reference it.

4 The Base Instruction Set

Figure 1 gives TOAM Jr.'s base instruction set. An instruction with operands is denoted as a Prolog structure whose functor denotes the name and whose

Control:
- allocate_det(i_1,i_2)
- allocate_nondet(i_1,i_2)
- return
- fork(l)
- cut
- fail

Branch:
- jmpn_constant(y, l_{var}, l_{fail}, a)
- jmpn_struct($y, l_{var}, l_{fail}, f/n, y_1, \ldots, y_n$)
- switch_on_cons($y, l_{nil}, l_{var}, l_{fail}, y_1, y_2$)
- hash($y, i, (val_1, l_1), \ldots, (val_i, l_i), l_{var}, l_{fail}$)

Unify:
- unify_constant(y, a)
- unify_value(y_1, y_2)
- unify_struct($y, f/n, z_1, \ldots, z_n$)
- unify_list($y, i, z_1, \ldots, z_i, z_{i+1}$)

Move:
- move_struct($y, f/n, z_1, \ldots, z_n$)
- move_list($y, i, z_1, \ldots, z_i, z_{i+1}$)

Call:
- call($p/n, z_1, \ldots, z_n$)
- last_call($i, p/n, z_1, \ldots, z_n$)

Fig. 1. The TOAM Jr. base instruction set.

arguments denote the operands; and an instruction with no operand is denoted as an atom. An operand is either a frame slot y, an integer literal i, a label l, a constant a, a functor f/n, a procedure symbol p/n, or a *tagged* operand z. If an instruction carries two or more operands of the same type, subscripts are used to differentiate them. In the examples to be given below, the following notation is used for tagged operands: $v(i)$ denotes an uninitialized frame slot (i.e., a first-occurrence variable) with offset i, $u(i)$ denotes an initialized frame slot, and $c(a)$ denotes a constant a. An untagged frame slot is denoted as $y(i)$ where i is the offset. A singleton variable, also called a dummy variable in Prolog, is denoted as $v(0)$ if tagged and $y(0)$ if untagged. Notice that tags used for operands have nothing to do with those used for terms at runtime.

4.1 Control instructions

The first instruction in the compiled code of a procedure is an allocate instruction which takes two operands: the arity and the size of the frame, counting out the arguments. By the time an allocate instruction is executed, the arguments of the current call should have been placed on top of the stack and the `AR` and `CP` slots should have been set by the caller. An allocate instruction is responsible for fixing the size of the current frame and saving status registers if necessary. In the actual implementation, an allocate instruction also handles events and interruption signals if there are any. For the sake of simplicity, these operations are not included in the definition. Nevertheless, it is assumed that any procedure can be interrupted and preempted by event handlers. Therefore, a runtime test is needed to determine if the current frame can be deallocated or reused.

– The `allocate_det` instruction starts the code of a determinate procedure. It sets the `BTM` and `TOP` slots and updates the `TOP` register.

```
allocate_det(arity,size){
```

```
    AR->BTM = AR+arity;
    TOP = AR-size;
    AR->TOP = TOP;
}
```

– The `allocate_nondet` instruction starts the code of a nondeterminate procedure. In addition to fixing the size of the frame, it also saves the contents of the status registers into the frame.

```
allocate_nondet(arity,size){
    allocate_det(arity,size);
    AR->B = B;
    AR->H = H;
    AR->T = T;
    HB = H;
}
```

– The `return` instruction returns control to the caller, and deallocates the frame if the current frame is the topmost one that is not pointed to by the `B` register.

```
return(){
    P = AR->CP;
    if (B!=AR && AR->TOP==TOP) TOP = AR->BTM;
    AR = AR->AR;
}
```

In the original TOAM, when the current frame is deallocated, the top of the stack is set to be the top of the parent frame or the latest choice point frame, whichever is younger. Nevertheless, with event handling this becomes unsafe because the chain of active frames in the spaghetti stack are not in chronological order [19]. For this reason, the top of the stack is set to be the bottom of the current frame after the current frame is deallocated.

