A Novel Implementation Method of Delay

Neng-Fa Zhou
Faculty of Computer Science and Systems Engineering
Kyushu Institute of Technology
680-4 Kawazu, Iizuka, Fukuoka 820, Japan
zhou@mse.kyutech.ac.jp

Abstract

The efficiency of delay depends to a large extent on the following four basic operations: *delay*, *wakeup*, *interrupt*, and *resume*. Traditional implementations of delay in the WAM are slow because three out of the four basic operations need to save or restore the argument registers. In this paper, we present a novel method for implementing delay in a Prolog machine called ATOAM. The main idea is to store delayed calls as frames, called *suspension frames*, on the control stack rather than as records on the heap. Since delayed calls, after being woken, can be executed directly by using their suspension frames, the four basic operations become very simple. This method had been predicated to cost a large amount of control stack space. However, with tail-recursion elimination, the control stack space requirement can be reduced dramatically. This method has been implemented in the B-Prolog system. For several benchmark programs where delay is used, the experimental results show that B-Prolog is significantly faster and sometimes consumes much less total space than SICStus, a WAM-based Prolog system.

1 Introduction

Delay has become a mechanism common to logic [3, 5, 12], functional logic [1, 8], constraint logic [9, 16], and concurrent logic [13, 14] programming languages. It relaxes the strict left-to-right computation rule adopted in Prolog and enables the execution of some predicate calls to be delayed until some variables in them are instantiated. Delay is an embedded mechanism in concurrent and constraint logic languages. In concurrent languages, a predicate call that does not carry enough information for the heads and guards of clauses to be executed will be delayed. In constraint languages, hard constraints are delayed until they are solvable.

Delay mechanism has been found useful in a wide range of applications. It can be used to describe perpetual processes and deal with infinite data structures [14]. It allows a generate-and-test program to be written as a test-and-generate program [12]. The latter is usually much faster than the former because tests can be performed as soon as they become testable and failure can be detected early. Delay mechanism also makes it possible to implement sound versions of negation and arithmetic [12]. In addition, it has been used to implement various constraint solvers [7, 10] and concurrent languages [6].

The efficiency of delay depends to a large extent on the following four basic operations: *delay*, *wakeup*, *interrupt*, and *resume*. The *delay* operation delays a predicate call when the call satisfies a delay condition. The *wakeup* operation is executed when some variables in a delayed call are bound. The delayed call becomes

a woken call and is added into the queue of calls that are ready for execution. A predicate is *interrupted* when the queue of woken calls is found not to be empty at certain points of execution. The execution of the interrupted predicate will be *resumed* after the woken calls are executed.

Carlsson's method [3] for implementing delay in the WAM has become the most popular one due to its simplicity. In order to retain the space saving technique of the WAM, it stores delayed calls as records on the heap. This idea originates in Boizumault's interpreter [2]. In this method, the four basic operations look like:

- **Delay:** The call to be delayed is created and stored on the heap.
- **Wakeup:** When some variables in a delayed call are bound, a wakeup event is signaled and the delayed call, which becomes a woken call now, is added into the queue of woken calls that are ready for execution.
- **Interrupt:** At the entry point of every predicate, if there is a wakeup event happening, the current predicate is interrupted and control is moved to the woken calls. Before that, an environment frame is pushed onto the control stack saving the argument registers such that the interrupted predicate can resume its execution afterwards. To execute each woken call, we have to move its arguments from the heap to appropriate argument registers.
- **Resume:** After the execution of woken calls finish, the interrupted predicate will resume its execution by restoring the argument registers from the environment frame.

This method is slow because three out of the four operations (except for wakeup) have to save or restore the argument registers.

In this paper, we present a novel method for implementing delay in the ATOAM [17], an abstract machine for Prolog that differs from the WAM mainly in: (1) arguments are passed directly into stack frames; (2) only one frame is used for each predicate call; (3) predicates are translated into matching trees if possible and indexed on all input arguments. The main idea of the method is to store delayed calls as frames, called *suspension frames*, on the control stack rather than as records on the heap. Since delayed calls, after being woken, can be executed directly by using their suspension frames, none of the four operations needs to save or restore the argument registers.

