Multi-Threading Inside Prolog for Knowledge-Based Enterprise Applications

Masanobu Umeda, Keiichi Katamine, Isao Nagasawa, Masaaki Hashimoto, and Osamu Takata

Graduate School of Computer Science and Systems Engineering,
Kyushu Institute of Technology, Iizuka, Fukuoka, Japan
umerin@ci.kyutech.ac.jp

Abstract. A knowledge-based system is suitable for realizing advanced functions that require domain-specific expert knowledge in enterprise-mission-critical information systems (enterprise applications). This paper describes a newly implemented multi-threaded Prolog system that evolves single-threaded Inside Prolog. It is intended as a means to apply a knowledge-based system written in Prolog to an enterprise application. It realizes a high degree of parallelism on an SMP system by minimizing mutual exclusion for scalability essential in enterprise use. Also briefly introduced is the knowledge processing server which is a framework for operating a knowledge-based system written in Prolog with an enterprise application. Experimental results indicated that on an SMP system the multi-threaded Prolog could achieve a high degree of parallelism while the server could obtain scalability. The application of the server to clinical decision support in a hospital information system also demonstrated that the multi-threaded Prolog and the server were sufficiently robust for use in an enterprise application.

1 Introduction

Advanced functions that utilize domain-specific expert knowledge are needed for enterprise-mission-critical information systems (hereinafter called enterprise applications) such as hospital information systems and logistics management systems. Clinical decision support [1] for preventing medical errors and order placement support for optimal inventory management are such examples. A knowledge-based system is suitable for realizing such functions because it can incorporate a knowledge base in which domain-specific expert knowledge is systematized and described.

Production systems in combination with Java technology [2] have been studied as a means to apply a knowledge-based system to an enterprise application [3–5]. They provide possibilities of improving the development and maintenance of an enterprise application to separate business rules, which are repeatedly updated, from workflow descriptions, which are rarely updated. However, certain issues involved in applying a production system to large business rules, that is, side effects, combinatorial explosion, and control saturation [6], have not been

sufficiently resolved in these systems. On the other hand, Prolog, which is suitable for knowledge processing, in combination with Java technology has also been studied with the aim of advancing the development of information systems [7–11]. However, there remain unresolved issues of scalability and transaction processing which are essential to enterprise applications.

The authors have been developing an integrated development environment called Inside Prolog [12], which is dedicated to knowledge-based systems. Various knowledge-based systems, such as design calculation support systems [13–15] and health care support systems [16, 17], have been developed using Inside Prolog and put to practical use [18]. Inside Prolog provides standard Prolog functionality, conforming to ISO/IEC 13211-1 [19], and also provides a large variety of Application Programming Interfaces (APIs) which are essential for practical application development. These features allow the consistent development of knowledge-based systems from prototypes to practical use. It has been, however, difficult to apply Inside Prolog to an enterprise application as is, because only a stand-alone system was within the scope of Inside Prolog.

Therefore, in order to cope with the scalability issue, the authors initially developed a new Prolog system that was capable of multi-threading by evolving Inside Prolog. The authors then developed the knowledge processing server [20] for operating various knowledge-based systems with enterprise applications by combining this multi-threaded Prolog system with Java technology. The knowledge processing server has been practically applied to clinical decision support [21, 22, 20] in the hospital information system CAFE [23, 24], and it enables validation of contraindications within diseases, drugs, and laboratory results, suggestion of the quantity and administration conditions of a medication order, and the summarization of clinical data such as laboratory results.

This paper describes an overview of Inside Prolog and its multi-thread extension for enterprise use. The knowledge processing server is then briefly introduced. Finally, the multi-thread feature and parallelism of multi-threaded Inside Prolog, and the scalability of the knowledge processing server are evaluated.

2 Overview of Inside Prolog

Inside Prolog is an ISO/IEC 13211-1 compliant Prolog system with various extensions. It is developed over the Prolog abstract machine TOAM, which is based on WAM [25]. Figure 1 illustrates the system architecture of Inside Prolog. It provides several optimization features, such as unification optimization using the matching tree [26] and the translation of determinate predicates to C functions [27]. It also provides advanced APIs, which are required for the development of practical applications, and integration APIs, which are required for integration with existing information systems, in a uniform platform-independent manner [1]. These features allow the consistent development of knowledge-based systems from prototypes to practical use using one programming language, Prolog.

[1] Platform-specific functions, such as OLE on Windows, are designed so as to eliminate platform differences from an application program by providing minimal libraries.