– The `fork` instruction resets the `CPF` slot of the current frame.

```
fork(addr){
    AR->CPF = addr;
}
```

– The `cut` instruction discards the alternative branches of the current frame, which must be a choice point frame.
– The `fail` instruction lets execution backtrack to the latest choice point frame.

Example The following shows a canonical-form program and its compiled code:

```
%    p:-true ? true.
%    p:-true : true.
```

```
p/0: allocate_nondet(0,8)
     fork l1
     return
 l1: cut
     return
```

Since the procedure is nondeterminate and there is no local variable, the allocated frame contains 8 slots reserved for saving the machine status.

4.2 Branch instructions

Unification calls in the guards of clauses in a procedure are encoded as *branch* instructions. Each branch instruction takes a label to go to on failure of the test l_{fail} and also a label to go to when the tested operand is a variable l_{var}. The `jmpn_struct` instruction fetches the arguments of the tested structure into designated frame slots when the test is successful. The `switch_on_cons` instruction moves control to the next instruction if the tested operand is a cons and to l_{nil} if it is an empty list. When the tested operand is a cons, the instruction also fetches the head and tail of the cons into the designated frame slots. The `hash` instruction determines the address of the next instruction based on the tested operand and a hash table.

The following shows an example.

```
%    p(F):-F=f(A),A=a  : true.
p/1: allocate_det(1,4)
     jmpn_struct(y(1),l_fail,l_fail,f/1,y(1))
     jmpn_constant(y(1),l_fail,l_fail,a)
     return
```

Notice that the argument slot with offset 1 allocated to the variable `F` is reused for `A`. None of the branch instructions carries tagged operands.

4.3 Unify instructions

Recall that in canonical-form Prolog every unification call in the bodies takes the form $V = T$ where V is a variable and T is either a variable, a constant, a list with no compound elements, or a compound term with no compound arguments. A unify instruction encodes a unification call where neither V nor T is a first-occurrence variable in the clause. For each type of T, there is a type of unify instruction. The `unify_constant` instruction is used if T is a constant; `unify_value` is used if T is a variable; `unify_list` is used if T is a list, and `unify_struct` is used if T is a structure. The `unify_list`$(y, i, z_1, \ldots, z_i, z_{i+1})$ encodes the list $[z_1, \ldots, z_i|z_{i+1}]$. The `unify_struct` and `unify_list` instructions have variable lengths and the operands for list elements or structure arguments are all tagged. In a `unify_struct` instruction, the number of tagged operands is determined by the functor f/n; and in a `unify_list` instruction the number is given as a separate operand.

A unify instruction for $V = T$ unifies the term referenced by V with T if V is not free. In WAM's terminology, the unification is said to be in `read` mode in this case. If V is a free variable, the instruction builds the term T and binds V to the term. This mode is called `write` in the WAM. Since a unification is encoded as only one instruction, there is no need to use a register for the mode.

Special care must be taken to ensure that no heap cell references a stack slot. The `unify_list` and `unify_struct` instructions must dereference a tagged operand if the operand is not a first-occurrence variable and globalize it if the dereferenced term is a stack variable. This dereference operation, however, is not as expensive as the general dereference operation since it stops walking the chain once the content of a stack slot is found to be a reference to the heap.

The following shows an example.

```
%    p(F):-true : F=f(L),L=[X,X,a].
p/1: allocate_det(1,4)
     unify_struct(y(1),f/1,v(1))
     unify_list(y(1),3,v(1),u(1),c(a),c([]))
     return
```

The argument slot with offset 1 allocated to the variable `F` is reused for `L` and later also for `X`. Since `L` is a first-occurrence variable, it is encoded as the tagged operand `v(1)`. The variable `X` occurs twice in `L=[X,X,a]`. The first occurrence is encoded as `v(1)` and the second one is encoded as `u(1)`. The tagged operand `c(a)` encodes the constant element `a` and the operand `c([])` encodes the empty tail of the list.

