IEEE COMPUTER SOCIETY REPRINT

CONCURRENT PROLOG COMPILER ON TOP OF PROLOG

Kazunori Ueda
Takashi Chikayama

Reprinted from IEEE PROCEEDINGS OF THE SYMPOSIUM ON LOGIC PROGRAMMING, Boston, Massachusetts, July 15-18, 1985

IEEE COMPUTER SOCIETY
1730 Massachusetts Avenue, N.W.
Washington, D.C. 20036-1903

IEEE COMPUTER SOCIETY PRESS

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

CONCURRENT PROLOG COMPILER ON TOP OF PROLOG

Kazunori Ueda and Takashi Chikayama

C&C Systems Research Laboratories
NEC Corporation
4-1-1, Miyazaki, Miyamae-ku, Kawasaki 213 Japan

ICOT Research Center
Institute for New Generation Computer Technology
1-4-28, Mita, Minato-ku, Tokyo 108 Japan

ABSTRACT

A Concurrent Prolog compiler, whose target language is (sequential) Prolog, was implemented in Prolog. The object program obtained can further be compiled into machine codes by a Prolog compiler. Due to the similarity among the source, target and implementation languages, the compiler and the runtime support were small and very rapidly developed. Benchmark tests showed that (twice) compiled Concurrent Prolog programs ran 2.7 to 4.4 times faster and 2.7 to 5.3 times slower than comparable Prolog programs running on the interpreter and compiler, respectively, of the same Prolog system. After these experiments, the Concurrent Prolog compiler was modified to obtain a compiler of the new parallel logic programming language, GHC (Guarded Horn Clauses), and almost the same efficiency was achieved. These compilers will serve for practice of parallel logic programming.

1 INTRODUCTION

Since Shapiro proposed Concurrent Prolog and its interpreter[7], that interpreter had been used for the practice of parallel logic programming around us. Although the interpreter, written in Prolog, was concise and useful for experiments of small programs, the slowdown from the bare Prolog system on which the interpreter ran amounted to two orders of magnitude.

Therefore, we decided to develop a Concurrent Prolog compiler. Writing a compiler is important, because in order to demonstrate the viability of the language, it is necessary to provide a programming environment in which one can write and test parallel logic programs of considerable size.

We chose (sequential) Prolog for the target and the description language for the following reasons.

(1) A Prolog program can be compiled into efficient machine codes[13].

(2) Similarity among the source, target, and description languages enables rapid development.

(3) We can get a portable implementation.

(4) The laborious work of writing system predicates is greatly reduced by interfacing between Concurrent Prolog and Prolog.

We omit the syntax and the semantics of Concurrent Prolog here, which will be found in other documents[7,8,9].

Our approach is similar to the approach taken by Clark and Gregory when they wrote a PARLOG system on top of Prolog[4]. However, it did not optimize unification, and its performance on a compiler-based Prolog implementation has not been reported. We tried to get maximum efficiency on a compiler-based Prolog implementation, both to make our compiler as practical as possible, and to know how fast Concurrent Prolog programs run.

2 LINGUISTIC AND NON-LINGUISTIC FEATURES

Our implementation is basically a compiler version of the original interpreter[7]. Some linguistic extensions we have made are as follows:

(1) Metacall facilities[2] have been provided. The metacall predicate 'call' has three arguments:

```
call(Goals, Result, Interrupt)
```

The argument `Result` gets the value '`success`' upon successful termination of `Goals`. When one instantiates `Interrupt` to '`stop`', the execution of `Goals` is aborted and `Result` gets the value '`stopped`'.

(2) Input and output have been made applicative. There is no '`read`' or '`write`' predicate à la Prolog. Instead, we have '`instream`' and '`outstream`' predicates which take one argument: a stream of request messages. Each request message must be an appropriate Prolog I/O goal. For example, if the goal

```
outstream([write(ok), nl | _])
```

is executed, the message '`ok`' followed by a newline is output. In order to guarantee the uniqueness of the input and output streams, neither '`instream`' nor '`outstream`' can be called twice.

(3) Mode declaration facilities similar to those of DEC-10 Prolog[1] have been provided. The purpose is to get smaller and more efficient codes.

On the other hand, we inherit the following linguistic and non-linguistic restrictions from the original interpreter.

(1) Selection of candidate clauses is not done in a pseudo-parallel manner. That is, until the head unification or the execution of the guard of some clause has suspended or failed, another clause is not tried.

(2) There is no distinction between suspension and failure. A goal for which there are no immediately selectable clauses may be re-scheduled, whether the cause is ultimate failure or suspension.

(3) Goals which have suspended due to read-only annotations do busy-waiting.