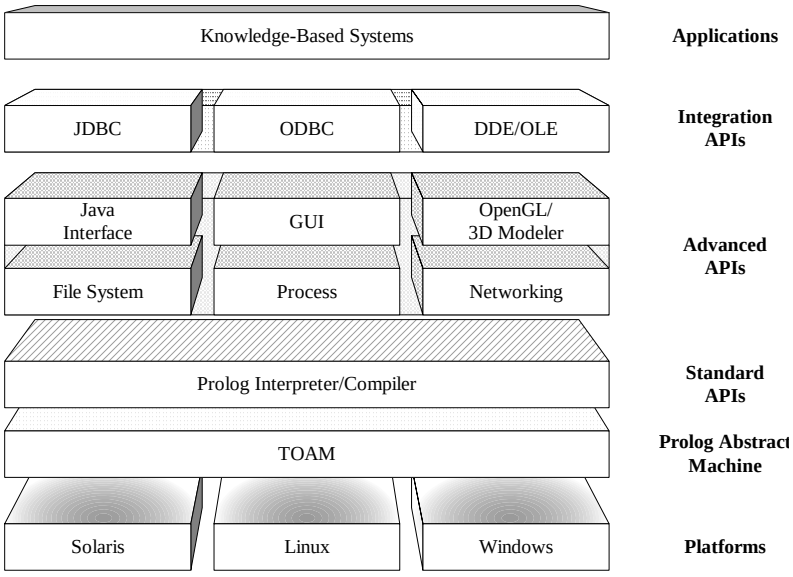

Fig. 1. Architecture of Inside Prolog

The following subsections briefly introduce several topics related to a multi-thread extension of Inside Prolog.

2.1 Memory Model

TOAM has three stacks - a control stack, trail stack, and global stack (or heap) - and a data area (or atom area) for storing global data such as predicate definitions. The data area is divided into two, i.e., a persistent area and a transient area. Data that are never modified dynamically, such as an inference engine and GUI of a knowledge-based system, are stored in the persistent area, while data that can be modified dynamically, such as a knowledge base and execution results, are stored in the transient area. The separation of the data area has the following advantages. That is, the performance of garbage collection can be improved due to the exclusion of the persistent area from its target; the reliability of a system can be improved due to the prevention of unexpected modifications while further optimizations are possible using the immutability of program locations as described in Sect. 2.3.

2.2 Program Code Representation

The components of a knowledge-based system can be classified into several categories. Immutable programs such as an inference engine and GUI, mutable data such as inference rules and clinical information, and programs generated dynamically from these data are such examples. These components can be expressed

in a uniform representation of a program code because Prolog's programs and data can be handled in the same manner. However, a program in each category has both advantages and disadvantages if a uniform representation is used. For example, if a program generated from an inference rule, as well as an inference engine are optimized, the program's execution speed can be improved; however, its optimization costs result in the inconvenience of the interactive debugging of inference rules. Therefore, Inside Prolog allows the choice of the most suitable representation of a program code from the following according to its role and scene in an application, and thus enables the development of practical knowledge-based systems.

Static program A static program is represented as a bytecode generated by an optimizing compiler [26]. It is appropriate to a static predicate whose execution speed is important, and one that is never modified while an application is running. A static program is stored in either the persistent area or the transient area according to a directive.

Native program A native program is a kind of a built-in predicate represented as C functions that are translated from a determinate predicate [27]. It is appropriate to a static predicate whose execution speed is strongly important, and one whose definition is rarely changed. Although the translation is automatic, its use is limited because the object binaries rely on the platform.

Incremental program An incremental program is represented as a bytecode [2] generated by an incremental (non-optimizing) compiler invoked by `asserta/1` and `assertz/1`. It is appropriate to a dynamic predicate that is defined and executed dynamically. In such a case, the balance between the time required for each is important. The compilation is speeded up by the omission of the optimization while the execution is speeded up by the omission of the logical database update [19].

Interpretive program An interpretive program is represented as a term that is interpreted by the Prolog interpreter. It is appropriate to a dynamic predicate which is defined and executed dynamically in the manner of an incremental program, but the compilation has little effect on execution time. In the case of a unit clause composed of ground terms, the same effect as that of structure copying [28] can be expected because terms are shared without being copied to the global stack. For example, it is appropriate to clinical information on drug-drug interactions [21, 22] and engineering information regarding product catalogues [29] because they can be represented as a set of unit clauses composed of ground terms.

2.3 Optimization by Instruction Rewriting

On the execution of a predicate, the symbol table is repeatedly referred to in order to determine the program code associated with a predicate name. The time required for referring to the table once is very short, but the cumulative time is

[2] It also has a term representation for `clause/2`.

not negligible if the same predicate is executed repeatedly. Therefore, predicate call instructions, such as `call` and `execute`, are optimized by rewriting these instructions according to the type of a program code being called. For example, if a predicate being called is a static program stored in the persistent area, a predicate call instruction `call` to this predicate is rewritten as a direct call instruction `call_direct` with an absolute address because the location of a static program is fixed in the persistent area. Likewise, a call instruction to a native program is rewritten as `call_native`, and others as `call_indirect` that refers to the symbol table. Thus, instructions that refer to the symbol table are limited to only a few instructions such as `call_indirect`.

