ICOT Technical Memorandum: TM-0831

TM-0831

Implementation and Evaluation of Dynamic Predicate on Sequential Inference Machine CHI

by
K. Atarashi, A. Konagaya, S. Habata
& M. Yokota (NEC)

December, 1989

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

Implementation and Evaluation of Dynamic Predicate on Sequential Inference Machine CHI

A. Atarashi, A. Konagaya, S. Habata, M. Yokota
C&C Systems Research Laboratory
NEC Corporation
4-1-1 Miyazaki Miyamae-ku
Kawasaki 213, Japan.

Keywords: Prolog, Logic Programming, Dynamic Predicate, Sequential Inference Machine

Abstract

This paper proposes the new technique, named Dynamic Clause Compilation in order to accelerate the execution of Prolog's dynamic predicates.

Using this method, dynamic predicates are compiled into the machine instructions like the ordinary static predicates, when they are assert'ed. They are then executed fast, when called from other predicates.

The measurement result using practical applications proves that dynamic clause compilation accelerates the execution speed of each dynamic predicate over 10 times, and shorten total execution time of those applications from 1/3 to 1/5 than conventional Prolog implementation

1 Introduction

The compiling technique based on WAM[11] has increased the execution speed of Prolog program remarkably. However some application programs can not take advantage of this compiling technique because of dynamic predicates.

In practical Prolog application programs, assert and retract are indispensable functions to alter the database or rules dynamically.

Since such modifiable predicates, called dynamic predicates, change their definition during program execution, they can not be compiled into concrete machine instructions. Further, predicates can be manipulated as data(functors) any time. This requires restoring source image and makes compiling dynamic predicates difficult. Because of reasons listed above, conventional Prolog system treats dynamic predicates as structured data and executes them by interpreter.

According to our analysis of application programs, however, some application programs spend more than the half of total execution time in manipulating dynamic predicates. This means that their execution speed dominates the performance of such practical Prolog applications, such as knowledge base system and rule based system.

To solve this problem, we introduced incremental compiling approach and named Dynamic Clause Compilation. Using this method, as soon as dynamic predicates are asserted, they are compiled into machine instructions.

Obviously, this approach makes slow down the execution speed of assert. Therefore, trade-off between assertion and execution of dynamic predicates is much important for effectiveness of dynamic clause compilation. We have implemented this dynamic clause compilation system on the sequential in-

ference machine CHI[8, 4]. And this research was done as a part of Japan's Fifth Generation Computer Project.

Our measurement using practical application program shows that the frequency of execution is much higher than those of assertion and total execution speed of that application increased 3 to 5 times compared to conventional Prolog implementation.

The organization of this paper is as follows: In section 2, we show how to solve restoring source image in dynamic clause compilation method. Section 3 is devoted to the implementation mechanism, especially internal data structure of dynamic predicates and special instructions to execute them. Finally, in section 4, we will show the measurement results and performance analysis using actual Prolog applications.

2 Source Image Restoration Problem

Since each clause has simple structure and has only local variables, it seems easier to introduce incremental compiling approach in Prolog than other programing languages. However, source image restoration problem must be solved in case of Prolog.

In general, the source image disappears at object level after compiling. However, source image of clauses is required in case of Prolog. For example, Prolog predicate `clause` can extract the source image of a clause. Or, using `retract`, we can delete clauses, whose source image match with the given argument.

```
?- assert((f(X,Y):-g(X),h(Y))).    % add a clause
yes
?- clause(f(X,Y),Body,_).          % restore source image
Body = g(X),h(Y)                   % Body is unified with g(X),h(Y)
yes
?- retract((f(X,Y):-g(X),h(Y))).   % delete a clause
yes
```

As for source image restoration, three methods are known.

Hold source image

The object code always holds source image as text or as structured data (functor). This approach is simple, but restoring process is slow. Object code size also becomes large.

De-compile from execution code [1]

In this method, the object for execution also performs source image restoration. This is somewhat tricky approach and satisfies fast restoring and less object code size. However, it is impossible to generate well optimized and fastest object code, since the object code can not discard the information required to restore the source image.