Although these might look true restrictions at a glance, they actually cause little inconvenience to the execution of useful Concurrent Prolog programs that currently exist:

(1) There have been few programs that require the (pseudo-) parallel execution of two or more guards. Moreover, such programs can be rewritten to programs that need no OR-parallelism by using metacall and other facilities.

(2) Typical Concurrent Prolog programs, which perform their tasks using stream communication, are written so that all goals may succeed except for small ones in guards.

(3) By employing bounded depth-first scheduling (see below), the number of suspensions can be made small compared with the number of reductions in most applications.

Non-linguistic features include scheduling strategies and trace facilities.

Since conjunctive goals must be solved in (pseudo-) parallel, we have to decide how to schedule the goals. We decided to use one goal queue, and employed 100-bounded depth-first scheduling as a default strategy. ***N*-bounded depth-first scheduling** means that each newly-scheduled goal is n-reducible. A ***newly-scheduled goal*** is a goal which was enqueued at the rear and is now taken from the front. That a goal `G` is $n(>0)$***-reducible*** means that when `G` is reduced to `B1`, ..., `Bm` by the clause

```
H :- G1, ..., Gk | B1, ..., Bm.
```

each `Bi` (i=`1`, ..., `m`) is (n-1)-reducible prior to the execution of the other goals in the queue. That a goal `G` is ***0-reducible*** means that `G` must be pushed at the rear of the goal queue and the goal at the front must be scheduled.

It is easy to see that n-bounded depth-first scheduling is so general as to interpolate breadth-first and depth-first scheduling. Bounded depth-first scheduling has both the fairness of breadth-first scheduling and the efficiency of depth-first scheduling, as we will see later.

The bound value can be specified at run time. When finite-bounded depth-first scheduling is unnecessary, one can generate a simpler code that uses depth-first scheduling and will gain more efficiency.

Execution trace is enabled by compiling a source program with a 'trace' option.

3 COMPILATION TECHNIQUE

A general advantage of compiler approach is that we can statically determine parts of what we must determine at run time in interpreter approach. In the case of Concurrent Prolog, such parts include scheduling and unification. These two aspects are discussed in the following.

3.1 Scheduling

Our compiler has inherited the following notions from Shapiro's original interpreter:

(1) a queue (represented as a difference list) of goals to be solved

(2) a flag showing whether computation may deadlock or not

(3) a cycle marker for detecting termination and deadlock of computation.

However, our compiler has not inherited a scheduler predicate. The compiler generates one Prolog predicate (object code) for each Concurrent Prolog predicate, and when such a Prolog predicate is called, it is given as arguments a goal queue, which will be called a *continuation* hereafter. The called predicate must perform a scheduling task by itself, which consists of continuation management and the call of another (Prolog) goal. This goal is either the first goal in the given continuation or a goal provided by the predicate itself. In the former case, we must pop the first goal, give it the rest of the continuation and call it. In the latter case, the given continuation must be passed to the goal. A Prolog predicate generated by the compiler never fails unless it calls a nonexistent predicate.

Each Prolog predicate generated by the compiler consists of the following three parts:

(1) (optional) prelude part for tracing and clause indexing,

(2) clause-by-clause part (one Prolog clause for each Concurrent Prolog clause) for doing reductions, and

(3) postlude part for re-scheduling itself in case all clauses fail to commit or the bound value reaches zero.

Figure 1(a) and 1(b) show the source and the object programs of quicksort. Note that candidate clauses are tested sequentially in our implementation.

Each compiled clause has five additional arguments:

(1) A counter maintaining the current bound value necessary for bounded depth-first scheduling.

```
qsort([Pivot|Xs],Ys0,Ys2) :-        % Ys0-Ys2: d-list
  part(Xs?,Pivot,Small,Large),
  qsort(Small?,Ys0,[Pivot|Ys1]),
  qsort(Large?,Ys1,Ys2).

qsort([],Ys,Ys).                    % Ys-Ys: empty d-list

part([X|Xs],Pivot,Small,[X|Large]) :- Pivot <  X |
  part(Xs?,Pivot,Small,Large).

part([X|Xs],Pivot,[X|Small],Large) :- Pivot >= X |
  part(Xs?,Pivot,Small,Large).

part([],_,[],[]).
```