3 Multi-Thread Extension for Enterprise Applications

This section describes a multi-thread extension of Inside Prolog for expanding its application domains to enterprise applications.

3.1 Multi-Threading Prolog

Several approaches are known to realize multi-threading in Prolog. The first is to realize scheduling and context switching in a Prolog abstract machine by itself [7, 30]. The second is to utilize a standard multi-thread library for scheduling and context switching [9, 8]. In the case of Java-based implementation, the third is to create multiple single-threaded Prolog engines, and run them in multiple Java threads [10, 31]. The first approach is a kind of user-level thread model, and has an advantage in performance because kernel resources are not consumed, and context switching and synchronization can be simplified. It is, however, difficult to utilize the multiple processors of an SMP system [32]. The second approach has disadvantages in regard to the costs of context switching and synchronization. However, the throughput can be improved by parallel processing on an SMP system. The third approach has advantages in parallel processing over other approaches, but it is inadequate for a large knowledge base because the data area cannot be shared between threads.

Inside Prolog adopts the second approach so that it can deal with a large knowledge base, and has advantages in throughput improvement on an SMP system. The POSIX thread library of Unix and Linux, and the Windows thread library can be used as the multi-thread library.

3.2 Execution Model

Figure 2 illustrates the execution model of multi-threaded Inside Prolog [3].

[3] Hereinafter, a multi-threaded version of Inside Prolog is also called Inside Prolog only when there is no possibility of confusion.

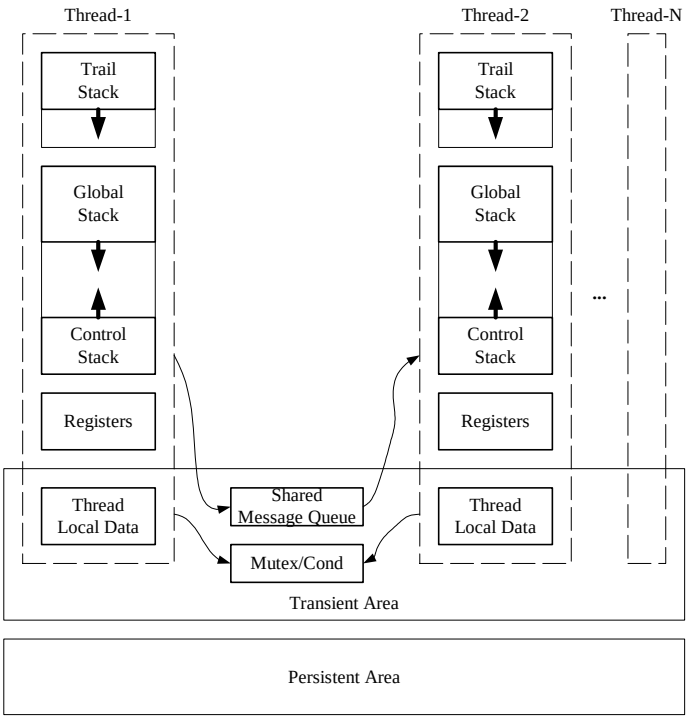

Fig. 2. Execution model of multi-threaded Inside Prolog

Shared variables Shared variables between threads require major changes of the abstract machine TOAM regarding the handling of backtracking and its organization [8]. Therefore, communication mechanisms using shared variables between threads are omitted.

Communication Shared message queues created in the transient area are used for communication between threads. The shared message queue is capable of sending and receiving messages in multi-thread safety. It is also possible to send an interrupt message as an exception from one thread to another.

Synchronization Mutual exclusion object (mutex), condition variable (cond), and read/write mutex based on POSIX threads are provided for the basic synchronization mechanism. Mutex, cond, and read/write mutex are created in the transient area, and are shared between threads.

Thread-local data Globally shared data that should not be reclaimed on backtracking are usually kept in the data area by associating them with the symbol table using `asserta/1`, `assertz/1` and so on. However, the risk of contention exists because the symbol table is shared by all threads. The thread-local data is provided for managing shared data specific to a thread.

3.3 Extension of Prolog Abstract Machine

Major changes of TOAM for realizing the multi-thread feature include the introduction of thread control data and synchronization [4]. The thread control data manages the state of multi-threaded TOAM per thread, such as stacks and registers; in single-threaded TOAM these are managed using global variables. Thread control data can be implemented using the thread local storage [5] of a standard multi-thread library. On the other hand, even though synchronization is unavoidable when accessing the symbol table and the data area, heavy use of synchronization may cause significant performance degradation, and multiple processors of an SMP system cannot be utilized effectively if the length of mutual exclusion is long.

