2003

TR-2003014: A High-Performance Abstract Machine for Prolog and Its Extensions

Neng-Fa Zhou

A High-Performance Abstract Machine for Prolog and its Extensions*

Neng-Fa Zhou
Department of Computer and Information Science
Brooklyn College & Graduate Center
The City University of New York
zhou@sci.brooklyn.cuny.edu

Abstract

This paper describes the design and the implementation of the TOAM (Tree-Oriented Abstract Machine) for Prolog and its extensions. The TOAM, as a Prolog machine, is based on the famous WAM model but differs from it in argument passing, stack management, and clause indexing. The original TOAM for Prolog was designed over ten years ago and the architecture was published in ACM TOPLAS in 1996 [26]. Since then, the machine has been extended to support several extensions of Prolog, including even-driven action rules, constraint solving, and tabling. The stack management scheme of the TOAM proved to be amenable to these extensions. The TOAM is employed in B-Prolog, a complete and efficient CLP system. The finite-domain constraint solver and the tabling system in B-Prolog represent the state-of-the-art implementations. This paper reviews the evolution of the TOAM as a Prolog machine, describes the changes needed to support the extensions, and reports the result of a comparison of B-Prolog and many other systems.

1 Introduction

The WAM [22, 2] has served as a good starting point for the design of many Prolog machines [17, 7]. Most of the machines are not drastically different from the original WAM. In the WAM, arguments of a call are passed through argument registers and two kinds of frames, namely *environments* and *choice points*, are used for storing information associated with calls. The original version of the TOAM [25] followed the same design, but provided a set of instructions for encoding *matching trees*, an intermediate representation of clauses.

In 1991, a question puzzled us for some time: "Is it really worth it to use argument registers in a software implementation of the WAM?". In a software implementation, registers are simulated by using memory. It is true that registers are faster than stack slots to access even in a software implementation since addresses of "registers" can be computed at load time. Nevertheless, since registers have to be saved and restored for nondeterminate predicates (predicates that have multiple applicable clauses) and non-binary clauses (clauses that have more than one call in the bodies), the advantage of fast access can be easily overshadowed by the traffic between registers and the stack.

We decided to try the alternative scheme of passing arguments through the stack. The result was a new version of TOAM [26]. As far as argument passing is concerned, the TOAM is more like the Pascal machine [1] and the JVM [12] than the WAM. A frame is used for each call regardless of whether the predicate is binary or non-binary/determinate or nondeterminate. In addition to the arguments, a frame holds a different set of information depending on the type of the predicate.

Since 1996, the TOAM has been extended to support several extensions of Prolog, including even-driven action rules (AR), constraint solving,

*Technical Report, CUNY CS , 2003.

and tabling. The stack management scheme of the TOAM proved to be amenable to these extensions.

The lack of facilities for programming *active* calls that can be reactive to the environment has been considered one of the weaknesses of logic programming. AR is an extension of Prolog designed to overcome this weakness. In AR, a call can be suspended when certain conditions are satisfied and can be activated by events. For such a call, a new type of frame, called *suspension frame*, is introduced into the TOAM. AR has been used to implement several constraint solvers in B-Prolog [27] including a very fast finite-domain constraint solver. If arguments were passed as in the WAM through registers, the registers would have to be saved on each suspension and restored on each activation of a call, and therefore the constraint solvers would not be as fast.

The TOAM was recently extended to support tabling, a technique that can get rid of infinite loops and redundant computations in the execution of recursive logic programs [28]. The main idea of tabling is to memorize the answers to calls and use the answers to resolve their variant descendents. Our tabling method differs from suspension-based systems such as XSB [18] in that it relies on iterative computation rather than suspension to compute fixpoints. For a tabled call, a new type of frame, called *tabled frame*, is introduced. Our early implementation was several times slower than XSB due to re-evaluation of tabled calls [28]. Our latest implementation, which incorporates several optimization techniques for avoiding redundant re-evaluation, competes favorably well with XSB in speed and outperforms XSB significantly in space efficiency.

This paper aims at providing a detailed description of the TOAM for Prolog and the extensions. The TOAM is the result of a research project that has lasted for over a decade. This paper reviews the evolution of the TOAM as a Prolog machine and describes for the first time the changes needed to support action rules, constraint solving and tabling. It also reports the result of a comparison of B-Prolog and many other systems.

2 The TOAM for Prolog

This section presents the architecture of the TOAM for running Prolog programs. Prolog is a dynamically typed language. Like in the WAM, data in the TOAM are all tagged. Free variables are represented as self-referencing pointers. A variable needs to be *dereferenced* to get its value.