(a) Concurrent Prolog source program

```
:-fastcode.                         % Compiler option
:-public qsort/8.
:-mode qsort(?,?,?, +,?,-,+,+).
qsort(Arg1,Ys0,Ys2, Bold,H,T,Flag,B0) :-
  ulist(Arg1,Pivot,Xs),             % Arg1=[Pivot|Xs]
  Bold > 0, !,                      % Bound check
  Bnew is Bold-1,                   % Decrement bound
  part(Xs?,Pivot,Small,Large,
    Bnew,
    [$(qsort(Small?,Ys0,[Pivot|Ys1],
            Bnew,H1,T1,Flag1,B0),
                H1,T1,Flag1      ),   % Push 1st qsort
     $(qsort(Large?,Ys1,Ys2,
            Bnew,H2,T2,Flag2,B0),
                H2,T2,Flag2      )    % and 2nd qsort
    | H],                          % to the top of the continuation
    T,nd,B0).

qsort(Arg1,Ys0,Ys1, Bold,H,T,Flag,B0) :-
  unil(Arg1), unify(Ys0,Ys1),       % Ys0=Ys1, Arg1=[]
  Bold > 0, !,                      % Bound check
  H=[$(Goal,H1,T,nd)|H1],  % Pop the first goal, give
  incore(Goal).            % appropriate continuation and
                           % deadlock flag, and schedule it
```

```
qsort(Arg1,Arg2,Arg3,                % Suspension processing
  Bold,
  [$(Goal,H1,T1,Flag)|H1],           % Pop the first goal,
  [$(qsort(Arg1,Arg2,Arg3,
          B0,H2,T2,Flag2,B0),        % push qsort itself,
             H2,T2,Flag2     )|T1],
  Flag,B0) :- incore(Goal).          % and call it

:-public part/9.
:-mode part(?,?,?,?, +,?,-,+,+).
part(Arg1,Pivot,Small,Arg4, Bold,H,T,Flag,B0) :-
  ulist(Arg1,X1,Xs),                 % Arg1=[X1|Xs]
  ulist(Arg4,X2,Large),              % Arg4=[X2|Large]
  unify(X1,X2),                      % X1=X2
  Bold > 0,                          % Bound check
  cpwait(Pivot,Pivot_w),             % Wait for Pivot
  cpwait(X1,X1_w),                   % and X1
  Pivot_w < X1_w, !,                 % and compare them
  Bnew is Bold-1,                    % Decrement bound
  part(Xs?,Pivot,Small,Large, Bnew,H,T,nd,B0).

part(Arg1,Pivot,Arg3,Large, Bold,H,T,Flag,B0) :-
  ulist(Arg1,X1,Xs), ulist(Arg3,X2,Small),
  unify(X1,X2), Bold > 0,
  cpwait(Pivot,Pivot_w), cpwait(X1,X1_w),
  Pivot_w >= X1_w, !, Bnew is Bold-1,
  part(Xs?,Pivot,Small,Large, Bnew,H,T,nd,B0).

part(Arg1,_,Arg3,Arg4, Bold,H,T,Flag,B0) :-
  unil(Arg1), unil(Arg3),            % Arg1=[], Arg3=[],
  unil(Arg4),                        % Arg4=[]
  Bold > 0, !,
  H=[$(Goal,H1,T,nd)|H1], incore(Goal).

part(Arg1,Arg2,Arg3,Arg4,
  Bold, [$(Goal,H1,T1,Flag)|H1],
  [$(part(Arg1,Arg2,Arg3,Arg4,
         B0,H2,T2,Flag2,B0),
            H2,T2,Flag2     )|T1],
  Flag,B0) :- incore(Goal).
```

(b) Object program in DEC-10 Prolog

Fig. 1 Compiling Concurret Prolog into Prolog

(2), (3) The head and the tail of the difference list representing a continuation.

(4) A deadlock flag showing whether or not some goal has been reduced since the last occurrence of the cycle marker. This flag is used by the next cycle marker for deadlock detection.

(5) The initial bound value given at run time.

Each element of a continuation has the following form.

```
$(Goal, Qh, Qt, Deadlock_flag)
```

`Goal` is a Prolog goal corresponding to some Concurrent Prolog goal. When `Goal` is called, its additional arguments must be given appropriate values. `Qh`, `Qt`, and `Deadlock_flag` are 'taps' of `Goal` used for this purpose. That is, `Qh`, `Qt`, and `Deadlock_flag` have been unified with the second, the third, and the fourth additional arguments of `Goal`, respectively.

A Concurrent Prolog clause

```
Head :- Guard | Body.
```

is transformed into a Prolog clause of the following form.

(receiving arguments) :-
    (`Head` unification),
    (bound check),
    (executing `Guard`), !,
    (decrementing bound),
    (scheduling `Body`).

The 'bound check' and 'decrementing bound' parts are not generated if depth-first scheduling is specified.

The last part, 'scheduling `Body`', does the following things.

(1) When no body goals exist (i.e., `Body` is '`true`'), the first goal in the continuation is called.