Generate a code to restore source image [2]

When a clause is asserted, generate a code to restore source image, as well a code for execution. Using this method, source image restoration is the very fast, and code can be optimized freely. However, compilation takes longer and the size for the entire code becomes big.

Since we find the execution speed the most important, we decided to adopt an improvement of the third method, with regards to code manage-

ment. The improvement we made is that we manage the code for execution and the code for source image restoration together, whereas Clocksin's method manages them separately and requires such predicate as `eraselast`.

As for the compilation time increase, we find that the frequency of clause addition is much less frequent than the frequency of dynamic clause execution, and that the increase of dynamic compilation time does not affect the total performance of application programs so much. As for the code size increase, the memory consumption problem of global stack area is much more important, and such increase of code size is not a serious problem.

3 Dynamic Predicate Implementation on CHI

In this section, we describe the details of dynamic predicate implementation on CHI.

3.1 Mechanism of Dynamic Predicate Execution

Figure 1 shows internal representation of a dynamic predicate. A functor manages a predicate, and holds code for the predicate. In this figure, predicate `f/1` has four clauses.

```
f([a]).
f(_).
f(a).
f(b(_)).
```

To allow addition/deletion of clauses at arbitrary time, clauses are manages separately, and are bi-directionary chained.

The major difference of codes for dynamic predicates and codes for static predicates, from the viewpoints of execution, is briefly summarized as follows:

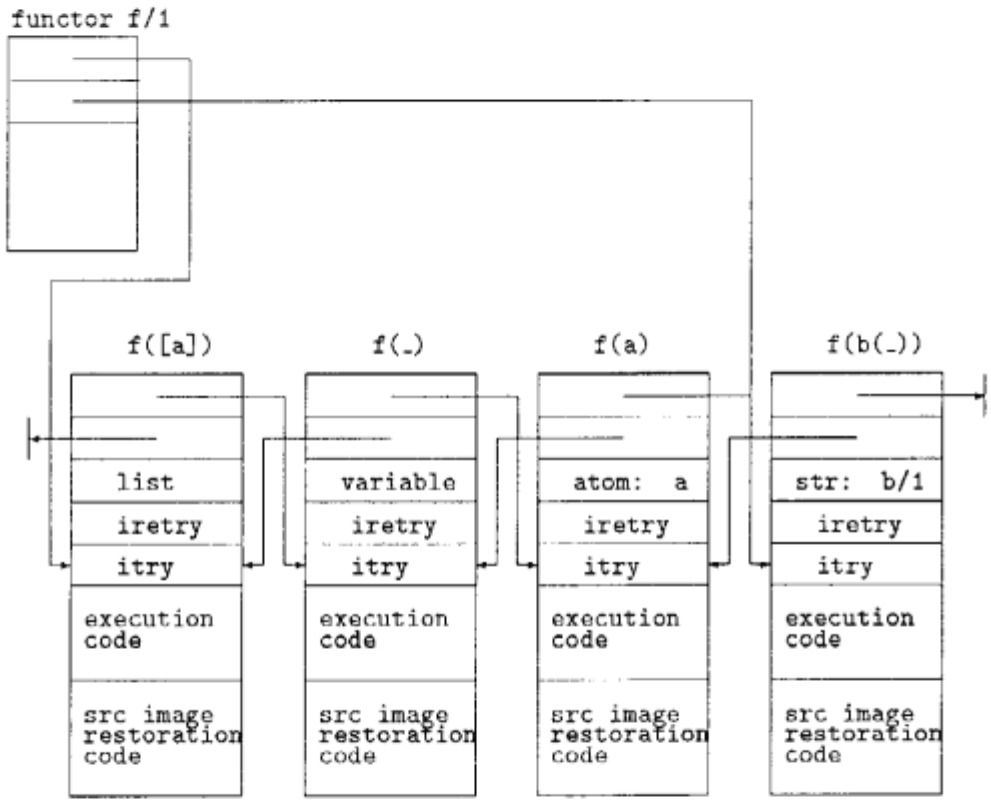

Figure 1: Management of Dynamic Predicates

indexing

In case of static predicates, which is a technique to avoid redundant execution of clauses, does indexing using hash tables. In case of dynamic predicates, indexing is done sequentially, checking indexing information of chained clauses.