Storing delayed calls on the control stack would suppress the space-saving technique for the control stack in the WAM. In concrete, the control stack below suspension frames must be frozen and the frames in the frozen area cannot be released even if they become useless. However, we found that frames of tail-recursive predicates can be reused as long as there are no choice points lying on top of them. Although the tail-recursion elimination technique cannot thoroughly solve the possible control stack space explosion problem, it can reduce dramatically the control stack space requirement for well-written programs.

The method has been implemented in the B-Prolog system, an emulator-based system. The experimental results are surprisingly good. For several benchmark programs where delay is used, the emulated code of B-Prolog is significantly faster than the emulated code and can sometimes beat the native code of SICStus. Furthermore, B-Prolog sometimes consumes much less total space than SICStus.

In the next section, we define the delay mechanism. In Section 3, we survey the main features of the ATOAM. In Section 4, we describe the method for implementing

delay in the ATOAM. In Section 5, we experimentally compare the performance of B-Prolog with that of SICStus. In Section 6, we conclude the paper.

2 Delay Mechanism

Different constructs, such as *freeze* in Prolog-II [5], *when* declaration in NU-Prolog [12], *block* declaration in SICStus [15], *delay clause* in Eclipse [11], *bind hook* in ESP [4], have been proposed for describing delay. We consider here delay clause [11] which is powerful enough for implementing other constructs.

2.1 Delay Clause

Each predicate is defined by optionally several *delay clauses* followed by a sequence of definite clauses. A delay clause takes the following form:

```
delay Head if Condition.
```

where `Head` is an atomic formula and `Condition` is defined recursively as follows:

```
Condition ::= var(X) |
              X\==Y  |
              Condition,Conditon |
              Condition;Condition.
```

The delay clause can be read as "for any call to the predicate of `Head`, if it matches `Head`, *i.e.,* the call is an instance of `Head`, and `Condition` is satisfied, then delay the call." Notice that, because matching rather than full unification is used here, the call will not be updated while delay clauses are executed. Notice also that a call would not be delayed if it does not match the head of any delay clause. For example, for the delay clause,

```
delay p(f(X,X)) if var(X).
```

the call `p(f(X,X))` will be delayed, whereas the call `p(f(X,Y))` will not be delayed because `p(f(X,Y))` does not match the head.

We assume that predicates containing delay clauses are all determinate for which no choice point is necessary. This seems to be a restriction, but actually not because any predicate can be translated into another one that meets this requirement. For a predicate `p(T1,...,Tn)` with delay clauses, if it is nondeterminate, then the predicate can be translated into:

```
delay clauses.
p(T1,...,Tn):-p_aux(T1,...,Tn).
```

where the predicate `p_aux(T1,...,Tn)` is defined by the original clauses defining `p(T1,...,Tn)`.

Variables in a call are called *suspending variables* if the call is delayed because they are uninstantiated. A delayed call will be woken up when one of its suspending variables is bound to another term. Since one suspending variable may participate in multiple delayed calls, there may be multiple woken calls ready for execution at a time. These calls form a queue. At the entry and exit points of every predicate, the queue of woken calls is checked. If it is not empty, then the current predicate is interrupted and control is moved to the woken calls. After the woken calls finish their execution, the interrupted predicate will resume its execution.

2.2 Implementing `freeze` by Using Delay Clause

To show the power of delay clause, we consider as an example how to implement *freeze* by using delay clause. The primitive `freeze(X,p(T1,...,Tn))` delays the call `p(T1,...,Tn)` until `X` is instantiated. The simplest way of implementing `freeze` is to define it as:

```
delay freeze(X,G) if var(X).
freeze(X,G):-call(G).
```

The problem with this definition is that delayed calls must be created on the heap. To deal with it, we propose to compile calls of `freeze` as follows: First replace `freeze(X,p(T1,...,Tn))` with `p_aux(T1,...,Tn)` where `p_aux` is a new predicate symbol, and then define the predicate `p_aux` as follows:

```
delay p_aux(T1,...,Tn) if var(X).
p_aux(T1,...,Tn):-p(T1,...,Tn).
```