(2) When just one body goal exists, that goal is given the same continuation that the current clause has received, and it is called.

```
:- public '$END'/3.
'$END'([],_,_) :- !.         % Succeeds if no goals remain
'$END'([$(Goal,H1,T1,d)|H1],      % Pop the first goal
       [$('$END'(H2,T2,Dnd2),     % and push itself
             H2,T2,Dnd2 )|T1],
     nd) :-                  % If deadlock flag is 'nd'
   incore(Goal).             % then call the popped goal
```

Fig. 2 System predicate for the detecion of deadlock and termination

```
:- public ulist/3.    :- mode ulist(?,-,-).
ulist([H|T],H,T) :- !.
ulist(X?,H,T) :- nonvar(X), ulist(X,H,T).

:- public unil/1.     :- mode unil(?).
unil([]) :- !.
unil(X?) :- nonvar(X), unil(X).

:- public cpwait/2.  :- mode cpwait(?,?).
cpwait(X?,Y) :- !, nonvar(X), cpwait(X,Y).
% Here, 1st arg is a non-variable.
cpwait(X,X).
```

Fig. 3 Some system predicates for unification and synchronization

(3) When two or more body goals exist, the second and the subsequent goals are put at the front of the continuation, and the first goal is called with the modified continuation.

Upon such a call, the fourth additional argument, the deadlock flag, is set to 'nd' (for 'no deadlock').

Of the above three cases, only the first case needs an indirect call; in the other cases, at least one of the body goals is directly called as long as the current bound value is not zero.

Avoiding indirect calls is important for efficiency. A major application of Concurrent Prolog is to describe a distributed system in which constituent processes, represented as conjunctive goals, communicate with one another using shared variables as streams[10]. In this case, most of the reductions use tail-recursive clauses having just one body goal. Our compiler translates such clauses into tail-recursive Prolog clauses. Since advanced Prolog implementations realize tail-recursion optimization which avoids the growth of the local stack and re-utilizes information left on the stack, a tail-recursive Concurrent Prolog program is expected to have good properties. Assume that 100-bounded depth-first scheduling is used and that $\ll 1\%$ of reductions use clauses with no bodies. Then, 99% of predicate calls are done by efficient direct scheduling.

The clause for handling suspension is included in the 'postlude' part of each predicate. It pushes the current goal at the rear of the given continuation, and calls its first goal.

Deadlock and termination are detected by a cycle marker: a call to the system predicate **'$END'** (Figure 2). This predicate receives a continuation and a deadlock flag, and simply terminates if the given continuation is empty. If the continuation is not empty and the deadlock flag has been set to 'nd' since the last call of **'$END'**, it enqueues itself, resets the deadlock flag to 'd' (for 'deadlock'), and calls the first goal in the continuation. Otherwise, the predicate **'$END'** fails. The goal **'$END'** is given as the initial continuation of a goal which is input from the terminal.

The object code of a clause having metacall facilities has to do more complex continuation processing. The goals to be solved under metacall facilities form a private queue with a special cycle marker, **'$CALLEND'**, pushed at the rear. Goals in the private queue are initiated by its manager, a **'$CALL'** goal, which is entered in a (more) global queue. When **'$CALL'** is called, the current continuation is not passed to the **'$CALL'** itself, but to the **'$CALLEND'** goal at the rear of the private queue maintained by the **'$CALL'** goal. Then **'$CALL'** examines whether a stop message has arrived, and if not, calls the first goal of its private queue, which starts the execution of the private queue. When **'$CALLEND'** is called after a while, it pushes the **'$CALL'** goal managing the new, reduced private queue at the rear of the global continuation, and calls the first goal in the continuation. If the goals to be solved under metacall facilities are uninstantiated, a different goal, **'$COMPCALL'**, is used instead of **'$CALL'**. A **'$COMPCALL'** goal waits until the goals are instantiated, forms a private queue, and then does the same thing as **'$CALL'** does.

3.2 Unification

Our implementation employs a Prolog functor '?' to represent read-only annotations. To realize the suspension mechanism of Concurrent Prolog, the unification procedure must be defined as a Prolog predicate. However, because one of the two terms to be unified is specified as a head argument, specialized unification procedures can be used depending on the form of the argument. The use of specialized unification procedures diminishes run-time overhead.

The code for unification is expanded at the beginning of each clause body in the form of a sequence of goals. For example, assume that one of the head arguments of some Concurrent Prolog clause is a list `[T1 | T2]`, where `T1` and `T2` are some terms. The corresponding Prolog code first tries to unify the goal argument with the term `[CAR | CDR]`, and if successful, executes the goals for processing its **CAR** and **CDR** according to the forms of `T1` and `T2`, which may in turn have been expanded. This idea is similar to the one employed in the DEC-10 Prolog compiler[13]: The only difference is that our compiler can expand a unification procedure to any level, as Warren's new Prolog machine architecture[14] enables.