4.4 Move instructions

A move instruction is used to encode a unification $V = T$ where V is a first-occurrence variable in the clause. T is assumed to be a compound term. If T is a constant or a variable, the unification can be performed at compile time by substituting all occurrences of V for T in the clause. For this reason, only `move_struct` and `move_list` instructions are needed.

4.5 Call instructions

A `call` instruction encodes a non-last call in the body of a clause.

`call`$(p/n, z_1, \ldots, z_n)$`{`
    for each z_i $(i = 1, ..., n)$ do
        `*TOP-- =` value of z_i
    `parent_ar = AR;`
    `AR = TOP;`
    `AR->AR = parent_ar;`
    `AR->CP = P;`
    `P = entrypoint(`p/n`);`
`}`

After passing the arguments to the callee's frame, the instruction also sets the `AR` and `CP` slots of the frame, and lets the `AR` register point to the frame.

The value of each tagged operand z_i is computed as follows. If it is $v(k)$, then the value is the address of the frame slot with offset k (it is initialized to be a free variable) unless when k is 0, in which case the value is the content of the `TOP` register. If it is $u(k)$, then the value is the content of the frame slot with offset k. Otherwise, the value is z_i itself, which is a tagged constant.

A `last_call` instruction encodes the last call in the body of a determinate clause or a clause in a nondeterminate predicate that contains cuts. For a nondeterminate clause in a nondeterminate predicate that does not contain cuts, the last call is encoded as a `call` instruction followed by a `return` instruction. Unlike the `call` instruction which always allocates a new frame for the callee, the `last_call` instruction reuses the current frame if possible. The `last_call` instruction takes an integer, called *layout bit vector*, which tells what arguments are misplaced and hence need to be rearranged into proper slots in the callee's frame when the current frame is reused. There is a bit for each argument and the argument need to be rearranged if its bit is 1.[2]

```
last_call(layout,p/n, z_1, ..., z_n){
  if (AR->TOP==TOP && B! =AR){ /* reuse */
      for each argument z_i(i = 1, ..., n) do
          if (z_i is tagged u and its layout bit is 1)
              copy z_i to a temporary frame;
      move AR->AR and AR->CP if necessary;
      arg_ptr = AR->BTM+1;
      for each argument z_i(i = 1, ..., n) do
          if (z_i's layout bit is 1)
              *(arg_ptr-i) = the value z_i;
      AR = AR+(AR->BTM)-n;
      P = entrypoint(p/n);
  } else
      call(p/n, z_1, ..., z_n);
}
```

The following steps are taken to reuse the current frame: Firstly, all the misplaced arguments that are tagged `u` are copied out to a temporary frame. Because of the enforcement of assertion 3 (Subsetion 3.5), it is unnecessary to fully dereference stack slots but free variables in the frame must be globalized since otherwise unrelated arguments may be wrongly aliased. Constants and first-occurrence variables in the arguments are not touched in this step. Secondly, if the arity of the current frame is different from the arity of the last call, the `AR` and `CP` slots are moved. Thirdly, all misplaced arguments are moved into the frame for the callee. For `u`-tagged arguments, the values in the temporary frame are used

[2] In the actual implementation, an integer is used for the layout vector which has 28 bits for the value. If the last call has more than 28 arguments, then the last-call optimization will be abandoned.

instead of the old ones because the old values may have been overwritten by other values. Finally, the AR register is reset to be AR+(AR->BTM)-n.

