TR-092

Concurrent Prolog Compiler on Top of Prolog

Kazunori Ueda (NEC Corporation)
and
Takashi Chikayama (ICOT)

December, 1984

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

Concurrent Prolog Compiler on Top of Prolog

Kazunori Ueda
(C&C Systems Research Laboratories, NEC Corporation)
and
Takashi Chikayama
(Institute for New Generation Computer Technology)

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 results show that a compiled Concurrent Prolog program runs much faster than a comparable Prolog program running on a Prolog interpreter. This compiler will serve for practice of parallel logic programming.

1. INTRODUCTION

Since Shapiro proposed Concurrent Prolog and its interpreter in [Shapiro 83], that interpreter has been used for the practice of parallel logic programming around us. Although the interpreter, written in Prolog, is concise and useful for experiments of small programs, the slowdown from the bare Prolog system on which the interpreter runs amounts to two orders of magnitude.

Therefore, we decided to write a Concurrent Prolog compiler. Writing a compiler is important from the following viewpoints.

1. To get an efficient implementation, it is necessary to examine how much static information can be extracted from the source program.

2. To demonstrate the descriptive power 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 [Warren 77].

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

3. We can get a portable implementation.

4. Much of the laborious work to write system predicates can be eliminated by interfacing between Concurrent Prolog and Prolog.

We omit the description of Concurrent Prolog here, which will be found in [Shapiro 83], [Shapiro and Takeuchi 83], and [Shapiro 84].

2. LINGUISTIC AND NON-LINGUISTIC FEATURES

Our implementation is basically a compiler version of the original interpreter in [Shapiro 83]. Some linguistic extensions we have made are as follows:

1. Metacall facilities after [Clark and Gregory 84] have been provided. A 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. Sequential AND operator has been provided. Although this can be implemented by using metacall facilities, it is necessary to directly support this for the sake of efficiency.

3. Mode declaration facilities similar to those of DEC-10 Prolog [Bowen 83] 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 are not done in a pseudo-parallel manner. That is, until the head unification and the execution of the guards of some clause has suspended or failed, another clause is not tried.

2. There is no distinction between suspension and failure. A goal that has no immediately selectable clauses may be re-scheduled, whether the cause is ultimate failure or suspension.

3. Suspended goals due to read-only annotations do busy-waiting.

Although these might look true restrictions at a glance, they actually cause little inconvenience for executing useful Concurrent Prolog programs that currently exist:

1. There have been few programs that requires the (pseudo-)parallel execution of two or more guards. Moreover, such programs can be described without OR-parallelism by using metacall and other facilities.

2. By employing bounded depth-first scheduling which we will describe later, the number of suspensions can be made small compared with the number of reductions in most applications.

3. In typical Concurrent Prolog programs which perform their tasks using stream communication, all goals succeed except for small goals in guards.

Non-linguistic features include scheduling strategies and trace facilities.

Since we have to solve conjunctive goals in (pseudo-)parallel, we have to decide how to schedule the goals. We have employed one goal queue, and have 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 end of the goal queue and the goal at the front must be scheduled.

It is easy to see that 1-bounded depth-first is equivalent to breadth-first and oo-bound depth-first is equivalent to depth-first. That is, the n-bounded depth-first scheduling is general and it interpolates breadth-first and depth-first scheduling.

The bound value can be specified at run time. If finite-bounded depth-first scheduling is not necessary, one can compile a program in depth-first mode and can gain more efficiency.

Execution trace is enabled by compiling a source program in `trace' mode.

3. COMPILING TECHNIQUE

A general advantage of the compiler approach is that we can statically determine parts of what we must determine at run time in the 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

Each Concurrent Prolog predicate is compiled into a Prolog predicate in a clause-by-clause manner. This clause-by-clause part is preceded by a `prelude' part for handling trace and clause indexing, and followed by a `postlude' part for handling trace and suspension. Figure 1(a) and 1(b) show the source and the object program of quicksort.

Each compiled clause has five additional arguments. The first one is a counter maintaining a current bound value necessary for bounded depth-first scheduling. The second and the third ones form a difference list standing for a goal queue. We call this difference list a continuation. The fourth one is a flag showing whether some goal has been reduced or not. The fifth one is the initial value of the bound.

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),
    (executing Body).
```

The last part, `(executing Body)', does the following things.

1. When no body goals exist (i.e., `Body' is `true'), a goal at the front of the continuation is called.

```
qsort([X|Xs], Ys0, Ys2) :-
  partition(Xs?, X, S, L),
  qsort(S?, Ys0, [X|Ys1]), qsort(L?, Ys1, Ys2).
qsort([], Ys, Ys).