Choice point frame allocation

Static predicates allocate choice point frames only if necessary. Dynamic predicate always allocates a choice point frame.

The execution of a dynamic predicate is briefly shown below:

1. A predicate call sets the current execution mode flag. This flag indicates, which of execution code or source image restoration code should be executed.

2. When a dynamic predicate is called, the entry instruction `itry` of the first clause is executed.

3. The `itry` instruction searches a candidate. It traces chained clauses, checking if the indexing information of the clause matches with the first argument.

4. If a candidate is found, execute its execution code or source image restoration code, according to the execution mode flag. Else, fail.

5. If the candidate fails, the retry instruction `iretry` is executed. It searches the next alternative, just like `itry` instruction does, and go to step 3.

Figure 2: Structure of a dynamic clause

The structure of each dynamic clause is shown in figure 2.

Indexing Information

The indexing information slot holds the type information of the first argument, and is referred by entry/retry instruction.

Entry/Retry Instruction

These instructions control execution over chained clauses. Entry instruction(`itry`) is the instruction, which is executed for the first time, when a dynamic predicate is called. Retry instruction(`iretry`) is the instruction, which is executed when a dynamic clause fails. The details of these instructions are explained in 3.2.2.

Execution Code

This is the code, which is executed when this clause is called. It starts from the next address of entry instruction. The code generation scheme is the same as WAM.

Source Image Restoration Code

This is the code, used to restore source image of a clause. If the clause has the form:

```
Head :- Bodys.
```

then its source image restoration code is obtained by imaginary creating a unit clause

```
clause(Head,Body,Clause).
```

```
concat([X|Y],Z,[X|YZ]) :- concat(Y,Z,YZ).
```

Execution code		source image restoration code	
glis	a0	gstr	a0,concat/3
urvr	a4	urvr	a0
urvr	a5	urvr	a3
grvr	a1,a6	urvr	a4
glis	a2	glis	a0
urvl	a4	urvr	a0
urvr	a7	urvr	a5
prvl	a0,a5	glis	a4
prvl	a1,a6	urvl	a0
prvl	a2,a7	urvr	a6
exec	concat/3	gstr	a1,concat/3
		urvl	a5
		urvl	a4
		urvl	a6
		gcon	a2,clause
		prcd	

Figure 3: Code for the second clause of concat/3

and compiling the unit clause. The location of this code is recorded in the operand of the entry instruction itry.

Figure 3 shows an example of an execution code and a source image restoration code for the second clause of well known concatenate predicate. The execution of execution code/source image restoration code is controlled by the flag explained in 3.2.1.

3.2 Instruction to Support Dynamic Predicate Execution

For the purpose of efficient dynamic predicate execution, we have extended the instruction set of CHI.

3.2.1 Execution Mode Flag

We have prepared a flag, which shows the current execution mode. This flag is managed by various call instructions. If this bit is on, then it indicates that the execution code of dynamic clauses should be executed. If this bit is off, then it indicates that the source image restoration code of dynamic clauses should be executed.

On CHI, this flag is assigned to a special hardware register bit. Since handling of the bit can be done in a micro step, parallel to other micro operations, there is no overhead with regards to mode handling.

3.2.2 Clause Execution Control Instructions

To control execution of dynamic clauses, we introduced two instructions, which are extensions to WAM's choicepoint control instructions `try_me_else`, `retry_me_else`, with regards to mode handling and the indexing on the first argument.

These instructions are the essence of our dynamic predicate implementation.

`itry`

This is an instruction, which is executed upon entry to a dynamic predicate. First, it checks execution mode flag, and in case of execution mode, does following jobs.

1. find a candidate clause, whose indexing information matches with the first argument.
2. if such a candidate does no exist, then fail.
3. else, create a choice point to prepare failure. The current execution mode flag is also saved in the choice point frame.
4. finally, execute the execution code of the candidate clause.
5. If the execution of the candidate clause fails, then the `iretry` instruction of the candidate clause is executed, which tries to execute the next alternative clause.