2.1 Data Areas

The TOAM uses all the data areas used in the WAM. The *program area* stores the byte code instructions of loaded programs, the symbol table, and dynamic clauses created during program execution. The *heap* stores terms, mostly structural terms, created during execution. The register `H` points to the top of the heap. The *trail* stack stores those updates that must be undone upon backtracking. For each update, the address of the memory cell that was updated and the old content of cell are stored. This trailing scheme, called *value trailing*, is needed for primitives such as `setarg`. The register `T` points to the top of the trail stack. The *control* stack stores frames associated with predicate calls.

Unlike in the WAM where arguments are passed through argument registers, arguments in the TOAM are passed through stack frames and only one frame is used for each predicate call. Each time when a predicate 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 predicates have different structures. For standard Prolog, a frame is either *determinate* or *nondeterminate*. A nondeterminate frame is also called a *choice point*. The register `FP` points to the current frame and the register `B` points to the latest *choice point*.

A determinate frame has the following structure:

`A1..An`:	Arguments
`FP`:	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 local variable[1]. The TOP register points to the next available slot on the stack. The BTM slot was not in the original version [26]. This slot was introduced for garbage collection purpose. The FP register points to the FP slot of the current frame. Arguments and local variables are accessed through offsets with respect to the FP 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 FP, and CP slots. The callee fills in the BTM and TOP slots and initializes the local variables[2].

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 when the frame is allocated. As in the WAM, a new register, called HB, is used as an alias for B->H. When a variable is bound, it must be trailed if it is older than B or HB[3].

The version presented in [26] had another type of frame, called *non-flat*, for determinate programs that have non-flat guards. This frame was abandoned since it is difficult for the compiler to extract non-flat guards and take advantage of this offering.

There are no argument registers for passing arguments, but there are temporary registers for holding data between predicate invocations. Temporary registers are denoted as x(1), x(2), and so on. In addition, 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 unified, and the RW denotes the mode of unification, which is either *read* or *write*[4]

2.2 Instruction Set

We present the instruction set through examples. We consider compilation of *matching clauses* in the form of

```
H:-G : B.

H:-G ? B.
```

where H is called the *head*, G the *guard*, and B the *body*. One-directional matching rather than full unification is used to choose clauses for a call. A clause is applicable to a call C if C matches the head, i.e., the head becomes identical to C after a substitution is performed to it ($H\theta = C$), and the guard succeeds ($G\theta$). The operator $' :'$ indicates commitment: the remaining clauses will be disregarded when B fails. The operator $'?'$ indicates nondeterminate choice: the remaining clauses will be tried automatically when B fails. A predicate is said to be *determinate* if no operator $'?'$ is used; otherwise, it is called *nondeterminate*.

2.2.1 Compiling determinate programs

The first instruction for a determinate predicate is allocate_det and the last instruction for a clause is return_a. The allocate_det instruction takes two operands: the arity and the number of local variables. Unification can be *input* or *output*. Output unification is compiled into a *unify* instruction followed a sequence of *unify_arg* instructions. For example,

```
%    p(V):-true : V=f(a).
p/1: allocate_det 1,0
     unify_struct y(1),f/1
     unify_arg_atom a
     return_a
```

[1] It is a convention in the literature that the stack is assumed to grow downwards

[2] Variables need to be initialized for garbage collection purpose.

[3] A variable is called a *stack* variable if it resides on the stack and a *heap* variable if it resides on the heap. A stack variable is older than B if it resides in a frame that is older than the latest choice point or it is an argument in the latest choice point. A heap variable is older than HB if it was created before the latest choice point was pushed on to the stack.

[4] It is possible to let the S register take over the role of the RW register: The current unification is in write mode if S is null, and in read mode if S points to the heap [personal communication with Bart Demoen].

Input unification is compiled into a *conditional jump* instruction followed possibly by a sequence of *fetch* instructions. For example,

```
%    p(f(a)):-true : true.
p/1: allocate_det 1,0
     jmpn_eq_struct y(1),f/1,fail
     fetch_var x(1)
     jmpn_eq_atom x(1),a,fail
     return_a
```

Each call in the body of a clause is compiled into a sequence of *argument passing* and *build* instructions followed by a *call* instruction. For example,

```
%    p(V):-true : q(f(V)).
p/1: allocate_det 1,0
     pass_struct f/1
     build_value y(1)
     call q/1
     return_a
```

The `TOP` register points to the slot for the next argument. Each time after an argument is passed, `TOP` moves to the next slot.

2.2.2 Last call optimization

Last call optimization is an important optimization technique that allows the last call of a clause to reuse the current frame. A last call is compiled into a sequence of *move* instructions that rearranges the arguments into the correct order followed by an *execute* instruction. Arguments may overwrite the `FP` and `CP` slots. If this is the case, the two slots must be saved before the arguments are moved and later restored after the arguments are in order. For this purpose, the *save_fp_cp* and *restore_fp_cp* instructions are introduced. For example,

```
% p(U,V):-true : q(V).
p/2: allocate 2,0
     save_fp_cp
     move_value y(2),y(1)
     restore_fp_cp 1
     execute q/2
```

The *move_value* moves `V` to the slot allocated to `U`, and the *restore_fp_cp* restores `FP` to the place originally used for `V` and `CP` to the slot next to it.