Note that some unification procedures cannot be expanded: a general unification procedure must be used for a variable which occurs more than once in a head (e.g., the variable **Ys** in 'qsort' in Figure 1).

Figure 3 shows the definitions of some unification and synchronization procedures used in the program in Figure 1. The 'cpwait' predicate is used for interfacing between Concurrent Prolog and Prolog. Note that these system predicates in Prolog fail upon suspension: Suspension processing is done by the last clauses of the predicates in which those system predicates are called.

Mode declaration facilities allow a user to declare one of the following three modes for each predicate argument.

(1) Index mode ('+'): allows clause indexing if the underlying Prolog implementation allows it. The object code of a predicate having this mode has a two-stage structure: the first stage for waiting for the arguments of this mode, and the second stage for clause selection. This mode is useful when there are lots of clauses.

(2) Normal mode ('?'): specifies that the argument be processed in the ordinary way.

(3) Output mode ('-'): declares that the goal argument is always an uninstantiated non-read-only variable. For arguments of the output mode, implicit head unification of Prolog is used instead of explicit unification procedures.

Figure 4 shows how object codes are affected by a mode declaration.

4 PERFORMANCE

Table 1 shows some benchmark results. As for the Concurrent Prolog compiler, four timing data were obtained for each program: with bounded depth-first/depth-first scheduling and with/without mode declarations. The benchmark programs were timed also on the original interpreter with breadth-first scheduling. Moreover, for each program, a Prolog program having the same input-output relation was written and timed. The Prolog system we used is DEC-10 Prolog on DEC2060.

Table 1 shows that our object codes ran 12 to 220 times as fast as the original interpreter. Moreover, they ran 2.7 to 4.4 times as fast as the comparable Prolog programs processed by the DEC-10 Prolog interpreter. They were, of course, slower than the comparable Prolog programs processed by the compiler, but the slowdown was 1/2.7 to 1/5.3, which we think is quite reasonable.

The 'append' program ran at more than 11.5kRPS (kilo Reductions Per Second: equivalent to kLIPS if there are no guards).

The mode declaration was effective for all the benchmark programs. The speedup was 19% to 84%. As for the benchmark programs, the source of improvement is the declaration of the output mode. The speedup brought by changing bounded depth-first scheduling to depth-first scheduling was 27% or less.

The third program that performs bounded-buffer communication[10] was inefficient, because process switching took place very often. We can see from Table 1 that we can make this program 2.75 times faster only by changing the buffer size to 10.

The column showing the number of suspensions indicates that the bounded depth-first scheduling provides good behavior (except for bounded buffer programs). The ill behavior of the one-bounded buffer program is inevitable, because that behavior is what the program has explicitly specified.

5 BRIEF HISTORY

The system explained above is the second version. The first version realized head unification optimization, but all scheduling tasks were done by a scheduler predicate which managed the goal queue. Therefore, the first version can be considered as a step from the original interpreter towards the second version. Although less efficient, the first version had an advantage that detailed trace information could be more easily obtained. This reflects the fact that the degree of compilation was smaller compared with the second version.

Before writing the second version, we made several mock-ups of object codes for simple programs and tested them. After determining the object code format, it did not take much effort to complete the compiler.

6 FROM CONCURRENT PROLOG TO GHC

6.1 GHC—Guarded Horn Clauses

After the Concurrent Prolog compiler was developed, some problems were found in the Concurrent Prolog language rules[11], and another parallel logic programming language called Guarded Horn Clauses (GHC) was proposed[12].

The syntax of GHC is almost the same as that of Concurrent Prolog and PARLOG. A GHC program is a finite set of guarded Horn clauses of the following form

```
H :- G1, ..., Gm | B1, ..., Bn.   (m>0, n>0)
```

where H, Gi's, and Bi's are atomic formulas. A goal clause has the following form:

```
:- B1, ..., Bn.
```

The semantics of GHC is different from that of Concurrent Prolog in the following point. In Concurrent Prolog, the result of any unification

- which is invoked directly or indirectly in the head or the guard of a clause (hereafter we call this part a *passive part*) and
- which binds a variable in the caller of that clause with a non-variable term or another variable in that caller

```
append([A|X],Y,[A|Z]) :- append(X,Y,Z).
append([],Y,Y).

:- mode append2(+,?,-).
append2([A|X],Y,[A|Z]) :- append2(X,Y,Z).
append2([],Y,Y).
```