There are three cases that require synchronization in TOAM. That is, the handling of `catch/3` and `throw/1` that deal with exceptions saved in the transient area; the rewriting of predicate call instructions such as `call`; and the handling of predicate call instructions that refer to the symbol table, such as `call_indirect`. The first case does not affect usual inference performance and parallelism because it happens only when exceptions are thrown. The influence of the second case must be negligibly small because synchronization is required only once for each instruction occurrence. On the other hand, the third case invokes a program stored in the transient area, and this invocation procedure consists of several inseparable steps. Therefore, synchronization is generally required so that the definition and execution of a predicate can be performed safely under a multi-threaded environment, though synchronization seriously affects inference performance and parallelism. In case the definition and execution of a predicate are performed in parallel, it is customary to ensure the consistency of a predicate definition using explicit synchronization by an application program itself, as described in Fig. 3 of Sect. 3.4. Consequently, the omission of synchronization by TOAM is less likely to become a practical issue.

Therefore, synchronization regarding static programs and incremental programs is omitted by TOAM in order to give priority to inference performance and parallelism. In contrast, interpretive programs are synchronized by `clause/2`, `assertz/1` and so on for ensuring the consistency of hash tables for clause indexing and preserving the logical database update. This allows the choice of the most suitable representation of a program code according to its role and scene from the viewpoint of inference performance and parallelism, and the consistency of a predicate definition.

3.4 Examples of Multi-Thread Programming

Figure 3 shows a programming example of the producer-consumer problem written in Inside Prolog. A condition variable is created by `cond_create/1` for sus-

[4] Built-in predicates that access the symbol table and the data area also must incorporate synchronization.

[5] For example, it is provided by `pthread_getspecific()` and its family of POSIX threads and `TlsGetValue()` and its family of Windows.

pension and resumption of threads, and a mutual exclusion object is created by `mutex_create/1`. Consumer and producer threads that execute `consumer/2` and `producer/2` predicates, respectively, are created by `thread_create/3`. A buffer shared by these two threads is represented by `buffer/1`, and its contents are updated by `assertz/1` and `retract/1` with synchronization using `with_mutex_lock/2`. Threads are suspended by `cond_wait/2` when the buffer is empty or full, and are resumed by `cond_signal/1` when the state of the buffer is changed.

4 Knowledge Processing Server

The knowledge processing server is a framework for operating a knowledge-based system written in Inside Prolog with an enterprise application, and for providing inference services to an enterprise application using a knowledge base. It is independent of any enterprise application and any knowledge-based system, and is realized by combining Inside Prolog and Java. The server improves interoperability with various enterprise applications due to its adaptation to distributed object technology, such as RMI, SOAP, and CORBA, using Java. The server also makes it easier to incorporate a knowledge-based system into a transaction system by allowing a knowledge-based system to inherit the transactions of an enterprise application in the J2EE environment.

Figure 4 illustrates a simplified system configuration of the server that applies a knowledge-based clinical decision support system to the hospital information system CAFE in a J2EE environment. The clinical knowledge base stores medical inference rules used for clinical decision support such as the validation of contraindications and proposals of appropriate administration conditions, while the clinical database consistently stores patient records such as the disease names and medication orders of patients. The EJB client provides clinical support functions to health care professionals using an application which handles workflow in a hospital, and the clinical decision support system. The session bean is a service interface to clinical decision support functions provided by the knowledge processing server. The inference engine for clinical decision support is a knowledge-based system written in Prolog. The knowledge base adaptor is a Prolog program that fits data types and data structures used in a service interface into an inference engine and a knowledge base, and vice versa. The external data interface is used to access external data, such as patient records in the clinical database, from the inference engine via EJB/JNDI services. The Prolog server is a generalized mechanism that mediates communications between a service interface written in Java and a knowledge-based system written in Prolog.