Figure 4 shows a moment, which is about to finish `itry` instruction execution. Predicate `f/1` has just been called with the first argument `g(10)`, and `itry` instruction of the first clause is being executed. The `itry` instruction first finds a clause such that the indexing information is `g/1` structure. So the `itry` instruction has made a choice point frame, such that the `iretry` instruction of the second clause is executed upon failure. And the new PC points to the execution code of the second clause.

In case of source mode, `itry` instruction does the following.

1. find a candidate clause such that the first element of the first argument matches with the index information of the clause.
2. if such a candidate does no exist, then fail.
3. else, create a choice point and execute the source image restoration code of the candidate clause.

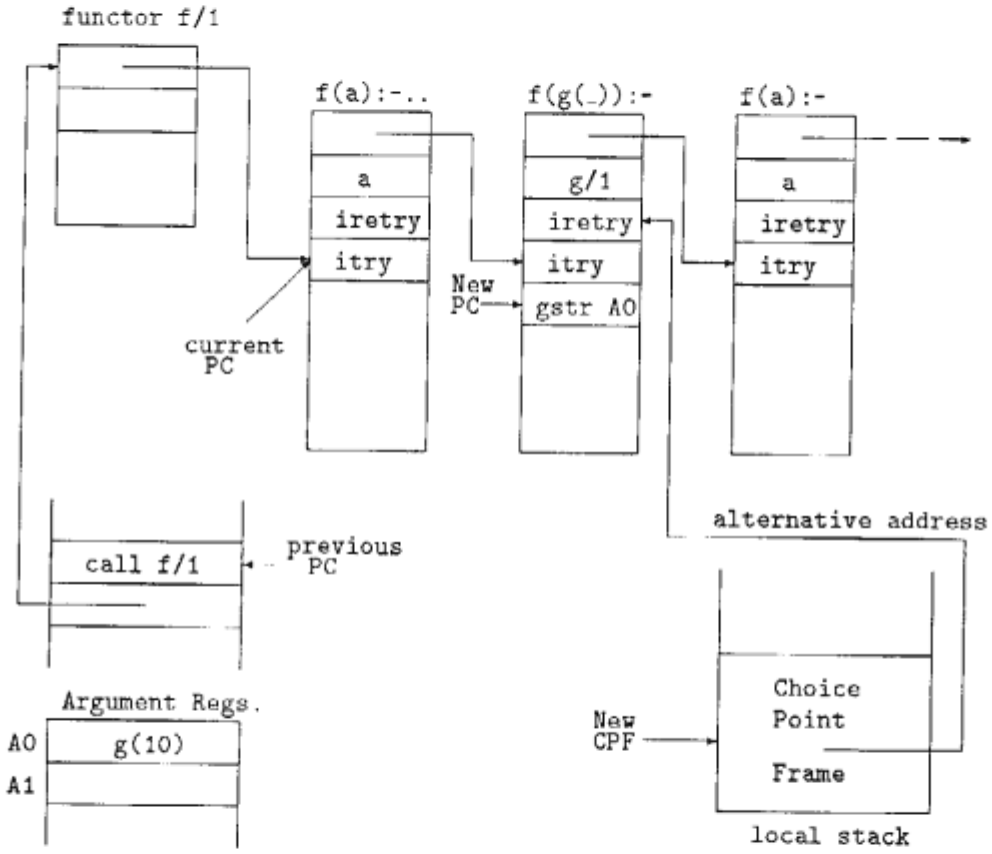

Figure 4: Calling `f/1`

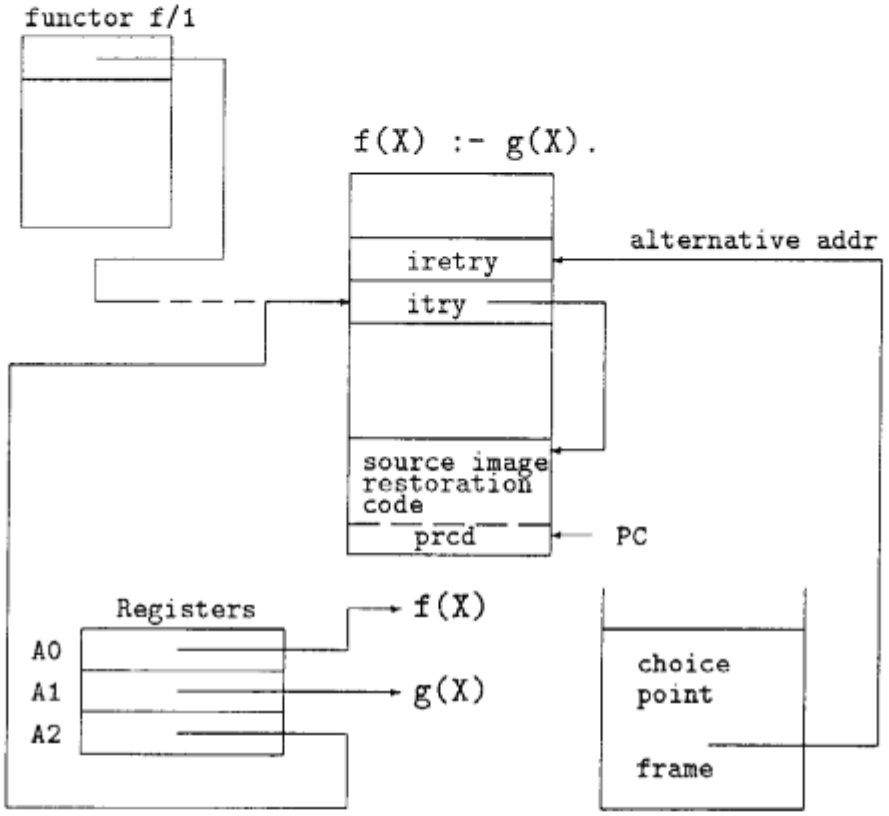

Figure 5: Execution of source image restoration code

4. if the execution of the candidate clause fails, then execute `iretry` instruction of the candidate clause.

Figure 5 shows a moment, where the source mode restoration execution of a candidate clause has just succeeded. Registers A_0, A_1 and A_2 are unified with the head goal image of the clause, the body goal image of the clause and the pointer to the code object, respectively.

`iretry`

This instruction is executed when the execution of a dynamic clause fails. It first backs up various control information, including current execution mode flag, and does the following jobs in case of execution mode.

1. find the next candidate clause, starting from the next clause of the current clause.
2. if no candidate clause exists, deallocate current choicepoint frame and fail.