:- mode partition(?, ?, -, -).
partition([X|Xs], A, S, [X|L]) :- A <  X |
  partition(Xs?, A, S, L).
partition([X|Xs], A, [X|S], L) :- A >= X |
  partition(Xs?, A, S, L).
partition([], _, [], []).
```

(a) Concurrent Prolog source program

```
:-fastcode.
:-public qsort/8.
:-mode qsort(?,?,?,+,?,-,+,+).
qsort(A,B,C,D,E,F,G,H) :-
  ulist(A,I,J), D>0, !,
  K is D-1,
  partition(J?,I,L,M,K,
    [$(qsort(L?,B,[I|N],K,O,P,Q,H),O,P,Q),
     $(qsort(M?,N,C,K,R,S,T,H),R,S,T)|E],
    F,nd,H).
qsort(A,B,C,D,E,F,G,H) :-
  unil(A), unify(B,C), D>0, !,
  E=[$(I,J,F,nd)|J], incore(I).
qsort(A,B,C,D,[$(E,F,G,H)|F],
  [$(qsort(A,B,C,I,J,K,L,I),J,K,L)|G],H,I) :-
  incore(E).

:-public partition/9.
:-mode partition(?,?,?,?,+,?,-,+,+).
partition(A,B,C,[D|E],F,G,H,I,J) :-
  ulist(A,D,K), F>0, cpwait(B,L), cpwait(D,M), L<M, !,
  M is F-1,
  partition(K?,B,C,E,N,G,H,nd,J).
partition(A,B,[C|D],E,F,G,H,I,J) :-
  ulist(A,C,K), F>0, cpwait(B,L), cpwait(C,M), L>=M, !,
  N is F-1,
  partition(K?,B,D,E,N,G,H,nd,J).
partition(A,B,[],[],C,D,E,F,G) :-
  unil(A), C>0, !,
  D=[$(H,I,E,nd)|I], incore(H).
partition(A,B,C,D,E,[$(F,G,H,I)|G],
  [$(partition(A,B,C,D,J,K,L,M,J),K,L,M)|H],
  I,J) :- incore(F).
```

(b) Object program in DEC-10 Prolog

```
:- public '$END'/3.
'$END'([], _, _) :- !.
'$END'([$(G,Ch,Ct,d)|Ch],
  [$('$END'(Ch2,Ct2,Dnd2),Ch2,Ct2,Dnd2)|Ct],
  nd) :- incore(G).
```

(c) System predicate for the detection of deadlock and termination

Fig. 1 Compiling Concurret Prolog into Prolog

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

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

Upon these calls, the deadlock flag in the fourth additional argument is reset.

The first case is the only one in which indirect call to a goal is necessary; in the other cases, at least one of the body goals is directly called.

Avoiding indirect calls is important from the viewpoint of efficiency. A major application of Concurrent Prolog is to describe a distributed system in which constituent processes, represented as goals, communicate with one another using shared variables in a stream manner [Takeuchi and Furukawa 83]. 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 to avoid the growth of the local stack, the final code of the Concurrent Prolog tail-recursive program is expected to have good properties.

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

Deadlock and termination are detected by the system predicate called `$END' (Fig. 1(c)). This predicate simply terminates if the given continuation is empty. Otherwise, it enqueues itself, set the deadlock flag, and calls the first goal in the continuation as long as the deadlock flag has been reset since the last call of `$END'. 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 sequential AND and/or metacall facilities has to do more complex continuation processing, but the

basic idea remains the same.

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 written as a head argument, specialized unification procedures can be used depending on the form of the argument.

The code for unification appears at the top of each clause body in the form of a sequence of goals. For example, assume that the head argument is a list. The corresponding code first checks whether the goal argument is unifiable with the form [_ | _], and if unifiable, calls unification procedures for processing its CAR and CDR, which may in turn be expandable. This idea is borrowed from DEC-10 Prolog compiler [Warren 77]. The only difference is that our compiler can expand a unification procedure to any level.

The 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 processing read-only annotations, 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 an ordinary way.

3. Output mode (`-'): declares that the goal argument is always an uninstantiated non-read-only variable. The explicit unification procedure can be replaced by the implicit head unification of Prolog.

Figure 2 shows how object codes are affected by a mode declaration.

Specifying the index mode never changes the semantics of the original program. Specifying the output mode does not change the semantics of

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

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

(a) Concurrent Prolog source program

```
:-fastcode.
:-public append/8.
:-mode append(?,?,?,+,?,-,+,+).
append(A,B,C,D,E,F,G,H) :-
  ulist(A,I,J), ulist(C,K,L), unify(I,K), D>0, !,
  M is D-1,
  append(J,B,L,M,E,F,nd,H).
append(A,B,C,D,E,F,G,H) :-
  unil(A), unify(B,C), D>0, !,
  E=[$(I,J,F,nd)|J], incore(I).
append(A,B,C,D,[$(E,F,G,H)|F],
  [$(append(A,B,C,I,J,K,L,I),J,K,L)|G],H,I) :-
  incore(E).