This process is repeated until no call to `freeze` exists. For example, the call

```
freeze(X,freeze(Y,p(X,Y)))
```

which will be delayed if either `X` or `Y` is uninstantiated, is replaced with a call `p1(X,Y)` which is defined as follows:

```
delay p1(X,Y) if var(X).
p1(X,Y):-p2(X,Y).

delay p2(X,Y) if var(Y).
p2(X,Y):-p(X,Y).
```

By folding the calls to `p2`, we get,

```
delay p1(X,Y) if var(X).
delay p1(X,Y) if var(Y).
p1(X,Y):-p(X,Y).
```

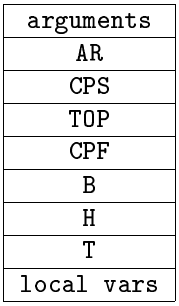
3 The Prolog Abstract Machine ATOAM

This section briefly surveys the architecture of the ATOAM. The detailed description of the architecture and its advantages over the WAM can be found in [17].

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 predicate calls. Each time when a predicate is invoked by a call, a frame is placed on top of the control stack unless the frame currently on top of the control stack can be reused.

Predicates are classified into flat, nonflat, and nondeterminate ones [1]. The structures of frames differ for different types of predicates. The frame for nondeterminate predicates contains the following fields:

[1] Each clause is divided into two parts: a condition on the calls to which the clause is applicable and all other remaining calls. A predicate is said to be *determinate* if: for each clause, if it is applicable to a call then the remaining clauses need not be tried for the call; and *nondeterminate* otherwise. A determinate predicate is said to be *flat* if the condition of any clause in it consists of only inline tests; and *nonflat* otherwise.



The AR slot points to the parent frame; the CPS slot stores the continuation program point to go to on success; the TOP slot points to the top of the control stack; the B slot points to the frame of the latest predecessor of the call that has alternative program points to be tried; the CPF slot stores the alternative program point to go to after a branch in the predicate fails; the H slot points to the top of the heap and the T slot points to the top of the trail stack. The frames for nonflat predicates do not contain the `H` and `T` slots and the frames for flat predicates do not contain `B`, `CPF`, `H` and `T`.

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 predicate *deep choice point*, and the frame of a nonflat predicate *shallow choice point*.

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, DB and SB are compared. If DB is younger than SB, then deep backtracking is invoked; otherwise, shallow backtracking is invoked. DB and SB never have the same contents.

The HB register is used in checking whether or not a variable needs to be trailed when the variable is bound. 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 must be trailed.

4 Delay in the ATOAM

In this section, we show how to implement the delay mechanism in the ATOAM. We first show how suspending variables and delayed calls are represented, then define the four basic operations, and finally describe the changes in memory management.

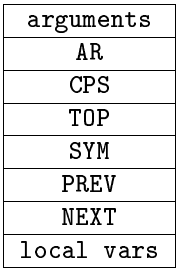
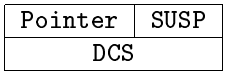


Figure 1: Suspension frame.

4.1 Suspending Variables and Delayed Calls

To implement the delay mechanism in the ATOAM, we introduce a new data type, called SUSP, for representing suspending variables. Each suspending variable is stored on the heap as a record with two fields:



The first field is a self-pointing pointer tagged with SUSP, and the second field DCS points to a list of delayed calls related to the variable. Each delayed call is simply represented as a pointer to its suspension frame on the control stack. Binding a suspending variable to a term means destructively updating the first field and letting it point to the term. If the suspending variable is older than HB, then the pair of the address and the old value is pushed onto the trail stack.

A suspension frame looks like a frame for a determinate predicate but contains three new slots (see Figure 1). All the suspension frames are connected by a doubly linked list. The Prev slot points to the previous, and the Next slot points to the next suspension frame. The slot SYM stores the predicate symbol with which the predicate can be reentered. A new register, called SF, is introduced that points to the latest suspension frame.