The need to rearrange arguments of last calls to make last call optimization possible is considered a weakness of the TOAM [6]. In the WAM, arguments need to be rearranged into correct 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 predicates.

2.2.3 Compiling nondeterminate predicates

The first instruction for a nondeterminate predicate is `allocate_nondet` and last instruction for a nondeterminate clause is `return_b`. The `return_b` instruction returns control to the caller without reclaiming the current frame. The `fork` instruction sets the backtracking pointer `CPF`. The `cut` instruction discards the current choice point, i.e., resets the `B` register to the `B` slot of the latest choice point (`B->B`). The following example illustrates the use of these instructions.

```
% p(X):-true ? X=a.
% p(X):-true : X=b.
p/1: allocate_nondet 1,0
     fork C2
     unify_atom y(1),a
     return_b
 C2: cut
     unify_atom y(1),b
     return_a
```

2.2.4 Last call optimization revisited

For a clause, the current frame may not be the topmost one when the last call is encountered if some call before it has left choice points on the stack. If this is the case, the last call cannot reuse the current frame. It is undecidable at compile time whether a last call can reuse the current frame. For this reason, the compiler generates two streams of code for each last call: One reuses the current frame and the other uses a new frame. For example,

```
% p(X):-true : q(X),r(X).
```

```
p/1: allocate_nondet 1,0
     para_value y(1)
     call q/1
     jmpn_top_frame lab
     execute r/1     % reuse the frame
lab: para_value y(1) % use a new frame
     call r/1
     return_b
```

The `jmpn_top_frame L` moves control to `L` if the current frame is not the top-most one.

2.2.5 Compiling predicates into matching trees

The TOAM compiler accepts matching clauses and compiles them into a compact form called *matching trees*. In this way, shared tests among different clauses are merged and thus need to be evaluated only once for a call. The B-Prolog compiler and the library are made up of matching clauses only.

For a standard Prolog program, the compiler translates it into matching clauses quite naively. For each predicate, if there are two consecutive clauses whose heads have nonvariable terms in the same argument position, then the compiler specializes it into two: one taking care of the input case and the other taking care of the output case of the argument. Specialization is only done on one argument.

2.3 Performance Evaluation

Table 1 compares the speed of seven Prolog systems: B-Prolog 6.4 (BP), Bin-Prolog 9.47 (BIN), Eclipse 5.5 #46 (EP), Gnu-Prolog 1.2.16 (GP), Sicstus 3.10 (SP), Swi-Prolog 5.0.10, and XSB 2.5. All the systems are emulator based and all the emulators were compiled with the Microsoft VC++ compiler. Several other popular systems including Cao-Prolog, K-Prolog, and YAP were not compared because their emulators were not compiled with MVC[5]. Several other systems such as IF/Prolog were not compared because they were not available for evaluation. For those systems that offer more than one execution mode, such as Eclipse and SWI, the fast execution mode was selected[6]. The programs used in the comparison are from the Aquarius benchmark suite[7]. Each program was run at least 10 times (some were run 10000 times) and the average was taken. The comparison was conducted on a Windows XP machine with a 1.7G CPU and 760M RAM.

B-Prolog is comparable with Sicstus and is faster than the other WAM-based systems compared. In [5] and [7] the following advices are given on implementing a fast emulator: (1) a good discipline for writing C code, (2) selective use of GCC features, (3) a decent basic abstract machine code generator, (4) some instruction compression, and (5) some instruction specialization. One advice that is missing is *pursuing a new architecture while preserving the wisdom of the WAM*. The implementation effort put into B-Prolog is arguably incomparable with that put into some of the very sophisticated systems. The high performance of B-Prolog is attributed to a large extent to the TOAM architecture.

In [6] Demoen and Nguyen attribute the good performance of B-Prolog to instruction compression and two-stream dispatching. That observation contradicts our measurement. There are 220 instructions in the TOAM for compiling Prolog. This number is not large compared with some machines that have over 300 instructions. Our measurement shows that instruction compression only leads to up to 30% speed-ups. That means BP would be ranked second in the compared systems even without any instruction compression. Two-stream dispatching is a technique used in some Prolog emulators that avoids read/write mode checking through two interpreters, one for read mode and the other for write mode unification instructions. Early versions of B-Prolog adopted two interpreters. In version 6.3 and newer, this two-stream dispatching scheme was abandoned. Interestingly this change didn't lead to any slow-down.

[5] Cao and YAP were compiled using Cgwin GCC, which does not provide a correct timer for Windows. YAP has the reputation as being the fastest Prolog emulator with GCC [5, 9].