5 Performance Evaluation

This section presents the evaluation results for the multi-thread feature and parallelism of Inside Prolog, and the scalability of the knowledge processing server.

```
producer_consumer :-
   cond_create(Cond),      % Create a condition variable
   mutex_create(Mutex),    % Create a mutual exclusion object
   %% Create a producer thread, and call producer/2.
   thread_create(Producer, producer(Cond, Mutex), []),
   %% Create a consumer thread, and call consumer/2.
   thread_create(Consumer, consumer(Cond, Mutex), []).

producer(Cond, Mutex) :-
   repeat,
   produce_item(Item),
   with_mutex_lock(Mutex,
    (%% Wait until the buffer has a vacant.
     (buffer(Items0), length(Items0, 100) ->
      cond_wait(Cond, Mutex) ; true),
     %% Add an item to the buffer.
     retract(buffer(Items)),
     append(Items, [Item], Items1),
     assertz(buffer(Items1)),
     %% Notify a consumer when the buffer becomes non-emtpy.
     (Items1 == [Item] -> cond_signal(Cond) ; true)
    )),
   fail.

consumer(Cond, Mutex) :-
   repeat,
   with_mutex_lock(Mutex,
    (%% Wait until the buffer becomes non-empty.
     (buffer([]) -> cond_wait(Cond, Mutex) ; true),
     %% Take out an item from the buffer.
     retract(buffer([Item | Items])),
     assertz(buffer(Items)),
     %% Notify a producer when the buffer has a vacant.
     (length(Items, 99) -> cond_signal(Cond) ; true)
    )),
   consume_item(Item),
   fail.

%% An initial value of the buffer is empty.
buffer([]).
```

Fig. 3. Programming example of the producer-consumer problem

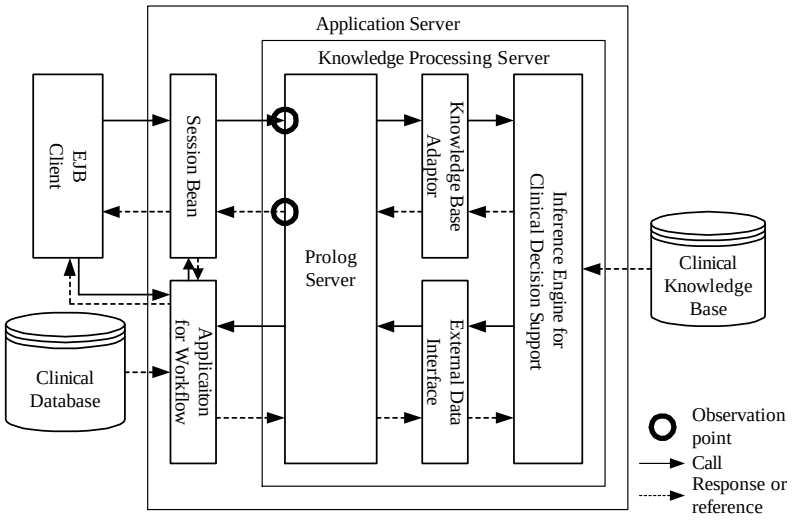

Fig. 4. System configuration of the knowledge processing server applied to a J2EE-based hospital information system

5.1 Overhead Costs of the Multi-Thread Extension

In order to evaluate the overhead of the multi-thread extension, the elapsed times of single- and multi-threaded versions were compared using benchmark programs boyer, 8 queens, qsort, and takeuchi. Interpretive, incremental, and static program code representations were applied. Static programs were stored in the persistent area and executed once to obtain normal performance by forcing the instructions to be rewritten before the measurement. Sun V880 with 1 CPU was used in this experiment.

Figure 5 shows the elapsed time ratios of the multi-threaded to the single-threaded version. The results indicate that the overhead costs of the multi-thread extension are about 20 % at its maximum. These costs are due to synchronization and representation changes of the TOAM state from global variables to pointer accesses through the thread control data, as in SWI-Prolog [9].

5.2 Parallelism on SMP Systems

In order to evaluate the parallelism of each program code representation on an SMP system, the elapsed times of the benchmark programs were measured using a Sun V880 with various CPU configurations. Each of the benchmark programs was executed in parallel. The number of CPUs varied from 1 to 4, and the number of threads that execute the programs in parallel varied from 1 to 8.

Figures 6 and 7 show the results of the interpretive and static programs of 8 queens. Elapsed times are normalized by the elapsed time in the case of one thread for each CPU configuration. The elapsed times of the interpretive

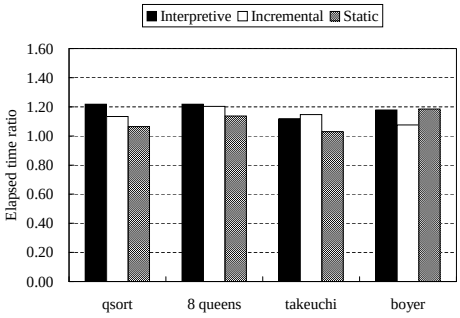

Fig. 5. Elapsed time ratios of multi-threaded version to single threaded version

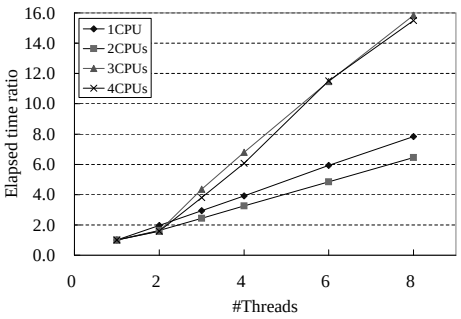

Fig. 6. Elapsed time ratios of the interpretive program of the 8-queens benchmark