(a) Concurrent Prolog source program

```
:-fastcode.
:-public append/8.
:-mode append(?,?,?, +,?,-,+,+).
append(Arg1,Y,Arg3, Bold,H,T,Flag,BO) :-
  ulist(Arg1,A1,X), ulist(Arg3,A2,Z),
  unify(A1,A2),                      % Head unification
  Bold>0, !,                         % Bound check
  Bnew is Bold-1,                    % and updating
  append(X,Y,Z, Bnew,H,T,nd,BO).     % Tail recursion
append(Arg1,Y1,Y2, Bold,H,T,Flag,BO) :-
  unil(Arg1), unify(Y1,Y2),          % Head unification
  Bold>0, !,                         % Bound check
  H=[$(Goal,H1,T,nd)|H1],            % Pop the next goal
  incore(Goal).                      % and call it
append(Arg1,Arg2,Arg3,
  Bold, [$(Goal,H1,T1,Flag)|H1],
  [$(append(Arg1,Arg2,Arg3, BO,H2,T2,Flag2,BO),
                                H2,T2,Flag2    )|T1],
  Flag,BO) :- incore(Goal).       % Suspension processing
:-public append2/8.
:-mode append2(?,?,?, +,?,-,+,+).
append2(Arg1,Arg2,Arg3, Bold,H,T,Flag,BO) :-
  cpwait(Arg1,Arg1_w),                 % Wait for Arg1
  Bold > 0, !,                         % Bound check
  '$$$append2'(Arg1_w,Arg2,Arg3,
               Bold,H,T,Flag,BO).      % Clause selection
append2(Arg1,Arg2,Arg3,
  Bold, [$(Goal,H1,T1,Flag)|H1],
  [$(append2(Arg1,Arg2,Arg3, BO,H2,T2,Flag2,BO),
                                 H2,T2,Flag2    )|T1],
  Flag,BO) :- incore(Goal).        % Suspension processing
'$$$append2'([A|X],Y,[A|Z],        % Unification for Arg1
                                   % and Arg3 is embedded
             Bold,H,T,Flag,BO) :- !,
  Bnew is Bold-1,                      % Bound updating
  append2(X,Y,Z, Bnew,H,T,nd,BO).      % Tail recursion
'$$$append2'([],Y,Y,               % Unification for Arg1
                                   % and Arg3 is embedded
             Bold,H,T,Flag,BO) :- !,
  H=[$(Goal,H1,T,nd)|H1],          % Pop the next goal
  incore(Goal).                    % and call it
'$$$append2'(Arg1,Arg2,Arg3,
  Bold, [$(Goal,H1,T1,Flag)|H1],
  [$('$$$append2'(Arg1,Arg2,Arg3,
                  BO,H2,T2,Flag2,BO),
                     H2,T2,Flag2    )|T1],
  Flag,BO) :- incore(Goal).        % Suspension processing
```

(b) Object program in DEC-10 Prolog

Fig. 4 The effect of mode declaration

Table 1. Concurrent Prolog Benchmark on DEC2060

Program	Processing (*1)	Reductions	Suspensions	Time(*2)/RPS(*3) (compiler without mode)	(compiler with mode)	(interpreter)
List concatenation (Append) (500+0 elements)	B	502	0	—	—	2313 / 217
	BD100	502	0	88.7/ 5660	54.8/ 9160	—
	D	502	0	79.0/ 6350	43.0/11700	—
	P			15.8/31800	11.9/42200	188 / 2670
Stream merge (100+100 elements)	B	202	0	—	—	1005 / 201
	BD100	202	0	42.9/ 4710	28.7/ 7040	—
	D	202	0	38.4/ 5260	23.6/ 8560	—
	P			8.3/24300	8.0/25300	73.7/ 2740
Bounded buffer (size=1)(*4)	B	204	0	—	—	1473 / 138
	BD100	204	200	147 / 1390	121 / 1690	—
	D	204	200	143 / 1430	119 / 1710	—
Bounded buffer (size=10)(*4)	B	204	0	—	—	1470 / 139
	BD100	204	20	60.2/ 3390	47.6/ 4290	—
	D	204	20	56.3/ 3620	43.3/ 4710	—
Primes (2 to 300) (without output)	B	2778	8445	—	—	80521 / 35
	BD100	2778	73	966 / 2880	769 / 3610	—
	D	2778	0	886 / 3140	689 / 4030	—
	P			216 /12900	188 /14800	2969 / 936
Quicksort (50 elements)	B	378	2225	—	—	20233 / 19
	BD100	378	0	125 / 3020	96.5/ 3920	—
	D	378	0	119 / 3180	91.3/ 4140	—
	P			21.3/17700	17.3/21800	246 / 1540