A suspension frame may be in one of the following states: *start*, *sleep*, *woken*, and *exit*. These states are illustrated in Figure 2. After a suspension frame is created, it enters the *start* state immediately. At this time, the three slots SYM, PREV and NEXT have not yet been filled in. When no delay clause succeeds, the state changes into *exit*. When some delay clause succeeds, the frame is connected into the list of suspension frames (the three slots SYM, PREV, and NEXT are filled in) and the state changes to *sleep*. In *sleep* state, the slots AR and CPS lose their meaning. When a suspending variable in the suspension frame is bound in unification, the state changes from *sleep* to *woken*. In this state, the three slots SYM, PREV and NEXT remain unchanged, and the frame is connected to the list of active calls being executed (AR and CPS are filled in). Executing the woken call may result in two cases: (1) some delay clause succeeds and the woken call is redelayed; and (2) no delay clause succeeds and the state changes to *exit*.

Notice that the data structures for suspending variables and suspension frames facilitate the basic operations. It is easy to find all the delayed calls related directly to a given suspending variable. With the SYM and arguments in a suspension frame, it is possible to construct the delayed call as a structure and print it out. In addition, it is easy to check termination: some calls are delayed if SF is not empty.

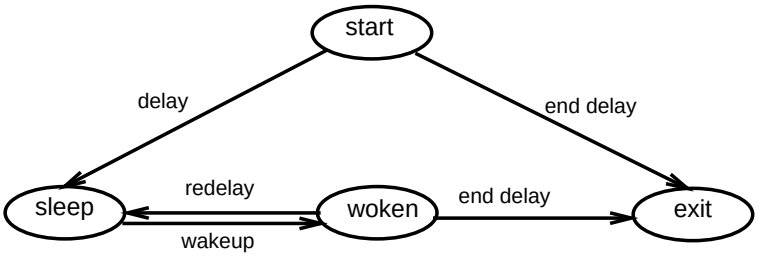


Figure 2: Diagram of state transition.

4.2 Delay

The compiled code for a predicate `p/n` with a delay clause is as follows:

```
p/n: allocate_susp N
     conditional jump instructions
     susp_var instructions
     delay p/n
L:   end_delay
     code for other clauses
```

The code for the delay clause consists of one or more conditional jump instructions followed by several `susp_var` instructions. It will move control to `L` if the delay condition is not satisfied. Each `susp_var` instruction corresponds to a call `var(X)` in the body of the delay clause. The `allocate_susp`, `susp_var`, `delay` and `end_delay` are newly introduced instructions, which are defined as follows:

- `allocate_susp N`
  This instruction moves the control stack pointer `TOP` to keep enough space for the predicate. It also fills in the `TOP` slot and initializes the state of the frame to be *start*.

- `susp_var Z`
  This instruction lets `Z` be a suspending variable and the current call be a delayed call in the `DCS` of the variable.

- `delay p/n`
  This instruction is executed after a delay clause succeeds. It changes the state of the current frame into *sleep*. When the current frame is in *start* state, then this instruction fills in the `SYM` slot with `p/n`, connects the current frame into the list of suspension frames, and returns control to the caller. Otherwise, if the current frame is in *woken* state, then control is returned to the caller with nothing done to the suspension frame. These two cases are illustrated by the two arcs in Figure 2 labeled with *delay* and *redelay* respectively.

- `end_delay`
  This instruction is executed after all the delay clauses in a predicate fail. It turns the state of the current frame from either *start* or *woken* to *exit*. When the current frame is *woken*, then it has to be disconnected from the list of suspension frames.

Consider, for example, the following predicate:

```
delay p(X,Y) if var(X);var(Y).
p(X,Y):-X>Y.
```

The compiled code is:

```
p/2: allocate_susp 6
     jmpn_var A1,L1  % code for delay clause
     susp_var A1
     delay p/2
L1:  jmpn_var A2,L2
     susp_var A2
     delay p/2
L2:  end_delay
     jmpn_gt A1,A2,F % F is the label for failure handler
     return
```

When either of the arguments is a variable, the call will be delayed.

4.3 Wakeup Event

When a suspending variable is bound, a wakeup event will be signaled and the suspending variable becomes a trigger variable. We introduce a new register, called `TRIGGER`, that points to the list of trigger variables.