program were increased as the number of threads increased. Elapsed times were also increased if the number of CPUs was greater than two. This is because an interpretive program requires synchronization of the transient area as described in Sect. 3.3. In contrast, the elapsed times of the static program decreased as the number of CPUs increased when the numbers of threads were the same. Especially, when the number of threads was equal to or smaller than that of CPUs, the elapsed times did not increase even if the number of threads increased. This is because synchronization is unnecessary for a static program after the instruction rewriting. Results similar to those of a static program were obtained for an incremental program except for real elapsed times due to optimization differences. The results of qsort and takeuchi were almost the same.

On the other hand, the results of all program codes of boyer were similar to those of the interpretive program of 8 queens as shown in Figs. 8 and 9. This is because parallelism is decreased due to the synchronization caused by the heavy use of `functor/3` which accesses the symbol table.

These results indicate that the multi-thread extension is effective in parallel processing on SMP systems for incremental and static programs if predicates including synchronization are not used frequently.

5.3 Scalability of the Knowledge Processing Server

In order to evaluate the scalability of the knowledge processing server on an SMP system, the server was applied to a J2EE-based application, and the elapsed times of the inference service were measured against multiple clients. The experimental application was modeled upon the hospital information system CAFE, and its system configuration was similar to that shown in Fig. 4 except for an EJB client, the inference rules in the clinical knowledge base, and the entities stored in the clinical database.

Initially, a client creates twenty entity beans, whose class is defined for this experiment and has about ten fields, using an application server. A client then invokes the inference rules stored in the knowledge base through a session bean.

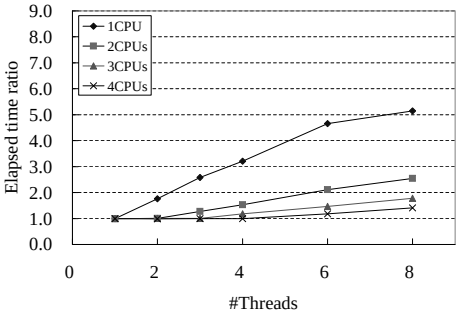

Fig. 7. Elapsed time ratios of the static program of the 8-queens benchmark

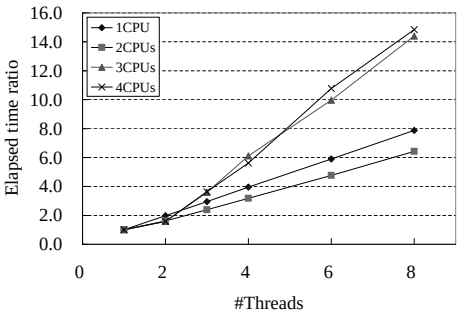

Fig. 8. Elapsed time ratios of the interpretive program of the boyer benchmark

The inference rules search for twenty entity beans that meet query conditions, and refer to the values of their fields.

The application server was deployed in a Sun V880 with from 1 CPU to 4 CPUs, and a database management system was deployed in a Sun Ultra60 with 2 CPUs. The number of threads that execute the inference engine was varied from 1 to 8. The number of clients varied from 1 to 32, and they were run on a maximum of eight machines. Elapsed times were measured at the point where a session bean invoked a method of the Prolog server (indicated as circles in Fig. 4).

Figures 10, 11, and 12 show elapsed times normalized by the elapsed time (about 0.027 seconds) in the case of one thread against one client. These results indicate that the elapsed times increased as the number of clients increased, but throughput speed was improved by increasing the number of threads and CPUs. For example, in the cases of 32 clients and 8 threads, the elapsed times of 2 and 4 CPUs cases were improved 0.51-fold (B in Fig. 11) and 0.28-fold (C in Fig. 12) over that of 1 CPU case (A in Fig. 10). The improvement ratios, however, did not reach the points estimated based on the number of CPUs, and they slowed down by degrees. Both the inference engine and the inference rules used in this experiment are represented as static programs [6], and the synchronization caused by these code representations is not included. Consequently, one reason for bounding scalability is the synchronization included in both the inference engine and the Java interface of Inside Prolog. However, it seems that this effect is sufficiently small because the elapsed times are not increased even on an SMP system of up to 4 CPUs, unlike boyer.

6 Conclusion

This paper describes a newly implemented multi-threaded Prolog system that evolves single-threaded Inside Prolog. It is intended as a means to apply a

[6] Inference rules are represented as incremental programs in a development phase and static programs in an operation phase.