For example,

```
% p(X,Y,Z):-S=f(X,Y),q(S),r(Z,Y,X,W).
p/3: allocate_det(3,5)
     move_struct(y(-1),f/2,u(3),u(2))  % S=f(X,Y)
     call(q/1,u(-1))                   % q(S)
     last_call(0b1011,r/4,u(1),u(2),u(3),v(0))
```

The binary literal '0b1101' is the layout bit vector for the last call which indicates that all the arguments except for the second one (Y) are misplaced. The variable W is a singleton variable in the clause and is encoded as v(0).

4.6 Storage allocation

Each variable is allocated a frame slot and is accessed through the offset of the slot. All singleton variables have offset 0. When an operand is tagged v, the offset must be tested. If the offset is 0, then it is known to be a singleton variable.

Frame slots allocated to variables are reclaimed as early as possible such that they can be reused for other variables. A variable is said to be *inactive* if it is not 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. Because of the existence of nondeterminate procedures, a variable may still be active even after its last occurrence. For example, consider the clause

```
a(U):-true : b(U,V),c(V,W),d(W).
```

The slot allocated to U can be reused after b(U,V) since the clause is determinate, but the slot allocated to V cannot be reused even after c(V,W) if b is nondeterminate.

5 Instruction Specialization and Merging

Instruction specialization and merging are two well-known important techniques used in abstract machine implementations. The omission of registers can make these techniques more effective. In this section, we discuss how some base instructions can be specialized and where instructions can be merged.

5.1 Instruction specialization

The variable length instructions that take tagged operands are targets for specialization. A variable length instruction is more expensive to interpret than a fixed length instruction since the emulator need to fetch the number of operands and iterate through the operands using a loop statement. A tagged operand is

more expensive to interpret than an untagged one because its interpretation involves the following overhead: (1) testing the tag; (2) untagging the operand if it is a variable tagged u or v; and (3) testing if the offset is 0 if the operand is an uninitialized variable tagged v. For an instruction of length up to n, $\Sigma_{i=1}^{n} 3^n$ specialized instructions can be created. Obviously, reckless introduction of specialized instructions will result in explosion of the emulator size and even performance degradation depending on the platform.

A specialized instruction carries the number and the types of the operands in its opcode. An instruction, named `unify_cons`(y, z_1, z_2), is introduced to replace `unify_list` that has two operands. For the `call` instruction, we introduce specialized instructions in the form of `call_`k`_u` ($k = 1, ..., 9$) which carries k initialized variables as operands in addition to the predicate symbol. We also introduce specialized versions of the `last_call` instruction that carry indexes of misplaced arguments explicitly as operands. In general, a specialized instruction for a last call takes the form `last_call_k`$(i_1, \ldots, i_k, p/n, z_1, \ldots, z_n)$ where the integers $i_1, \ldots, i_k$ are indexes of misplaced arguments that need to be rearranged. The currently implemented abstract machine has three specialized instructions ($k = 0, 1, 2$). Statistical data show that about 75% of last calls in the Aquarius benchmark suite [12] have 2 or fewer misplaced arguments.

5.2 Instruction merging

The dispatching cost is considered one of the biggest sources of overhead in abstract machine emulators. Even with fast dispatching techniques such as threaded code, the overhead cannot be neglected. A widely used technique in abstract machine implementations for reducing the overhead is called instruction merging, which amounts to combining several instructions into one. Although our instructions have large granularity, there are still opportunities for merging instructions.

It is often the case that a `switch_on_cons` or `fork` instruction is followed by a unify instruction and a unify instruction is followed by a `cut` instruction. So it makes sense to introduce merged instructions for these cases. We also introduce merged unify instructions for combining unify instructions and `return`. In addition, `cut` and `fail` are merged as well as `cut` and `return`.

When merging two instructions, we do not just combine the routines for the original instructions to create the routine for the merged instruction. Sometimes the merged instruction can be interpreted more efficiently. For example, consider the merged instruction `fork_unify_constant`(l, y, a), which combines `fork`(l) and `unify_constant`(y, a). The alternative program pointer `CPF` is set to be l if the unification succeeds. If the unification fails, however, execution can simply jump to l because the machine status has not changed since the creation of the frame. So the merged instruction not only saves the setting of `CPF` but also replaces expensive backtracking with cheap jumping.