[6] GP has native code compilers for several platforms, but not for Windows yet.

[7] All the benchmarks used in this comparison and other comparisons are available from `probp.com/bench.tar.gz`.

Table 1: Speed comparison of Prolog systems (CPU times).

Program	BP	BIN	EP	GP	SP	SWI	XSB
boyer	1	1.97	1.25	4.22	0.85	3.57	3.43
browse	1	2.04	1.56	3.74	0.88	3.41	2.26
cparser	1	1.47	1.31	2.57	1.20	2.15	1.89
crypt	1	1.00	1.50	4.75	1.00	3.50	2.50
fast_mu	1	2.00	2.00	5.00	1.10	4.00	3.00
flatten	1	1.70	1.00	3.00	0.60	2.60	3.00
meta_qsort	1	2.11	0.80	3.00	0.60	2.10	2.21
mu	1	0.93	1.00	2.44	0.71	2.29	1.64
nreverse	1	1.50	0.75	4.25	0.50	5.25	2.25
poly_10	1	1.52	1.31	3.54	0.89	2.89	3.78
prover	1	1.41	1.27	2.82	0.99	2.39	2.25
qsort	1	1.43	1.29	4.00	0.71	3.87	3.00
queens_8	1	1.30	1.89	6.37	0.95	4.43	2.66
query	1	1.90	2.33	4.67	1.33	3.03	2.67
reducer	1	1.60	1.12	3.00	0.98	2.62	2.70
sendmore	1	3.11	3.14	9.37	2.05	5.96	4.00
sanalyzer	1	1.50	1.00	2.37	0.67	2.00	2.00
tak	1	3.18	2.93	10.24	1.46	6.61	5.88
zebra	1	0.86	0.78	0.88	0.62	1.33	1.05
<mean>	1	1.71	1.49	4.22	0.95	3.37	2.75

3 The TOAM for Action Rules and Constraint Solving

Prolog is a goal-driven programming language. Many applications including interactive graphical user interfaces, propagation-based constraint solvers, and agent-based systems require event-driven computing. The TOAM is extended to support *action rules* (AR), an event-driven programming language.

3.1 Action Rules

An *action rule* takes the following form:

```
Agent Condition {Event} => Action
```

where `Agent` represents a pattern for agents, `Condition` is a sequence of conditions on the agents, `Event` is a pattern for events that can activate the agents, and `Action` is a sequence of actions performed by the agents when they are activated.

All conditions in `Condition` must be in-line tests. The `Event` together with the enclosing braces is optional. If an action rule does not have any event pattern specified, then the rule is called a *commitment rule*. A set of built-in events is provided for programming constraint propagators and interactive graphical user interfaces. A user program can create and post its own events and define agents to handle them. A user-defined event takes the form of `event(X,T)` where `X` is a variable, called a *suspension variable*, that connects the event with its handling agents, and `T` is a Prolog term that contains the information to be transmitted to the agents. If the event poster does not have any information to be transmitted to the agents, then the second argument `T` can be omitted. The built-in `post(E)` posts an event.

When an agent is created, the system searches in its definition for a rule whose agent-pattern *matches* the agent and whose conditions are satisfied. This kind of rules is said to be *applicable* to the agent.

The rules in the definition are searched sequentially. If the rule found is a commitment rule in which no event pattern is specified, the actions will be executed. The agent will commit to the actions and a failure of the actions will lead to the failure of the agent. If the rule found is an action rule, the agent will be suspended until it is *activated* by an event. When the agent is activated, the conditions are tested *again*. If they are met, the actions will be executed. A failure of any action will cause the agent to fail. The agent does not vanish after the actions are executed, but instead turns to wait until it is activated again. So, besides the difference in event-handling, the action rule '`H,C,{E} => B`' is similar to the guarded clause '`H :- C | B, H`', which clones the agent after the action `B` is executed.

There is no primitive for killing agents explicitly. An agent vanishes only when a commitment rule is applied to it or it fails.

For example, the following defines an agent that echoes the messages sent to it by event posters.

```
echo_agent(X), {event(X,Message)} =>
   write(Message).
```

The following query,

```
?-echo_agent(Ping), post(event(Ping,ping))
```

creates an agent `echo_agent(Ping)` and then activates it by posting an event.

3.2 Constraint Solvers

AR extends various delay constructs such as freeze [4], when declaration [15], and delay clause [13] to allow for the descriptions of not only delay conditions on calls but also activating events and actions. All other delay constructs can be expressed easily in AR. For example, the `freeze` predicate [4] can be implemented as follows:

```
freeze(X,G), var(X), {ins(X)} => true.
freeze(X,G) => call(G).
```

where `ins(X)` is an event posted when `X` is instantiated. As long as `X` is a free variable, the call `freeze(X,G)` will be delayed. Only when `X` becomes a non-variable term, can the second rule be applied.