The unification procedure `unify(T1,T2)` is modified as follows to take suspending variables into account:

1. if `T1` is a Prolog variable and `T2` is a suspending variable, then bind `T1` to `T2`[2].
2. if one of the two arguments is non-variable and the other is a suspending variable, then bind the suspending variable to the non-variable term.
3. if `T1` and `T2` both are suspending variables, then create a new suspending variable whose `DCS` points to the concatenation of the two lists associated with `T1` and `T2`, and bind `T1` and `T2` to the new variable.

4.4 Wakeup Event Handler

Figure 3 shows the pseudo-code of the wakeup event handler. Recall that delayed calls in `DCS` of each trigger variable are actually pointers to their suspension frames on the control stack. The event handler connects the woken calls and the frame of the interrupted predicate into a chain and execute the calls in the chain one by one. The program point to execute for each woken call `Frame` is computed as follows:

```
P = entrypoint(Frame->SYM)+sizeof(allocate_susp);
```

Since the frame used to execute the woken call already lies on the control stack, the `allocate_susp` instruction must be skipped. After the woken calls are connected into the chain of calls, there is no difference between them and usual predicate calls. Thus, signaling another wakeup event during the execution of woken calls is permitted.

[2]Exercise: `T2` should to assigned to `T1` after the tag `SUSP` is removed. Why?

```
Wakeup_Event_Handler:
   for each trigger variable in TRIGGER do
      remove the variable from TRIGGER;
      for each frame Frame in DCS of the variable do
         if Frame is in sleep state then
            Frame->AR = AR;
            Frame->CPS = P;
            AR = Frame;
            P = entrypoint(AR->SYM)+sizeof(allocate_susp);
            change state of Frame to woken;
         endif
      enddo
   enddo
```

Figure 3: Wakeup event handler.

4.4.1 How are woken calls scheduled?

The chain of calls is constructed on a first-come-first-served basis. The earlier a suspending variable became a trigger variable, the earlier its woken calls will be executed. For each trigger variable, the earlier a woken call entered the DCS, the earlier it will be executed.

4.4.2 How to resume the interrupted predicate?

Nothing special is necessary to resume the execution of the interrupted predicate. After the execution of the woken calls, control will be moved smoothly to the interrupted predicate.

4.4.3 When to invoke the wakeup event handler?

At the entry and exit points of any predicate, the list of trigger variables is checked. If it is not empty, then the current predicate is interrupted and control is moved to the woken calls of the trigger variables. In general, no choice point can be created between the point where a wakeup event is signaled and the point where the event is handled. To obey this principle, we must take care to ensure that the wakeup event handler is invoked before a choice point is created at the entry point of a nondeterminate predicate.

There is an *allocate_nondet* instruction at the entry point of a nondeterminate predicate that allocates a choice point. If the wakeup event handler is invoked after a choice point is created, then the behavior of the program may become strange. The following example illustrates this situation.

```
go:-freeze(X,p(X)),X=[Y],q(X),write(X).

p(X):-X=[f(a)].

q(X):-fail.
q(X).
```

The correct output of this program is `[f(a)]`. However, if the wakeup event signaled by `X=[Y]` is handled after the choice point for q/1 is created, then the output would be `[Y]` where `Y` is an unbound variable. The reason for this abnormal phenomenon is that the unification `X=[f(a)]`, i.e. `[Y]=[f(a)]`, is done after the choice point of q/1 is created, and the value bound to `Y` is lost upon backtracking.

To deal with this problem, we split the *allocate_nondet* instruction into two: *allocate_flat* and *flat_to_nondet*. The *allocate_flat* instruction, which already exists in the ATOAM, allocates a flat frame and checks the wakeup event signal. The *flat_to_nondet* instruction changes the frame from flat to nondeterminate by saving the status registers. Notice that some woken calls may be executed between these two instructions and some choice points or suspension frames may be left on the control stack. Thus, when *flat_to_nondet* is executed, the current frame may not be the top-most one. In this case, the *flat_to_nondet* instruction has to copy the current frame to the top of the control stack.