The same idea can be applied to merged instructions of unify and cut. Consider the merged instruction `unify_constant_cut`(y, a). If y is a free variable, then `cut` can be performed before y is bound to a. In this way, unnecessary trailing of y can be avoided.

Table 1. Comparison on CPU times.

program	TOAM Jr.	TOAM	
		Linux	Windows
boyer	1	1.80	1.55
browse	1	1.89	1.63
chat_parser	1	1.73	1.42
crypt	1	1.62	1.46
fast_mu	1	2.08	1.59
flatten	1	2.45	2.28
meta_qsort	1	1.84	1.64
mu	1	2.05	1.70
poly_10	1	1.79	1.61
prover	1	1.92	1.73
qsort	1	1.82	1.44
queens_8	1	1.96	1.21
query	1	1.56	1.34
reducer	1	1.85	1.70
sendmore	1	1.96	1.52
simple_analyzer	1	2.33	1.88
tak	1	1.76	1.47
unify	1	2.08	1.73
zebra	1	1.32	1.25
Average	1	1.89	1.59

6 Experimental Results

TOAM Jr. has been employed in B-Prolog since Version 7.0. The implemented machine has 18 basic instructions and over 300 specialized and merged instructions for Prolog. Table 1 compares TOAM Jr. (B-Prolog Version 7.1) with TOAM (B-Prolog Version 6.9) on CPU time on a Windows XP machine (1.4 GHz Intel Celeron and 1G RAM) and a Linux machine (3.8GHz CPU and 2G RAM) using the Aquarius benchmark [12].

TOAM Jr. is on average 59% faster than TOAM on Windows and 89% faster on Linux. It is unclear why there are more gains on Linux than on Windows. The speedups are mainly attributed to specialized instructions, which would be more difficult to have if registers were existent. For a base instruction with n tagged variable operands, there are 2^n possible specialized instructions. If registers were allowed, then the number would be 4^n.

Certain speed-ups are attributed to new changes made to the implementtion. For example, the garbage collector in Version 7.1 requires no initialization of stack variables. This change alone contributes nearly 10% to the speed-up. The result on the famous benchmark, named `nreverse`, is not included because Version 7.1 adopts a specialized instruction for `append` which triples the speed on the benchmark.

As far as the Prolog part is concerned, the abstract machine of B-Prolog had not changed until Version 7.0. B-Prolog used to be one of the fastest Prolog systems [18], but during the last ten years its performance has been dragged down by the introduction of new features such as garbage collection, event-handling action rules, domain variables, and tabling. With TOAM Jr., B-Prolog is back to be one of the fastest Prolog systems. There are experimental results available elsewhere that compare B-Prolog with other Prolog systems (e.g. the logtalk benchmark results available at `http://logtalk.org/performance.html`).

7 Discussion

Compiling a high-level language into an abstract or virtual machine has become a popular implementation method, which has traditionally been adopted by compilers for Lisp, Prolog, and recently made popular by implementations of Java and Microsoft .NET. One of the biggest issues in designing an abstract machine concerns whether to have procedure arguments passed through registers or stack frames. Stack abstract machines are more common than register machines as exemplified by the Java Virtual Machine and Microsoft Intermediate Language.

One of the biggest advantages of passing procedure arguments through stack frames over through registers is that instructions for procedure calls need not to take destinations of arguments explicitly as operands. This leads to more compact bytecode and less interpretation overhead as well. For historical reasons, most Prolog systems are based on the WAM, which is a register machine, except for B-Prolog which is based on a stack machine called TOAM. Even TOAM retains registers for temporary variables.

