ICOT Technical Report: TR-142

TR-142

Evaluation of PSI Micro-Interpreter

by
H. Nakashima (Mitsubishi Electric Corp.).
H. Nishikawa (Matsushita Research Institute).
A. Yamamoto, M. Mitsui (Oki Electric Industry).
K. Nakajima, M. Yokota, K. Taki and S. Uchida
(ICOT)

November, 1985

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

EVALUATION OF PSI MICRO-INTERPRETER

K.Nakajima*1, H.Nakashima*2, M.Yokota*1, K.Taki*1, S.Uchida*1
H.Nishikawa*3, A.Yamamoto*4, M.Mitsui*4

*1 Institute for New Generation Computer Technology (ICOT)
*2 Mitsubishi Electric Corporation
*3 Matsushita Research Institute Tokyo
*4 Oki Electric Industry Company

ABSTRACT

This paper describes the evaluation of the micro-programmed interpreter of the personal sequential inference machine PSI developed in the Japanese Fifth Generation Computer Project.

The execution speed of PSI is almost the same as that of DEC-10 Prolog on DEC 2060. Particularly PSI is a little slower with some simple benchmark programs and a little faster with the programs in which a lot of backtrackings may occur.

With some application programs, *KL0* (PSI's machine language) dynamic characteristics such as "*Backtrack Ratio*","*Predicate Call Ratio*" and "*Execution Time Ratio*" in the interpreter were also evaluated. In those programs, the call ratio of user-defined predicates is not so high (7%–30%) compared with that of builtin predicates, however, "*Execution Time Ratio*" for them is very high (46%–77%).

A *KL0* program is compiled to a table-type internal code. In order to attain flexible control and efficient execution, the interpreter executes it directly. There is another efficient way to execute a logic programming language, in which clauses are compiled to specially designed machine instructions. We have implemented it experimentally on PSI. Though the load for compilation is a little heavy, the execution speed by this implementation is about twice as fast as current one.

1 INTRODUCTION

A personal sequential inference machine PSI has been developed as part of the Fifth Generation Computer Systems (FGCS) project in Japan. PSI is a tool for software development in the project and for a research on inference machine architecture[1](Figure 1,2).

One of the main features of PSI is its high level machine language *KL0* (FGCS Kernel Language Version 0), a logic based language similar to DEC-10 Prolog[2]. A *KL0* program is compiled to a set of table-type internal codes each of which corresponds one to one to the source clauses(Figure 3). They are executed

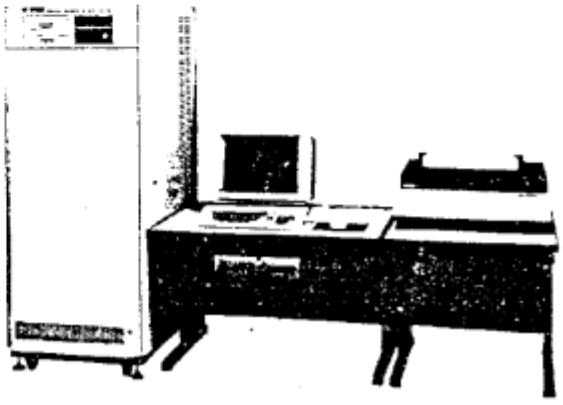

Figure 1. The appearance of PSI

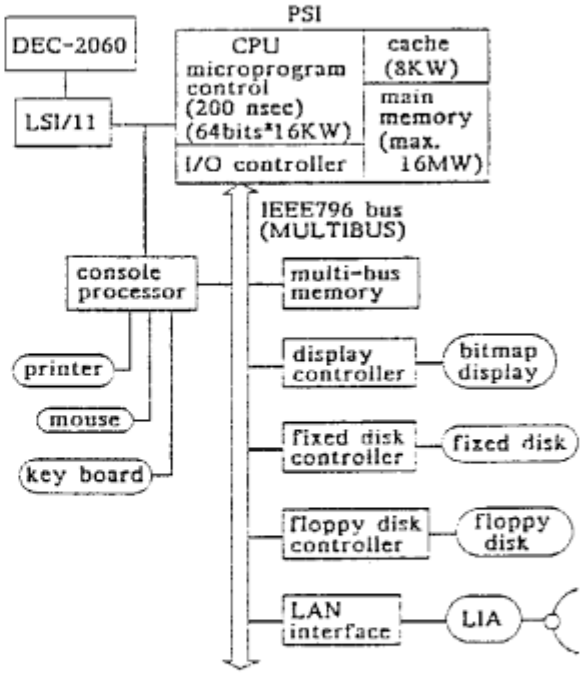

Figure 2. PSI Hardware System Configulation

directly by a microprogrammed interpreter[3]. About 160 builtin predicates are also executed by the interpreter. Extended execution control functions of *KL0* such as "remote cut" and "bind hook" are supported as the builtin predicates. The interpreter is stored in 64-bit by 16 Kw control storage.

We have completed developing the interpreter and evaluated its performance and some dynamic characteristics of *KL0* programs.

2 PSI MICRO-INTERPRETER

First of all, we give a brief explanation of the PSI interpreter.