*1 B—breadth-first scheduling; BD100—bounded depth-first scheduling (bound=100); D—depth-first scheduling; P—Prolog-10 compiler (with 'fastcode' option) and interpreter.
*2 In milliseconds. Overhead for timing has been excluded:
*3 RPS—number of Reductions Per Second. An RPS value does not count reductions in guards. RPS values of Prolog programs were calculated using the number of reductions of the corresponding Concurrent Prolog programs.
*4 A Prolog counterpart does not exist.

```
qsort([Pivot|Xs], Ys0, Ys2) :- true |
  part(Xs, Pivot, Small, Large),
  qsort(Small, Ys0, [Pivot|Ys1]),
  qsort(Large, Ys1, Ys2).
qsort([],         Ys0, Ys1) :- true | Ys0 = Ys1.

part([X|Xs], Pivot, Small, Large) :- Pivot <  X |
  Large = [X|L1], part(Xs, Pivot, Small, L1).
part([X|Xs], Pivot, Small, Large) :- Pivot >= X |
  Small = [X|S1], part(Xs, Pivot, S1, Large).
part([],     _,     Small, Large) :- true        |
  Small = [], Large = [].
```

(a) GHC source program

```
:-fastcode.
:-public qsort/8.
:-mode qsort(?,?,?, +,?,-,+,+).
qsort(Arg1,Ys0,Ys2, Bold,H,T,Flag,B0) :-
  nonvar(Arg1), ulist(Arg1,Pivot,Xs),
  Bold > 0, !, Bnew is Bold-1,
  part(Xs,Pivot,Small,Large,
    Bnew,
    [$(qsort(Small,Ys0,[Pivot|Ys1],
             Bnew,H1,T1,Flag1,B0),
                  H1,T1,Flag1    ),
     $(qsort(Large,Ys1,Ys2,
             Bnew,H2,T2,Flag2,B0),
                  H2,T2,Flag2    )
    | H],
    T, nd, B0).
qsort(Arg1,Ys0,Ys1, Bold,H,T,Flag,B0) :-
  nonvar(Arg1), unil(Arg1),
  Bold > 0,
  Ys0 = Ys1,!,
  H = [$(Goal,H1,T,nd)|H1], incore(Goal).
qsort(Arg1,Arg2,Arg3,
  Bold, [$(Goal,H1,T1,Flag) | H1],
  [$(qsort(Arg1,Arg2,Arg3,
          B0,H2,T2,Flag2,B0),
               H2,T2,Flag2    ) | T1],
  Flag,B0) :- incore(Goal).
:-public part/9.
:-mode part(?,?,?,?, +,?,-,+,+).
part(Arg1,Pivot,Small,Large, Bold,H,T,Flag,B0) :-
  nonvar(Arg1), ulist(Arg1,X,Xs),
  Bold > 0,
  nonvar(Pivot), nonvar(X), Pivot < X,
  Large = [X|L1], !,
  Bnew is Bold-1,
  part(Xs,Pivot,Small,L1, Bnew,H,T,nd,B0).
part(Arg1,Pivot,Small,Large, Bold,H,T,Flag,B0) :-
  nonvar(Arg1), ulist(Arg1,X,Xs),
  Bold > 0,
  nonvar(Pivot), nonvar(X), Pirot >= X,
  Small = [X|S1], !,
  Bnew is Bold-1,
  part(Xs,Pivot,S1,Large, Bnew,H,T,nd,B0).
part(Arg1,_,Small,Large, Bold,H,T,H,I) :-
  nonvar(Arg1), unil(Arg1),
  Bold > 0,
  Small = [], Large = [], !,
  H = [$(Goal,H1,T,nd) | H1], incore(Goal).
part(Arg1,Arg2,Arg3,Arg4,
  Bold, [$(Goal,H1,T1,Flag) | H1],
  [$(part(Arg1,Arg2,Arg3,Arg4,
         B0,H2,T2,Flag2,B0),
            H2,T2,Flag2    ) | T1],
  Flag,B0) :- incore(Goal).
```

(b) Object program in DEC-10 Prolog

```
:- public unil/1.    :- mode unil(+).
unil([]).
:- public ulist/3.   :- mode ulist(+,-,-).
ulist([H|T], H, T).
```

(c) System predicates for unification

Fig. 5 Compiling GHC into Prolog

must be recorded locally until commitment[7]*. In GHC, any unification which makes such bindings simply suspends instead. This modified rule provides GHC with a synchronization primitive: Read-only annotation is no longer necessary.

The major difference of GHC from PARLOG and Kernel PARLOG[3] is that GHC has neither modes or specialized unifications at the language level. GHC shares the suspension mechanism with Qute[6].

Figure 5(a) shows a quicksort program—GHC counterpart of the program in Figure 1(a). Note that unification for exporting bindings through head arguments must be specified in the body of a clause. Explicit unification ('=') is heavily used for this purpose in a GHC program.