AR is a powerful language for implementing propagation-based constraint solvers. A domain variable is a suspension variable with some information attached to it. For each domain type, there is a set of built-in events. For example, for finite-domains the following set of events are provided:

`ins(X)`	`X` is instantiated
`bound(X)`	Either bound of `X` is updated
`dom(X,E)`	An internal element `E` is excluded

AR can be used to implement various kinds of propagation algorithms [27]. The following shows an example:

```
p(X,Y,C),var(X),var(Y),{dom(Y,Ey)} =>
   Ex is Ey+C,
   fd_exclude(X,Ex).
p(X,Y,C) => true.
```

The propagator maintains the arc consistency on `X` for the constraint `X=Y+C`. Whenever an internal element `Ey` is excluded from the domain of `Y`, it excludes `Ex`, the counterpart of `Ey`, from the domain of `X`. To have the arc consistency fully maintained, we need other propagators to take care of bound updates of the domain and the instantiation of the domain variable.

3.3 Suspension Frames

A new type of frame, called *suspension frame*, is introduced for calls that can be suspended and reactivated. A suspension frame extends a determinate frame to contain the following extra slots:

`SFP`:	Suspension frame pointer
`STATE`:	State of the call
`EVENT`:	Triggering event
`REEP`:	Re-entrance program pointer

The `SFP` slot connects this frame to the previous suspension frame, `STATE` indicates the current state of the call, `EVENT` stores the current triggering event, and `REEP` stores the re-entrance pointer to continue the execution with when the call is activated.

A call is in the `start` state when it is created and transits through the `inactive` and `active` states before it could reach the `exit` state. The state `inactive` means that the call is being suspended, and the state `active` means that the call has been activated.

There are three chains of frames on the stack: *active*, *choice point*, and *suspension* frame chains. All the active frames are connected by the `FP` slots, all the choice point frames are connected by the `B` slots, and all the suspension frames are connected by the `SFP` slots. With suspension frames, the chain of active frames may not be chronological. A frame may have its `FP` slot point to a frame on top of it. This kind of *spaghetti stack* has been used in the implementation of Lisp and Smalltalk [14].

3.4 Modification of the TOAM for Prolog

Some of the instructions of the TOAM for Prolog have to be redefined to support action rules and constraint solving, and the garbage collector needs to be modified to garbage collect useless frames on the stack.

The unification instructions are modified to take care of suspension variables. When a suspension variable `X` is bound to a term, the event `ins(X)` is be posted.

For the sake of efficiency, events are not checked immediately after they are posted but instead are postponed until before the execution of the next non-inline call. The allocate instructions are modified to check events. In case an event has been posted on a suspension variable, all the suspension frames of the variable are added into the active chain and the `EVENT` slots of the frames are

filled. If a suspension frame is already on the active chain, then a copy of the frame is made and the copy is added into the active chain. In this way, when there are multiple events posted that are all expected by an agent, the agent will be executed once for each of the events.

With suspension frames, the reclamation of stack frames becomes more complicated. Before when a call exits, the top of the stack could be reset to the top of the latest choice point or the parent frame, whichever is younger. With suspension, however, only the frame itself can be reclaimed. Since any call can be interrupted, run-time checking is needed to ensure the safety of the reclamation of stack frames. For example,

```
% p(X).
p/1: allocate_det 1,0
     jmpn_top_frame lab
     return_a
lab: return_b
```

Even for the unit clause `p(X)`, run-time checking is needed. The `jmpn_top_frame` checks whether the current frame is the top-most one. If so, the frame is reclaimed and control is returned; otherwise, only control is returned.

There may be frames that are younger than the latest choice point and that are connected by neither the active nor the suspension chain. These frames are useless and their space is claimed by the garbage collector.

3.5 Compiling Action Rules

Four new instructions are introduced for action rules. The following example illustrates their usage:

```
%  p(X,Y,C),var(X),var(Y),{dom(Y,Ey)} =>
%     Ex is Ey+C,
%     fd_exclude(X,Ex).
%  p(X,Y,C) => true.
p/3: allocate_susp 3,0
c1:  jmpn_var y(3),c2   % var(X)
     jmpn_var y(2),c2   % var(Y)
     neck_susp y(2),x(1),c1 % dom(Y,Ey)
     add x(1),y(1),x(1) % Ex is Ey+C
     para_value y(3)
     para_value x(1)
     call fd_exclude/2  % fd_exclude(X,Ex)
     return_susp
c2:  end_susp
     jmn_top_frame c21
     return_a
c21: return_b
```

The `allocate_susp` instruction allocates a suspension frame and initializes the state to `start`. The two operands tell the arity and the number of local variables, respectively. The `neck_susp` takes three operands: `y(2)` refers to the suspension variable, `x(1)` refers to the triggering event, and `c1` is the re-entrance pointer. This instruction behaves differently depending on the state of the frame. If the frame is in `start` state, it registers the frame into the suspension variable and returns control after changing the state to `inactive`. If the frame is `active`, then it fetches the triggering event from the `EVENT` slot and stores it in `x(1)`. The `return_susp` instruction returns control after changing the state of the current frame to `inactive`. The `end_susp` instruction changes the state to `exit`.