For Prolog, a register machine such as the WAM does have its merits even when registers are simulated. Firstly, registers are represented as global variables in C and the addresses of the variables can be computed at load time rather than run time. Secondly, in some implementations a register never references a stack slot, and hence when building a compound term on the heap the emulator need not to dereference a component if it is stored in a register. In a highly specialized abstract machine such as the one adopted in Quintus Prolog [11], the registers an instruction manipulates can be encoded as part of the opcode rather than taken explicitly as operands. In this way, if the emulator is implemented in an assembly language to which hardware registers are directly available, abstract machine registers can be mapped to native registers.

Nevertheless, using registers has more cons than pros for Prolog emulators. Firstly, as mentioned above, instructions for procedural calls have to carry destination registers as operands which results in less compact code. Secondly, long-lived data stored in registers have to be saved in stack frames and loaded later when they are used. In Prolog, variables shared by multiple chunks[3] or multiple clauses are long-lived. According to the statistics reported by Zhou [18], passing

[3] A chunk consists of a non-inline call preceded by inline calls. The head of a clause belongs to the chunk of the first non-inline call in the body.

arguments through registers only pays off when accessing register is five times faster than accessing memory.[4] In an abstract machine emulator where registers are simulated, this is never the case even when the addresses of "registers" are computed at load time. Finally, registers make it more expensive to interpret tagged operands and harder to combine instructions because two more operand types, namely, uninitialized and initialized register variables, have to be considered. An alternative approach to facilitating instruction merging is to store all data in registers, as done in the BinWAM [15]. Nevertheless, this approach is hardly competitive because of the necessity to create continuations as first-class terms.

The need to rearrange arguments of last calls is considered a weakness of the TOAM [4]. TOAM Jr. inherits the memory architecture of the TOAM as well as this necessity. In the WAM, arguments need to be rearranged into proper registers for first calls. It is easy to find program patterns that make one machine arbitrarily worse than the other. Nevertheless, our investigation of a large number of programs shows that last calls have more to share with the heads than first calls in most tail recursive procedures. The code generation for last calls for TOAM Jr. is much simpler than that for the TOAM. The TOAM compiler adopts a sophisticated algorithm for generating code for last calls. For a last call, it builds a bipartite graph mapping the locations of arguments between the current and new frames, and optimizes the number of move instructions needed to rearrange the arguments.

Another design issue of abstract machines concerns the granularity of instructions. The WAM has a fine-grained instruction set in the sense that an instruction roughly encodes a symbol in the source program. The TOAM follows the WAM as far as granularity of instructions is concerned. There are Prolog machines that provide more fine-grained instructions such as explicit dereference instructions [13]. A fine-grained instruction set opens up more operations for optimization in a native compiler, but hinders fast interpretation due to a high dispatching cost. Instruction specialization and merging are two widely used techniques in abstract machine emulators for reducing the cost for Prolog [2, 5, 11]. In terms of granularity of instructions, TOAM Jr. resides between the WAM and a Prolog interpreter [9] where terms are interpreted without being flattened. The use of coarse-grained instructions reduces the code size and the number of executed instructions for programs, leading to a reduced dispatching cost.

Nevertheless, the interpretation of a variable number of tagged operands imposes certain overhead. After terms are flattened and registers are omitted, the number of possible operand types is reduced to three (constants, uninitialized variables and initialized variables). Because of the existence of only three operand types, the cost of interpreting tagged operands is much smaller than the dispatching cost. Also, the specialization of most frequently executed instructions makes interpretation of tagged operands unnecessary.

Currently our compiler basically performs no program analysis and uses no information on modes of arguments or determinacy of procedure calls. Mode

[4] That statistics did not take initialization of local stack variables into account.

information [3] is useful for translating a program into a more compact and efficient canonical form, and determinacy information is useful to help the storage allocator detect the life spans of variables and allow variables to share frame slots. It is a future task to introduce a program analyzer to infer and make use of these kinds of useful information.