3. else, modify the current choicepoint frame, and execute the execution code of the candidate clause.
4. if the execution of the next candidate clause fails, `iretry` instruction of this current candidate clause is executed.

The action of `iretry` instruction at source restoration mode is easy to infer, and is omitted here.

3.2.3 Source Image Restoration Instructions

In order to implement source image restoration predicate `clause/3` and `instance/2`, we introduced following two instructions.

predicate	instruction
clause	exec_clause
instance	exec_instance

4 Evaluation

In this section, we evaluate the performance of dynamic clause compilation and how they affect in real application program.

Table 1: Speed of dynamic clause assertion

data	`assert` (msec)	`encode` (msec)	`assert/encode`
1	1.49	0.17	8.76
2	1.93	0.19	10.16
3	4.14	0.31	13.35
4	2.09	0.25	8.36
5	3.26	0.35	9.31

4.1 Performance of Dynamic Clause Compilation

First we evaluated the performance of dynamic clause compilation, from the view points of compilation time, object code size and execution speed.

As a comparison, we used predicates `encode` and `decode`. `encode` freezes a term on the global stack, and create a corresponding heap data. `decode` melts a heap data and creates a term on thc global stack area. These are used to implement dynamic predicates in conventional

We used five simple clauses as follows:

```
1  foo(a).
2  foo(X,X).
3  foo([X|Y],f(X),g(Y)).
4  foo(X) :- bar(X).
5  foo(X) :- bar(X),baz(X).
```

Table 1 compares the speed of `assert` and the speed of `encode`. In average, `assert` is 10 times slower than `encode`. The reason is that CHI's dynamic clause compiler is written in Prolog, whereas `encode` is written in low level machine instructions, and are very carefully tuned.