Action rules are compiled into matching trees such that shared tests among different rules do not need to be executed multiple times. This technique is useful for speeding-up constraint propagators that are defined by multiple rules with shared tests, such as a propagator for the Boolean constraint `and`.

3.6 Performance Evaluation

AR has been used to implement constraint solvers over several domains including terms, integers, finite-domains of ground terms, and finite sets. Table 2 compares the speed of B-Prolog and three other CLP(FD) systems: Eclipse (EP), Gnu-Prolog (GP), and Sicstus (SP). B-Prolog is the fastest among the compared systems.

The high performance of the finite-domain constraint solver of B-Prolog is attributed to the following three factors:

1. *Coarse granularity of propagators.* In GP and SP finite-domain constraints are compiled into indexicals [3, 9] while in BP constraints are

Table 2: Speed comparison of CLP(FD) systems (CPU times).

Program	BP	EP	GP	SP
alpha	1	9.79	1.21	4.31
bridge	1	2.17	0.51	2.54
cars	1	4.22	1.00	2.60
color	1	6.20	0.99	2.55
eq10	1	4.16	3.29	3.86
eq20	1	3.90	1.90	2.79
magic3	1	6.75	1.58	3.77
magic4	1	7.33	1.50	5.17
olympic	1	11.27	2.00	4.75
queens1	1	3.81	0.38	4.26
sendmoney	1	7.17	4.01	8.65
sudoku81	1	5.50	2.00	6.00
zebra	1	6.92	2.16	7.16
<mean>	1	6.09	1.73	4.49

compiled into propagators defined in action rules. One propagator in BP normally corresponds to a bunch of indexicals.

2. *Suppress of redundant activations of propagators.* Some events that cannot lead to the shrinking of any domains are ignored. For example, if multiple events of `bound(X)` are posted at the same time, then only one of them needs to be handled, and if `bound(X)` and `ins(X)` are posted at the same time, then the `bound(X)` event is ignored.

3. *Fast suspension and activation of propagators.* Propagators are stored as suspension frames on the stack, and can thus be suspended and activated without saving or restoring the arguments.

4 The TOAM for Tabling

Tabling for Prolog was first proposed by Tamaki and Sato in 1986 [21]. Since then, the research group at SUNY Stony Brook led by David Warren has worked intensively on its theory, implementation, and applications [10, 18, 23]. Recently, tabling has lured researchers from outside the research group[8, 11, 20, 28]. Tabling was first implemented in B-Prolog in 2000 [28], and was recently re-implemented to provide high-performance needed by a statistical learning system [24, 19].

Our new tabling system inherits the main idea from linear tabling [20, 28]: use iterative computation rather than suspension to compute fixpoints. A significant difference between linear tabling and OLDT [21, 23] lies in the handling of variant descendents of a call. In linear tabling, after a descendent consumes all the answers, it either fails or turns into a producer, producing answers by using the alternative clauses of the variant ancestor [28]. A call is called a *looping call* if a variant occurs as a descendent in its evaluation. The evaluation of top-most looping calls, i.e., calls that do not depend on their ancestors to be complete, must be iterated to ensure the completeness of evaluation. A call is said to be in the *iterative mode* when it is re-evaluated.

The new tabling system incorporates new control strategies and optimization techniques. A descendent variant call fails after it consumes all the current answers. This is different from our early implementation in which a descendent call steals the choice point of its variant ancestor. Another difference is in the timing of answer consumption. The early implementation adopts the eager consumption strategy: answers are consumed as early as possible. The new implementation adopts the lazy consumption strategy: for a top-most looping call, answers are consumed after all the answers have been produced. This strategy, which is similar to the *local scheduling* strategy implemented in XSB [10], allows for some optimization techniques for avoiding redundant re-computations.

4.1 Tabled Frames

A table is used to record calls and their answers. For each call and its variants, there is an entry in the table that stores the state of the call (complete or not) and an answer table for holding the answers generated for the call. Initially, the answer table is empty.