* Shapiro's interpreter and our compiler do not maintain local environments: all bindings are global. However, since these systems perform clause selection as an indivisible operation using Prolog's backtracking, no problems arise.

6.2 GHC Compiler

Due to the similarity of Concurrent Prolog and GHC, all we had to do to develop a GHC system was to modify the code generator for passive parts and some runtime support routines including unification.

The current GHC system does not allow user-defined goals in a guard, since this restriction makes generation of codes for suspension easier. Suspension of unification invoked in the passive part of a guarded Horn clause is realized by the 'nonvar' and '==' goals expanded in Prolog codes. The goal '`nonvar(X)`' precedes the codes for unification which may bind `X` occurring in a head with a non-variable. The goal '`X1==X2`' is generated for two GHC variables `X1` and `X2` which both occur in a head and are unified in a guard. The predicate '==' is also used for the unification operation involved in the multiple occurrences of a variable in a head. Figure 5(b) shows the object code of the GHC program shown in Figure 5(a). Figure 5(c) shows the unification procedures used in Figure 5(b).

Compiled GHC programs ran almost as fast as the comparable Concurrent Prolog programs.

There are two approaches to allowing user-defined goals in a guard: static approach and dynamic approach. In static approach, one must analyze all user-defined goals in guards to determine whether each piece of unification which may be performed can suspend or not. This approach is used in PARLOG for compile-time mode analysis[3]. Dynamic approach is used in Miyazaki's compiler[5]. This compiler makes use of the fact that in DEC-10 Prolog, a newer global variable has a larger address than older ones. This fact enables distinguishing non-writable variables (in a caller) from writable ones (newly created in a passive part) by using a 'threshold' address.

7 CONCLUDING REMARKS

We have implemented fast, portable compilers of Concurrent Prolog and GHC on top of Prolog. If a Prolog system is available, one can immediately get started with parallel logic programming.

Both Concurrent Prolog and GHC systems are less than 800 lines long. It took only a few days to have the first working version of the Concurrent Prolog compiler. Modifying the Concurrent Prolog system to make the GHC system took one and a half days. In other methods, it would take much more efforts to make a system with the same efficiency. It is well known that Prolog is a good tool for rapid prototyping of another logic programming language, but all these facts show that an efficient Prolog implementation is a good tool also for getting an *efficient* implementation of another logic programming language rapidly.

REFERENCES

[1] Bowen D. L. (ed.), *DECsystem-10 PROLOG User's Manual*, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1983.

[2] Clark, K. L., Gregory, S., "PARLOG: Parallel Programming in Logic," Research Report DOC 84/4, Dept. of Computing, Imperial College, London, 1984.

[3] Clark K. L., Gregory S., "Notes on the Implementation of PARLOG," Research Report DOC 84/16, Dept. of Computing, Imperial College, London, 1984.

[4] Gregory S., "How to Use PARLOG (C-Prolog version)," Dept. of Computing, Imperial College, London, 1984.

[5] Miyazaki, T., GHC-to-Prolog compiler, unpublished program, 1985.

[6] Sato, M. and Sakurai, T., "Qute: A Functional Language Based on Unification," *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, Institute for New Generation Computer Technology, pp. 157–165, 1984.

[7] Shapiro, E. Y., "A Subset of Concurrent Prolog and Its Interpreter," ICOT Technical Report TR-003, Institute for New Generation Computer Technology, 1983.

[8] Shapiro, E. and Takeuchi, A., "Object Oriented Programming in Concurrent Prolog," *New Generation Computing*, Vol. 1, No. 1, pp. 25–48, 1983.

[9] Shapiro E., "Systems Programming in Concurrent Prolog," *Conf. Record of the 11th Annual ACM Symp. on Principles of Programming Languages*, pp. 93–105, 1984.

[10] Takeuchi, A., Furukawa K., "Interprocess Communication in Concurrent Prolog," *Proc. Logic Programming Workshop '83*, Universidade nova de Lisboa, 1983.

[11] Ueda, K., "Concurrent Prolog Re-Examined," ICOT Tech. Report TR-102, Institute for New Generation Computer Technology, 1985.

[12] Ueda, K., "Guarded Horn Clauses," ICOT Tech. Report TR-103, Institute for New Generation Computer Technology, 1985.

[13] Warren, D. H. D., "Implementing PROLOG—Compiling Predicate Logic Programs," Vol. 1–2, D. A. I. Research Report No. 39, Dept. of Artificial Intelligence, University of Edinburgh, 1977.

[14] Warren, D. H. D., "An Abstract Prolog Instruction Set," Tech. Report 309, Artificial Intelligence Center, SRI International, 1983.