4.4.4 How to skip woken calls that have already been executed?

Since a call may be delayed due to multiple suspending variables, when it is woken, it might have already been executed. Consider, for example, the inequality constraint on finite-domain variables $X \neq Y$. The propagation procedure for the constraint can be defined as follows:

```
delay inequality(X,Y) if var(X),var(Y).
inequality(X,Y):-nonvar(X),nonvar(Y),!,X\==Y.
inequality(X,Y):-nonvar(X),!,exclude_value(Y,X).
inequality(X,Y):-exclude_value(X,Y).
```

The constraint `inequality(X,Y)` is delayed when both `X` and `Y` are variables. Suppose initially `X` and `Y` are variables. After `X` is assigned a value, the constraint is woken and `X` is excluded from `Y`'s domain by `exclude_value(Y,X)`. When, however, `Y` is assigned a value afterwards, the constraint need not be executed again. To handle this situation, we check to see whether or not the frame for a woken call is in sleep state. If not, we skip the woken call.

4.5 Memory Management

The management of the trail and control stacks are changed to support delay mechanism. We now describe these changes.

4.5.1 Trail Stack

Like in the WAM, the trail stack in the original ATOAM contains addresses of variables that need be reset to unbound upon backtracking. This address-only trailing scheme is not adequate now because updates to the list of suspension frames `SF` and the `DCS` field of suspending variables can be undone only when the old values of modified cells are available. For this reason, the trail stack of the ATOAM is made to contain address-value pairs.

Several trailing schemes have been proposed to trail destructive updates. In ECRC Prolog [11], three trail stacks are used. In some implementations of constraint languages, the trail stack is designed to contain tagged elements where the tag of each element determines whether the element is only an address or a pair of an address and a value. We adopt the address-value trailing scheme because this

scheme is simple and imposes only minor overhead (2 to 3 percent) to standard Prolog programs.

4.5.2 Control Stack

In the original ATOAM, each time when a predicate is invoked by a call, a frame is placed on top of the control stack unless the frame currently on top of the control stack can be reused. Consider the following clause

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

If there is no clause in the predicate remaining to be executed after the clause fails and `foo(X)` never leave choice points behind after its execution, then the frame for the head predicate is determined to lie at the top of the control stack and thus can be reused by `q(Xs)`. However, with delay, things are not so simple. Some woken calls may be executed at the entry point and `foo(X)` may create some suspension frames on the control stack. For this reason, it has to be checked at run time whether the current frame is at the top. If not, a new frame has to be allocated for the tail call `q(Xs)`.

The rule for frame reusage can be relaxed to enable non-top-most frames to be reused. For tail-recursive predicates, if the current frame is a determinate one and there is no choice point younger than it, then tail-recursive calls can reuse the frame. This means that frames of tail-recursive predicates can be reused as long as there are no choice points lying on top of them. The revised rule for frame reusage is very important to limit the space explosion of the control stack. Consider the coroutines:

```
consume(X),produce(X).
```

where `consume` and `produce` are defined tail-recursively, and `consume` is delayed when no data is available and is invoked as soon as there are data available. If both predicates are determinate, then the program only requires constant stack space.

4.6 Summary

In our method, the four basic operations for delay look like:

- **Delay:** The frame of the current call is connected to the list of suspension frames if the call is delayed first time.
- **Wakeup:** The suspending variable that was bound is added into the list of trigger variables `TRIGGER`.
- **Interrupt:** The frames for the woken calls are connected into a chain such that the woken calls can be executed one by one.
- **Resume:** After the list of woken calls is executed, control will be moved smoothly to the interrupted predicate.

Except when the interrupted predicate is nondeterminate and some woken calls leave choice point or suspension frames on the control stack, no argument needs be moved.

To support the delay mechanism, we modified the ATOAM in the following aspects:

- A new data type, called `SUSP`, is introduced for representing suspending variables. The unification procedure is modified to take suspending variables into account.
- The following two new registers are introduced: `SF` pointing to the latest suspension frame, and `TRIGGER` pointing to the list of trigger variables.
- The following four new instructions are introduced: `allocate_susp`, `susp_var`, `delay`, and `end_delay`.
- The management of the trail and control stacks is modified as described in the previous Subsection. In addition, the semantics of some instructions of the ATOAM are modified such that the wakeup event signal can be checked at the entry and exit points and the top of the control stack can be computed correctly at the exit point of every predicate.