A new frame structure, called *tabled frame*, is introduced for tabled predicates. The frame for a tabled predicate contains the following three slots in addition to those slots stored in a choice point frame:

`CallTable`:	Pointer to the table entry
`CurrentAnswer`:	Pointer to the current answer
`Revised`:	Table revised or not

The `CallTable` points to the call table entry. The `CurrentAnswer` points to the answer that was just consumed. The next answer can be reached from this reference on backtracking. The field `Revised` tells whether any new answers have been added into the table since the frame was pushed on to the stack. This field will be propagated to the ancestor frames when this frame is deallocated. When execution backtracks to a top-most looping call, if the `Revised` field is set, then the call will be re-evaluated. A top-most looping call is complete if the field is unset after a round of evaluation. At that time, the call and all its dependent calls will be set to *complete*.

4.2 Instructions

Three new instructions are introduced for tabling, namely, `allocate_table`, `memo`, and `check_completion`. The following example illustrates their usage:

```
% p(X,Y):-p(X,Z),e(Z,Y).
% p(X,Y):-e(X,Y).
p/2: allocate_table 2,1
     fork c2
     para_value y(2)
     para_var y(-13)
     call p/2          % p(X,Z)
     para_value y(-13)
     para_value y(1)
     call e/2          % e(Z,Y)
     memo
c2:  fork c3
     para_value y(2)
     para_value y(1)
     call e/2          % e(X,Y)
     memo
c3:  check_completion p/2
```

Let A be a call to the predicate. The `allocate_table` instruction allocates a frame for A, and adds an entry to the table if A has not been registered yet. If A has an entry in the table whose state is *complete*, then A is resolved by using the answers in the table. If A is a pioneer of the current path, meaning that it is encountered for the first time, then control is moved to the next instruction. If A is a follower of some ancestor A_0, meaning that a loop has been encountered, then it is resolved by using the answers in the table, and is failed after the answers are exhausted.

The `memo` instruction is executed when an answer is found for A. If the answer A is already in the table, then just fail; otherwise fail after the answer is added into the table. The failure of `memo` postpones the consumption of answers until all paths have been explored.

The `check_completion` instruction is executed when A is being resolved by using program clauses and all the paths have been explored. If A has never occurred in a loop, then A's state can be set to *complete* and A can be failed after all the answers are consumed. If A is a top-most looping call, we check whether any new answers were produced during the last round of evaluation. If so, A is resolved again by using program clauses starting at `p/2`. Otherwise, if no new answer was produced, A is resolved by answers after being set to *complete*. Notice that a top-most looping call does not return any answers until it is complete. If A is a looping call but not a top-most one, A will be resolved by using answers after its state is set to *temporary complete*. A will be set to *complete* after its top-most looping call is complete.

4.3 Performance Evaluation

The new tabling system incorporates several optimization techniques for avoiding redundant recomputations. For example, for the transitive closure example, the clause `'p(X,Y):-e(X,Y)'` needs not be re-evaluated in the iterative mode and the joins of `p(X,Z)` and `e(Z,Y)` in the clause `'p(X,Y):-p(X,Z),e(Z,Y)'` must have one new answer involved. These optimization techniques significantly improve the speed of the tabling system.

Table 3 compares the speed and space performance of B-Prolog and XSB on a set of benchmarks: `tcl` and `tcr` are, respectively, the left-recursive and the right-recursive definitions of the transitive closure of a relation, `peep` and `read` are two program analyzers from [8], and `atr` is a

Table 3: Comparing Two Tabling Systems.

program	BP	XSB		
		Time	Stack space	Table space
tcl	1	1.48	1.14	1.15
tcr	1	1.04	13.83	1.01
read	1	1.83	15.52	0.16
peep	1	2.45	3.56	0.10
atr	1	1.97	37.46	3.08
<mean>	1	1.76	14.31	1.10

parser of a natural language defined by 800 grammar rules. B-Prolog is faster than XSB and consumes an order of magnitude less stack space[8] than XSB. For `peep` and `read`, BP consumes 5-10 times more table space than XSB because of the data structures used for tables and also because of the fact that BP tables automatically some extra predicates to avoid re-computation. In BP hash tables are used while in XSB tries [16] are used in the management of tables.

5 Conclusion

This paper has presented the TOAM, a high-performance abstract machine for Prolog and some of its extensions including event-driven programming, constraint solving, and tabling. One of the big differences between the TOAM and the WAM is in argument passing: arguments are passed through stack frames in the TOAM. Our decision to choose the old argument-passing scheme is a right one. The stack management scheme of the TOAM proved effective for Prolog and the extensions. For each of the extensions, a new type of frame is introduced. Since arguments are placed on the stack, it is unnecessary to save or restore the arguments of a call when the call is suspended or restored. It is unnecessary either to save or restore the arguments when a tabled call is re-evaluated.

B-Prolog, which adopts the TOAM, is comparable in speed with the fastest Prolog systems compared and is considerably more efficient than its peers as a CLP(FD) system and a tabling system. The implementation effort put into B-Prolog is arguably incomparable with that put into some of the very sophisticated systems. The high performance is attributed to a large extent to the right decision we made in the design of the TOAM.