Table 2 shows comparison of the the object size. It shows that `assert` consumes approximately 60% more memory than `encode`. However, we find

Table 2: Code Size of Dynamic Predicate

data	`assert` (words)	`encode` (words)	`assert/encode` .
1	21	12	1.75
2	21	13	1.62
3	38	20	1.9
4	22	17	1.29
5	34	22	1.55

Table 3: Execution Speed of Dynamic Predicate

data	assert (msec)	decode (msec)	assert/decode
1	0.036	0.095	0.38
2	0.037	0.110	0.34
3	0.043	0.163	0.26

that the problem of global stack consumption in application program is much more critical compared to heap area consumption. So, We find such increase of memory consumption is permissible.

The comparison between the execution speed of a dynamic predicate, and the speed of `decode` is shown in table 3. We used only sample clauses 1 from 3, because clauses 4 and 5 have body goals, which makes the correct evaluation difficult. This shows that calling a dynamic predicate takes only 30% of decoding an encoded data.

However, this execution retio is measured by a predicate, consisting of only one clause. However, predicates have several clauses in application execution, so So calling dynamic predicate will be much faster than decode.

4.2 Effects of dynamic clause compilation in application program

To prove the effectiveness of our dynamic clause compilation method, we evaluated the performance of an application program.

The application is a bibliography information retrieval program, which consists of 2100 lines of program and 7700 lines of bibliography data. The application uses an inductive inference method based on the stochastic logic [9]. During program execution, the inductive inference method generates, tests and cancels hypothesises repeatedly. Dynamic predicates are used to represent such hypothesis management.

The flow of this program is as follows: First, the system gives a user, 10 bibliography information according to initial retrieval commands. After that, the user repeats following three operations, until he is satisfied with the bibliographies that the system gives.

Select bibliographies

The user tells the system, which bibliographies are necessary and which are not. This is called *select* phase.

Calculate a new retrieval command

According to the select phase information, the system calculates new retrieval command. This is the kernel phase of the system, and we call it *think* phase.

Find new bibliographies

According to the new retrieval command obtained in *think* phase, the system newly retrieves 10 bibliographies and gives them the user. This is called *find* phase.

Of the three phases explained above, we analyzed *think* and *find* phase, since *select* phase does nothing but interactions with the user. These two operations include no I/O operations.

As stated before, this application repeatedly generates/cancels hypothesises. To represent hypothesises, the application uses `record`. `record` can be regarded as a special dynamic predicate, that the system provides. Its rough implementation is as follows:

```
redorda(X,Y,Z) :- asserta('Functor for record'(X,Y),Z).
redordz(X,Y,Z) :- assertz('Functor for record'(X,Y),Z).
recorded(X,Y,Z) :- clause('Functor for record'(X,Y),_,Z).
```

The difference between records and ordinary predicate is as follows:

- Adding records is 30% faster than adding clauses.
- Object code for records is approximately 30% smaller than objects codes for ordinary dynamic predicates.
- calling records is approximately 30 % slower than ordinary predicates.

First, we measured how many times record manipulating predicates such as `recorda`, `recordz` and `recorded` are called. The result is shown in table 4. This tells that the record reference frequency is 8 times higher than record registration.

We then redefined `recorda`, `recordz` and `recorded` using `encode` and `decode`, to evaluate the time spent for adding records, the object size required for adding records and the time spent for calling records. The results are shown in table 5, table 6 and table 7, respectively.

Table 4: Predicate call counts

phase	recorda/z	recorded
think	388	4268
find	237	827

Table 5: Speed of dynamic clause assertion in application

phase	recordaz (msec)	encode (msec)	recordaz/encode	assert' (msec)	assert'/encode
think	1180	196	6.02	1534.0	7.9
find	783	120	6.53	1017.9	8.48

In the tables, the recordaz field shows the sum of the results of recorda and recordz. assert' field shows the estimated value, if we use ordinary dynamic predicates instead of records.