5 Performance Evaluation

We have implemented the method described in this paper in B-Prolog, an emulator-based Prolog system, and compared the performance of B-Prolog with that of SICStus Prolog Version 3.0. For standard Prolog programs, B-Prolog is about 35 percent faster than the emulated code (SP-bc) and 40 percent as fast as the native code (SP-nc) of SICStus on a SPARC-10[3].

5.1 Benchmark Programs

The following benchmark programs were tested:

- **nreverse** The well known naive reverse program where the call to `append` is placed before the call to `nreverse` and calls to `append` are delayed if the first list to be concatenated is a variable. The size of the given list to be reversed is 500.
- **queens** A test-and-generate program for finding all solutions to the 8-queen problem, where the test-part generates a series of inequality constraints and the generate-part assigns values to variables. Each inequality constraint is invoked as soon as both of its arguments become ground.
- **sendmory** A test-and-generate program for solving the puzzle `SEND + MORE = MONEY`.
- **psort** The naive sort program that first generates constraints on all pairs of consecutive elements in the sorted list and then permutes the given list. The size of the input list is 15.

5.2 Execution Time

Table 1 shows the ratios of the cpu times taken by SICStus to those taken by B-Prolog. The numbers in the parentheses indicate the cpu times in seconds. We compared different constructs for describing delay available in the two systems. We used `freeze/2` and `delay` clause for B-Prolog, and `freeze/2` and `block` declaration

[3]Both systems were compiled by using "CC -O".

program	B	SP-bc		SP-nc	
	delay & freeze	block	freeze	block	freeze
nreverse	1(0.30s)	8.37	28.67	5.83	10.23
queens	1(0.23s)	2.22	13.87	1.00	3.00
sendmory	1(7.71s)	1.96	10.71	1.35	5.29
psort	1(2.42s)	2.31	13.26	0.90	2.20

Table 1: Ratios of cpu times.

program	Control	Heap	Trail	Total
nreverse	0.33	7.39	125.25	7.19
queens	0.37	2.87	0.26	0.70
sendmory	0.55	1.46	0.30	0.74
psort	0.38	1.64	0.38	0.95

Table 2: Ratios of space requirements ($\frac{SP}{BP}$).

for SICStus. While using `freeze` and `delay` clause does not cause much difference in execution time in B-Prolog, using `block` declaration is much faster than using `freeze/2` in SICStus.

For the four programs, B-Prolog is significantly faster than SP-bc and can in most cases beat SP-nc. The speed-ups are due mostly to the novel implementation method of delay adopted in B-Prolog. For the original `nreverse` program without delay, B-Prolog is only 45 percent faster than SP-bc.

It is difficult to tell to what extent delay affects the execution time because to do so we have to get rid of the time taken to run predicates that never delay. However, as more than 90 percent of the predicate calls in `nreverse` delay in execution, the ratios in the row for `nreverse` roughly tells us about the difference between the performance of the two systems.

5.3 Space Efficiency

Table 2 compares the space requirements for various stacks. SICStus uses two separate stacks, namely, the environment stack and the choice point stack, to represent the control stack in the WAM. The control stack space stands for the sum of the two stack spaces. In this comparison, we used delay clause for B-Prolog and block declaration for SICStus.

B-Prolog consumes more control stack space than SICStus because delayed calls are stored on the control stack. It takes at least $n + 6$ cells to store a delayed call with n arguments. In contrast, SICStus consumes more heap space than B-Prolog because delayed calls are stored on the heap. It takes $n + 1$ to store a delayed call with n arguments. In total, B-Prolog consumes much less space than SICStus for `nreverse`. Two reasons can be given. First, the tail recursion elimination technique is retained which can prevent the control stack space explosion from occurring for `nreverse`. Second, during forward execution, useless suspension frames can be reused in the ATOAM, whereas, delayed calls stored on the heap in the WAM all become garbage after their execution. For the remaining three programs, B-Prolog

consumes a little more space in total than SICStus. This is because these programs are all in test-and-generate style and no suspension frames can be reused.