[8] The total of local, global, choice point, trail, and SLG completion stack spaces for XSB, and the total of control, heap, and trail stack spaces for BP.

References

[1] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: principles, techniques, tools.* Addison-Wesley, 1986.

[2] Ait-Kaci, H. *Warren's Abstract Machine.* The MIT Press, Cambridge, MA, 1991.

[3] Carlsson, M., Ottosson, G., and Carlson, B. An open-ended finite domain constraint solver. In *Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'97)* (Sept. 1997), vol. 1292 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin.

[4] Colmerauer, A. Equations and inequations on finite and infinite trees. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS-84)* (Tokyo, Japan, Nov. 1984), ICOT, pp. 85–99.

[5] Costa, V. Optimizing bytecode emulation for Prolog. In *LNCS (PPDP)* (Sept. 1999), vol. 1551 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 261–277.

[6] Demoen, B., and Nguyen, P.-L. On the impact of argument passing on the performance of the wam and B-Prolog. Technical Report CW 300, Katholieke Universiteit Leuven.

[7] Demoen, B., and Nguyen, P.-L. So many WAM variations, so little time. *LNCS (CL) 1861* (2000), 1240–1254.

[8] Demoen, B., and Sagonas, K. CHAT: The copy-hybrid approach to tabling. *LNCS (PADL) 1551* (1999), 106–121.

[9] Diaz, D., and Codognet, P. GNU Prolog: Beyond compiling Prolog to C. *LNCS (PADL) 1753* (2000), 81–96.

[10] Freire, J., Swift, T., and Warren, D. S. Beyond depth-first: Improving tabled logic programs through alternative scheduling strategies. *LNCS (PLILP) 1140* (1996), 243–257.

[11] Guo, H.-F., and Gupta, G. A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. *LNCS (ICLP) 2237* (2001), 181–195.

[12] Lindholm, T., and Yellin, F. *The Java Virtual Machine Specification*, second ed. Addison-Wesley Publishing Co., Reading, MA, 2000.

[13] Meier, M. Better late than never. In *Implementations of Logic Programming Systems*, E. Tick and G. Succi, Eds. Kluwer Academic Publishers, 1994.

[14] Moss, J. E. B. Managing stack frames in Smalltalk. *ACM SIGPLAN Notices 22*, 7 (July 1987), 229–240.

[15] Naish, L. *Negation and Control in Prolog.* PhD thesis, University of Melbourne, 1985.

[16] Ramakrishnan, I., Rao, P., Sagonas, K., Swift, T., and Warren, D. Efficient access mechanisms for tabled logic programs. *J. Logic Programming 38* (1998), 31–54.

[17] Roy, P. V. 1983–1993: The wonder years of sequential Prolog implementation. *Journal of Logic Programming 19,20* (1994), 385–441.

[18] Sagonas, K., and Swift, T. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Transactions on Programming Languages and Systems 20*, 3 (1 May 1998), 586–634.

[19] Sato, T., and Zhou, N. A new perspective of PRISM. In *IJCAI Workshop on Learning Statistical Models from Relational Data* (2003), pp. 133–139.

[20] Shen, Y., Yuan, L., You, J., and Zhou, N. Linear tabulated resolution based on Prolog control strategy. *Theory and Practice of Logic Programming (TPLP) 1*, 1 (Feb. 2001), 71–103.

[21] Tamaki, H., and Sato, T. OLD resolution with tabulation. In *Proceedings of the Third International Conference on Logic Programming* (London, 1986), E. Shapiro, Ed., Lecture Notes in Computer Science, Springer-Verlag, pp. 84–98.

[22] Warren, D. H. D. An abstract Prolog instruction set. SRI International, Menlo Park, Calif., May 1983.

[23] Warren, D. S. Memoing for logic programs. *Comm. of the ACM, Special Section on Logic Programming 35*, 3 (Mar. 1992), 93.

[24] Zhou, N., and Sato, T. Efficient fixpoint computation in linear tabling. In *Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming* (2003), pp. 275–283.

[25] Zhou, N.-F. Global optimizations in a Prolog compiler for the TOAM. *Journal of Logic Programming 15*, 4 (Apr. 1993), 275–294.

[26] Zhou, N.-F. Parameter passing and control stack management in Prolog implementation revisited. *ACM Transactions on Programming Languages and Systems 18*, 6 (Nov. 1996), 752–779.

[27] Zhou, N.-F. Implementing constraint solvers in B-Prolog. In *IFIP World Congress, Intelligent Information Processing.* Kluwer Academic Publishers, 2002, pp. 249–260.

[28] Zhou, N.-F., Shen, Y.-D., Yuan, L.-Y., and You, J.-H. Implementation of a linear tabling mechanism. *LNCS (PADL) 1753* (2000), 109–123.