:-public append2/8.
:-mode append2(?,?,?,+,?,-,+,+).
append2(A,B,C,D,E,F,G,H) :-
  cpwait(A,I), D>0,
  '$$$append2'(I,B,C,D,E,F,G,H).
append2(A,B,C,D,[$(E,F,G,H)|F],
  [$(append2(A,B,C,I,J,K,L,I),J,K,L)|G],H,I) :-
  incore(E).
'$$$append2'([A|B],C,[A|D],E,F,G,H,I) :- !,
  J is E-1,
  append2(B,C,D,J,F,G,nd,I).
'$$$append2'([],A,A,B,C,D,E,F) :- !,
  C=[$(G,H,D,nd)|H], incore(G).
```

(b) Object program in DEC-10 Prolog

Fig. 2 The effect of mode declaration

the original program as long as the declared restriction is obeyed.

4. PERFORMANCE

Table 1 shows some benchmark results. As for the Concurrent Prolog compiler, four timing data were obtained for each program: in bounded depth-first and in depth-first mode with and without mode declarations. The programs were timed also on the original interpreter in breadth-first mode. Moreover, for each program, a Prolog program which has the same input-output relation was written and was 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 (Reductions Per Second: equivalent to kLIPS if there are no guards), and the `naive reverse' program in [Warren 77] (not in the table) ran at more than 8.0kRPS.

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

The fourth program that performs bounded-buffer communication [Takeuchi and Furukawa 83] is inefficient, because process switching takes place very often. We can see from Table 1 that we can improve the efficiency by 2.75 times only by changing the buffer size to 10. The column of the number of suspensions indicates that the bounded depth-first scheduling provides fairly good behavior (except for bounded buffer programs), while allowing non-terminating programs to run. The ill behavior of the one-bounded buffer program is inevitable, because that behavior has been explicitly specified in the program.

Table 1. Concurrent Prolog Benchmark on DEC2060

Program	Processing (*1)	Reductions	Suspensions	Time(*2)/RPS(*3) (compiler without mode)	(compiler with mode)	(interpreter)
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
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 = Concurrent Prolog breadth-first mode;
BD100 = Concurretn Prolog bounded depth-first mode (bound=100);
D = Concurrent Prolog depth-first mode;
P = Prolog-10 compiler ('fastcode' mode) 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 Concurret Prolog programs.
*4 A Prolog counterpart does not exist.

5. BRIEF HISTORY

The version we have explained above is the second version. The first version concentrated mainly on optimizing head unification, and all scheduling tasks were done by a scheduler predicate which manages the goal queue. Therefore, the first version can be considered as a step from the original interpreter towards the current version. Although less efficient, the first version had an advantage that more detailed trace information can be easily obtained. This reflects the fact that the degree of compilation was smaller compared with the current 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.

In retrospect, our compiler development process seems closely related to partial evaluation. We must have had a partial evaluator of Prolog programs in mind. We must have applied it to the Shapiro's original interpreter with respect to some example programs, from which we learned how to compile a Concurrent Prolog program into a Prolog program and wrote a compiler. A little bit slower version of the compiler could have been made from the original interpreter mechanically by partial evaluation and program transformation.

6. CONCLUDING REMARKS

We have implemented a fast, portable Concurrent Prolog compiler on top of Prolog. If a Prolog system is available, one can immediately get started with parallel logic programming. Parallel programming requires much training and experience, and our system would help a lot.

The size of the compiler is less than 300 lines, and the size of the runtime support is also less than 300 lines. It took only a few days to have the first working version. 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

[Bowen 83] Bowen D.L. (ed.), DECsystem-10 PROLOG User's Manual, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1983.

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

[Shapiro 83] Shapiro, E.Y., A Subset of Concurrent Prolog and Its Interpreter, ICOT Technical Report TR-003, Institute for New Generation Computer Technology, 1983.

[Shapiro and Takeuchi 83] Shapiro, E. and Takeuchi, A., Object Oriented Programming in Concurrent Prolog, New Generation Computing, Vol.1, No.1, pp.25-48, 1983.

[Shapiro 84] Shapiro E., Systems Programming in Concurrent Prolog, Conf. Record of the 11th Annual ACM Symp. on Principles of Programming Languages, pp.93-105, 1984.

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

[Warren 77] Warren, D.H., Implementing PROLOG--Compiling Predicate Logic Programs, Vol.1-2, D.A.I. Research Report No.39, Dept. of Artificial Intelligence, University of Edinburgh, 1977.