6 Discussion

Although the space efficiency of our method is not so bad compared with that of SICStus for the benchmark programs, our strategy for frame reusage is still too weak to prevent control stack space explosion from occurring. Recall the producer & consumer program and see what would happen if both the producer and the consumer are mutually defined predicates. The consumer is delayed before the producer produces any data and a suspension frame is pushed onto the control stack. The producer, after being invoked, first has a frame pushed onto the control stack and then repeatedly produces data. After one item of data is produced, the consumer will be woken. Because the consumer is not a tail-recursively defined predicate, another new frame has to be pushed on top of the frame for the producer. In this way, no frame can be reused and thus linear size space will be required.

To deal with the space explosion problem, we eventually need a control stack garbage collector that can collect useless frames and move useful frames to the bottom end of the control stack. Garbage collection for the control stack can be done much more quickly than that for the heap because no pointers in other areas reference the control stack. The further work is to develop the garbage collector and investigate its impacts on the performance of the system for a large number of benchmark programs.

Acknowledgement

I am grateful to Mats Carlsson for answering my questions about how to get the data from SICStus-Prolog and the three referees for helpful comments on the presentation and performance evaluation.

References

[1] Ait-Kaci, H.: Functions as Passive Constraints in LIFE, *ACM Transactions on Programming Languages and Systems*, 16:1279-1318, 1994.

[2] Boizumault, P.: A General Model to Implement DIF and FREEZE, *Proc. 3rd ICLP*, 585-592, 1986.

[3] Carlsson, M.: Freeze, Indexing, and other Implementation Issues in the WAM, *Proc. 4th ICLP*, 40-58,1987.

[4] Chikayama, T.: *ESP Reference Manual*, Technical Report TR-044, ICOT, 1984.

[5] Colmerauer, A.: Equations and Inequations on Finite and Infinite Trees, *Proc. of the International Conference on Fifth Generation Computer Systems (FGCS'84)*, ICOT, 85-99, 1984.

[6] Debray, S.K.: QD-Janus: A Sequential Implementation of Janus in Prolog, *Software - Practice and Experience*, 23:1361-1377, 1993.

[7] De Schreye, D., Pollet, D., Ronsyn, J., Bruynooghe, M.: Implementing finite-domain constraint logic programming on top of a PROLOG-system with delay-mechanism, *Proc. ESOP '90. 3rd European Symposium on Programming*, 106-117, 1990.

[8] Hanus, M.: The Integration of Functions and Logic Programming: From Theory to Practice, *Journal of Logic Programming*, 19/20:583-628, 1994.

[9] Jaffar, J. and Maher, M.J.: Constraint Logic Programming: A Survey, *Journal of Logic Programming*, 19/20:503-581, 1994.

[10] Kawamura, T., Ohwada, H., Mizoguchi, F.: CS-prolog: a Generalized Unification-based Constraint Solver, *Proc. Logic Programming Conference*, 19-39, 1988,

[11] Meier, M.: Better Late Than Never, *Implementations of Logic Programming Systems*, Tick E. and Succi, G., Eds., Kluwer Academic Publishers, 1994.

[12] Naish, L.: *Negation and Control in Prolog*, Lecture Note in Computer Science, 238, 1985.

[13] Saraswat, V.A.: *Concurrent Constraint Programming*, MIT Press, 1993.

[14] Shapiro, E. (Ed.) : *Concurrent Prolog*, MIT Press, 1987.

[15] *SICStus Prolog User's Manual*, Programming Systems Group, Swedish Institute of Computer Science, 1995.

[16] Van Hentenryck, P. : *Constraint Satisfaction in Logic Programming*, MIT Press, 1989.

[17] Zhou, N.F.: Parameter Passing and Control Stack Management in Prolog Implementation Revisited, available through anonymous ftp from ftp.kyutech.ac.jp in the directory /pub/Language/prolog, an early version appears in *Proc. ICLP'94*, 159-173, 1994.