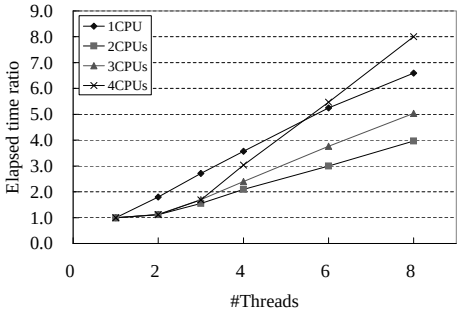

Fig. 9. Elapsed time ratios of the static program of the boyer benchmark

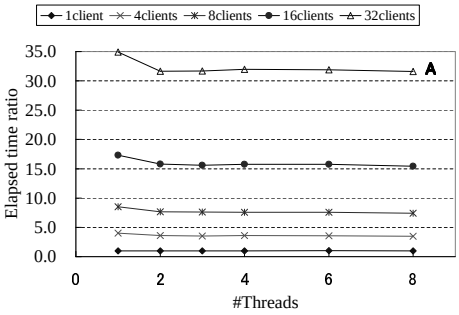

Fig. 10. Elapsed time ratios of the session bean of the knowledge processing server on 1 CPU machine

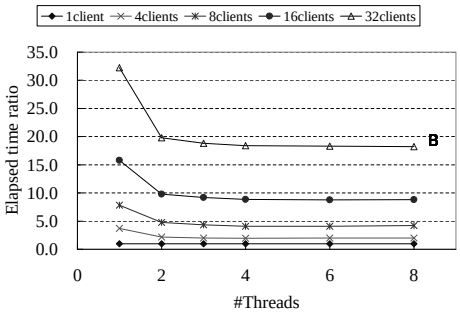

Fig. 11. Elapsed time ratios of the session bean of the knowledge processing server on 2 CPUs machine

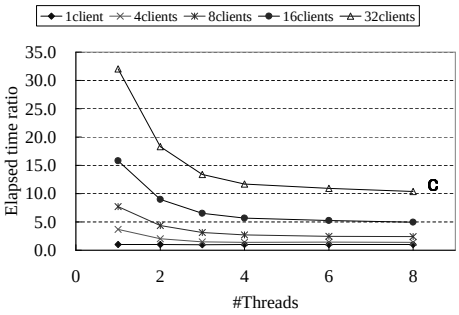

Fig. 12. Elapsed time ratios of the session bean of the knowledge processing server on 4 CPUs machine

knowledge-based system written in Inside Prolog to an enterprise application. It provides several representations of a program code, and allows the choice of the most suitable one according to its role and scene in an application. This allows the realization of a high degree of parallelism on an SMP system by minimizing mutual exclusion in the Prolog abstract machine TOAM. Also briefly introduced is the knowledge processing server which is a framework for operating a knowledge-based system written in Inside Prolog with an enterprise application.

The results of experiments using benchmark programs indicated that the overhead cost of the multi-thread extension was about 20 % at its maximum, and predicates represented as a bytecode could achieve a high degree of parallelism on an SMP system. The results of experiments regarding the knowledge processing server also indicated that the extension was effective for the improvement of throughput speed on an SMP system, and the server could obtain scalability on it.

The knowledge processing server has been practically applied to clinical decision support in the hospital information system CAFE. It processes about a

thousand prescription orders per day, and contraindications on one order are validated within about a second. The server has been problem-free for over a year. This indicates that Inside Prolog and the server are sufficiently robust for use in an enterprise application.

Because the application for the workflow of the hospital information system CAFE became too large to run in a 32 bits version of Java VM, a 64 bits version for workflow and a 32 bits version for the knowledge processing server were combined irregularly. It is necessary for Inside Prolog to support a 64 bits architecture in order to cope with large enterprise applications.