8 Conclusion

This paper has presented an abstract machine for Prolog, named TOAM Jr., which shares the memory architecture as the TOAM implemented in early versions of B-Prolog but differs from it in: (1) no registers are used for arguments or temporary variables; and (2) coarse-grained instructions are used to encode flattened calls. While there are virtual machines designed for other languages such as JVM that provide no argument registers, no such a machine for Prolog has been experimented with before.

TOAM Jr. has the following advantages: (1) the omission of registers facilitates interpretation of tagged operands, and instruction specialization and merging; and (2) the use of coarse-grained instructions results in more compact code and execution of fewer instructions, leading to a reduced dispatching cost. Benchmarking shows that TOAM Jr. significantly improves the performance of B-Prolog.

References

1. Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.
2. Vitor S. Costa. Optimizing bytecode emulation for Prolog. In *Principles and Practice of Declarative Programming (PPDP)*, pages 261–277. LNCS 1702, 1999.
3. Saumya K. Debray and David Scott Warren. Automatic mode inference for logic programs. *J. Log. Program.*, 5(3):207–229, 1988.
4. Bart Demoen and Phuong-Lan Nguyen. On the impact of argument passing on the performance of the WAM and B-Prolog. Technical Report CW 300, Katholieke Universiteit Leuven, 2000.
5. Bart Demoen and Phuong-Lan Nguyen. So many WAM variations, so little time. In *CL2000: Proceedings of the 1st International Conference on Computational Logic*, volume 1861 of *LNAI*, pages 1240–1254. Springer Verlag, 2000.
6. Daniel Diaz and Philippe Codognet. Design and implementation of the GNU Prolog system. *Journal of Functional and Logic Programming*, 2001(1):1–29, 2001.
7. Serge Lidin. *Inside Microsoft .NET IL Assembler*. Microsoft, 2002.
8. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Publishing Co., second edition, 2000.
9. David Maier and David S. Warren. *Computing with Logic: Logic Programming with Prolog*. The Benjamin/Cummings Publishing Company, 1988.
10. José F. Morales, Manuel Carro, Germán Puebla, and Manuel V. Hermenegildo. A generator of efficient abstract machine implementations and its application to emulator minimization. In *ICLP, LNCS 3668*, pages 21–36, 2005.

11. Henrik Nässén, Mats Carlsson, and Konstantinos F. Sagonas. Instruction merging and specialization in the SICStus Prolog virtual machine. In *ACM PPDP*, pages 49–60, 2001.
12. Peter Van Roy. *Can Logic Programming Executes as Fast as Imperative Programming?* PhD thesis, Dept. of Computer Science, Univ. of California, Berkeley, Calif., December 1990.
13. Peter Van Roy. 1983–1993: The wonder years of sequential Prolog implementation. *Journal of Logic Programming*, 19,20:385–441, 1994.
14. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.
15. P. Tarau and M. Boyer. Elementary Logic Programs. In P. Deransart and J. Małuszyński, editors, *Proceedings of Programming Language Implementation and Logic Programming*, number 456 in Lecture Notes in Computer Science, pages 159–173. Springer, 1990.
16. David H. D. Warren. An abstract Prolog instruction set. Technical report, SRI International, 1983.
17. D.H.D. Warren. *Implementing Prolog-Compiling Predicate Logic Programs.* PhD thesis, Dept. of Artificial Intelligence, Univ. of Edinburgh, Edinburgh, U.K., 1977.
18. Neng-Fa Zhou. Parameter passing and control stack management in Prolog implementation revisited. *ACM Transactions on Programming Languages and Systems*, 18(6):752–779, 1996.
19. Neng-Fa Zhou. Programming finite-domain constraint propagators in action rules. *Theory and Practice of Logic Programming (TPLP)*, 6(5):483–508, 2006.
20. Neng-Fa Zhou, Taisuke Sato, and Yi-Dong Shen. Linear tabling strategies and optimizations. *Theory and Practice of Logic Programming (TPLP)*, to appear, 2007.