As for the time and memory consumption, the result is consistent with the result of the result of simple data. However, if we compare the result of calling records with that of simple data, we can see that the execution speed of calling records is much faster. This tells that records have several alternatives, and

Table 6: Code size of dynamic predicates at application program

phase	recordaz (words)	encode (words)	recordaz/encode	assert' (words)	assert'/encode
think	18540	13093	1.42	24102	1.84
find	14508	10857	1.34	18860	1.30

Table 7: Execution speed of dynamic predicates at application program

phase	recordaz (msec)	encode (msec)	recordaz/encode	assert' (msec)	assert'/encode
think	3760	52803	0.071	2892.3	0.055
find	586	8263	0.071	450.8	0.055

Table 8: Execution speed of application program

phase	recordaz (msec)	encode (msec)	recordaz/encode	assert' (msec)	assert'/encode
think	10899	58958	0.18	10394.3	0.17
find	2214	9230	0.24	2313.7	0.25

`itry` and `iretry` instruction successfully controls the execution over chained dynamic clauses.

Finally, the time spent for total time for spent application execution is shown in table 8. The result shows the effectiveness of the dynamic clause compilation method. Although assert process itself is slow, execution of compiled clauses is very fast. And execution of dynamic compiled clauses is much more frequent than dynamic clause compiling, so the dynamic clause compilation greatly contributes to the fast execution of application programs.

5 Conclusion

Dynamic Clause Compilation is proposed to accelerate dynamic predicate execution. And its implementation on CHI and evaluation results are de-

scribed. The measurement results using practical Prolog application shows that the average execution speed of dynamic predicates is 14 times faster than conventional Prolog implementation. And this result shorten the total execution time 1/3 to 1/5, in spite of the slow execution speed of assert. Since modification of database or rules is essential in practical Prolog applications, especially in AI applications, speed up of dynamic predicates is very important. The proposed dynamic clause compilation will have large contribution to practical use of logic programming.

Acknowledgments We would like to thank Dr. Shunichi Uchida (ICOT) for support this project, to Shinji Yanagida, Satoshi Oyanagi (NSIS) in implementation of CHI system and to all the members who are engaged in CHI project. We would also like to thank Kazuhiko Ohno, who was kind enough to allow us to use his bibliography retrieval program. Thanks are also expressed to Nobuhiko Koike and Dr. Tatsuo Ishiguro (NEC) for their continuous encouragement, valuable advice and support.

References

[1] Buettner, K.A., "Fast Decompilation of Compiled Prolog Clauses", in *Proc. the Third International Conference on Logic Programming*, New York, 1987

[2] Clocksin, W.F., "Implementation Techinquies for Prolog Databases", Software — Practice and Experience, Vol. 15(7), 1985

[3] Clocksin, W.F., "Design and Simulation of a Sequential Prolog Machine", New Generation Computing, Vol. 3, 1985

[4] Habata, S., Nazakaki, R., Konagaya, A., Atarashi, A. and Umemura, M., "Co-operative High Performance Sequential Inference Machine: CHI", in *Proc. ICCD'87*, New York, 1987

[5] Konagaya, A., "Implementation and Evaluation of a Fast Prolog Interpreter", in IPC Japan SIG-SYM 46-4, 1988 (in Japanese)

[6] Konagaya, A., Habata, S., Atarashi, A., Yokota, M. "Evaluation of Sequential Inference Machine CHI" to appear in North American Conference on Logic Programming, 1989

[7] Lindholm T.G., O'Keefe R.A., "Efficient Implementation of a Defensible Semantics for Dynamic Prolog Code", in *Proc. the Fourth International Conference on Logic Programming*, Melbourne, 1987

[8] Nakazaki, R., Konagaya, A., Habata, S., Shimazu, H., Umemura, M., Yamamoto, M., Yokota, M. and Chikayama, T., "Design of a High-speed Prolog Machine (HPM)", in *Proc. of the 12th International Symposium on Computer Architecture*, 1985

[9] Ohno, K. "Document Retrieval using Inductive Inference Method on Stochastic Logic", in SIG-FAI-8803-2, 1988 (in Japanese)

[10] "Quintus Prolog Reference Manual (Version 10)", Quintus Computer Systems, Inc.,1987

[11] Warren, D.H.D, "AN ABSTRACT PROLOG INSTRUCTION SET", Technical Note 309, SRI International, 1983