References

1. Kaplan, B.: Evaluating informatics applications – clinical decision support systems literature review. International Journal of Medical Informatics **64** (2001) 15–37
2. Toussaint, A.: Java rule engine api. JSR-94 (2003)
3. YASU Technologies http://yasutech.com/products/quickrulesse/index.htm: QuickRules. (2005)
4. ILOG, Inc http://www.ilog.com /products/jrules: ILOG JRules. (2006)
5. Drools Project http://drools.org: Drools. (2006)
6. Kobayashi, S.: Production system. Journal of Information Processing Society of Japan **26** (1985) 1487–1496
7. Eskilson, J., Carlsson, M.: SICStus MT – a multithreaded execution environment for SICStus prolog. In: Principles of Declarative Programming: 10th International Symposium. Volume 1490 of Lecture Notes in Computer Science., Springer-Verlag GmbH (1998) 36–53
8. Carro, M., Hermenegildo, M.: Concurrency in prolog using threads and a shared database. In: International Conference on Logic Programming. (1999) 320–334
9. Wielemaker, J.: Native preemptive threads in SWI-Prolog. In: Logic Programming. Volume 2916 of Lecture Notes in Computer Science., Springer-Verlag GmbH (2003) 331–345
10. Denti, E., Omicini, A., Ricci, A.: tuProlog: A light-weight prolog for internet applications and infrastructures. In: Practical Aspects of Declarative Languages. Third International Symposium, PADL 2001. Proceedings. Volume 1990 of Lecture Notes in Computer Science., Springer-Verlag GmbH (2001) 184–198
11. Tarau, P.: Jinni: Intelligent mobile agent programming at the intersection of java and prolog. In: Proceedings of the Fourth International Conference on the Practical Applications of Intelligent Agents and Multi-agent Technology. (1999) 109–124
12. Katamine, K., Umeda, M., Nagasawa, I., Hashimoto, M.: Integrated development environment for knowledge-based systems and its practical application. IEICE Transactions on Information and Systems **E87-D** (2004) 877–885
13. Tegoshi, Y., Nagasawa, I., Maeda, J., Makino, M.: An information processing technique for a searching problem of an architectural design. Journal of Architecture, Planning and Environmental Engineering (1989)
14. Nagasawa, I., Maeda, J., Tegoshi, Y., Makino, M.: A programming technique for some combination problems in a design support system using the method of generate-and-test. Journal of Structural and Construction Engineering (1990)
15. Umeda, M., Nagasawa, I., Higuchi, T.: The elements of programming style in design calculations. In: Proceedings of the Ninth International Conference on Industrial

and Engineering Applications of Artificial Intelligence and Expert Systems. (1996) 77–86
16. Furukawa, Y., Ueno, M., Nagasawa, I.: A health care support system. Japan Journal of Medical Informatics **10** (1990) 121–132
17. Furukawa, Y., Nagasawa, I., Ueno, M.: HCS: A health care support system. Journal of Information Processing Society of Japan **34** (1993) 88–95
18. Umeda, M., Nagasawa, I.: Project structure and development methodology toward the IT revolution – lesson from practice –. In: Proceedings of the Fourth Joint Conference on Knowledge-Based Software Engineering. (2000) 1–8
19. ISO/IEC: 13211-1 Information technology – Programming Languages – Prolog – Part 1: General core. (1995)
20. Umeda, M., Nagasawa, I., Ohno, K., Katamine, K., Takata, O.: Knowledge base development environment and J2EE-compliant inference engine for clinical decision support. In: The Proceedings of The 8th World Multi-Conference on Systemics, Cybernetics and Informatics. Volume 1. (2004) 43–48
21. Ohno, K., Umeda, M., Nagase, K., Nagasawa, I.: Knowledge base programming for medical decision support. In: The Proceedings of the 14th International Conference on Applications of Prolog. (2001) 202–210
22. Ohno, K., Nagasawa, I., Umeda, M., Nagase, K., Takada, A., Igarashi, T.: Development of medical knowledge base for clinical decision support. In: The Proceedings of The 8th World Multi-Conference on Systemics, Cybernetics and Informatics. Volume 7. (2004) 193–198
23. Nagase, K., Takada, A., Igarashi, T., Ouchi, T., Amino, T., Ohno, K.: Development and implementation of J2EE based physician order entry system with clinical decision support function. In: The Proceedings of the 23rd Joint Conference on Medical Informatics. (2003) 1–G–2–2
24. Takada, A., Nagase, K., Ouchi, T., Amino, T., Igarashi, T.: Enhanced communication realized with UML utilization in the development of hospital information system. In: The Proceedings of the 23rd Joint Conference on Medical Informatics. (2003) O–3–2
25. Ait-Kaci, H.: Warren's Abstract Machine. The MIT Press (1991)
26. Neng-Fa, Z.: Global optimizations in a prolog compiler for the TOAM. J. Logic Programming (1993) 265–294
27. Katamine, K., Hirota, T., Zhou, N.F., Nagasawa, I.: On the translation of prolog program to c. Transactions of Information Processing Society of Japan **37** (1996) 1130–1137
28. Li, X.: A new term representation method for prolog. The Journal of Logic Programming **34** (1998) 43–57
29. Umeda, M., Nagasawa, I., Ito, M.: Knowledge representation model for engineering information circulation of standard parts. Transactions of Information Processing Society of Japan **38** (1997) 1905–1918
30. Clark, K., Robinson, P., Hagen, R.: Multi-threading and message communication in Qu-Prolog. Theory and Practice of Logic Programming **1** (2001) 283–301
31. IF Computer http://www. ifcomputer .co.jp/MINERVA: MINERVA. (2005)
32. Fukuda, A.: Parallel operating systems. Journal of Information Processing Society of Japan **34** (1993) 1139–1149