The design of the PSI interpreter is based on the DEC-10 Prolog interpreter, but some other mechanisms are adopted in it to optimize performance. The DEC-10 Prolog interpreter[4] has 3 stacks. The first is a "*Local Stack*", used to maintain the environment for predicate call/return and backtracking. The second is a "*Global Stack*", for the information that can not be discarded even when a predicate returns determinately. The third is a "*Trail Stack*", used to "undo" the unified variables when backtracking.

For the PSI, to decrease execution overhead of inner-clause OR, we decided to divide the control information from the arguments in the environment and to maintain it in another stack, "*Control Stack*". And to improve argument handling in the process of predicate call, we introduced the "*Argument Copy*"

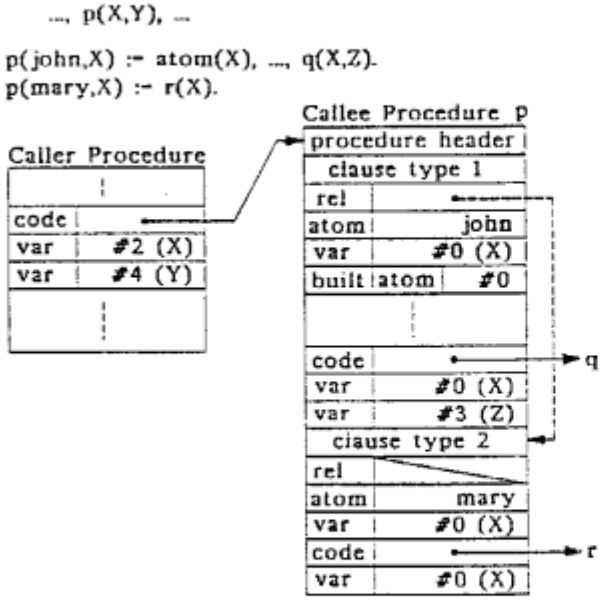

Figure 3. An Example of the Internal Code

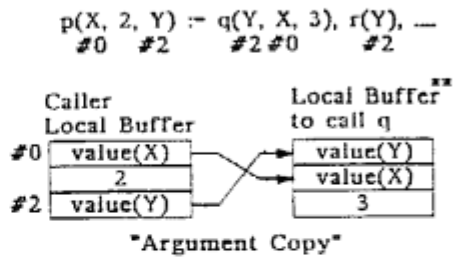

Figure 4. Argument Copy

method(Figure 4). In this method we use two "*Local Buffers*"(each is 32 w) on a register file, one to maintain the "former" environment in *Local Stack* and the other for the "new" environment. Before a "unification" in the predicate call process, the values of the caller variable arguments are copied from the current environment (which will be "former") into the "new" *Local Buffer*, and constant arguments are directly put into the "new" *Local Buffer*. By this method, the current environment can be released immediately after the *Argument Copy* operation if the predicate call is deterministic. Furthermore, even if the unification fails, the "new" *Local Buffer* can be reused for the other unification with the alternative clause, because its content is independent of callee clause arguments.

To decrease the number of the memory accesses, we also have a buffer for the *Trail Stack* on the same register file as the *Local Buffers*. This "*Trail Buffer*" (32 w) merely maintains the top part of the *Trail Stack*.

There are two major methods to treat structure data. The one is "*Structure Copying*" method which is simple and fast for small structure data. The other is "*Structure Sharing*" method which, for large structure data, requires less memory and is faster than *Structure Copying* method. As PSI should cover large application, PSI's interpreter chose *Structure Sharing* method.

The interpreter supports some time-consuming functions for PSI's operating system (SIMPOS: Sequential Inference Machine Programming and Operating System) such as memory management, process control and interrupt handling. As a builtin predicate, garbage collection is also carried out by the interpreter.

The interpreter consists of about 12 Kw: 2.5 Kw for basic control, 7.5 Kw for builtin predicates (including 1.5 Kw Garbage collector) and 2.0 Kw for others such as SIMPOS support functions mentioned above.

3 EVALUATION

3.1 Performance Comparison of PSI and DEC-10 Prolog

We evaluated the performance of the PSI interpreter with some benchmark programs like "Nreverse (Naive reverse)", "Quick sort" and "8 queens". Although the detailed specification of KL0 differs from that of DEC-10 Prolog, we compared the execution time of PSI with that of compiled code of DEC-10 Prolog on DEC 2060 as a criterion of performance.

Table 1. shows that PSI has almost the same performance as DEC-10 Prolog. However, DEC-10 Prolog is a little faster than PSI with the simple programs, especially with "Nreverse" in which optimization by compiler is rather effective. In contrast, with nondeterministic programs such as "Tree traversing" and "Reverse function", PSI is a little faster than DEC-10 Prolog. This is mainly because PSI's *Argument Copy* method is effective when backtracking occurs frequently.

3.2 *KL0* dynamic characteristics and analysis of PSI interpreter

We measured and evaluated some dynamic characteristics of practical application programs in *KL0* to extract the items that affect the execution speed. Furthermore, we analyzed the behavior of the PSI interpreter for the programs. We selected four programs;

(1) *Window Subsystem (WINDOW)* .... 2896 clauses

This program creates one window on the bitmap display, displays a character "A" on it and then deletes the window.

Test Program	PSI (msec)	DEC (msec)	DEC/PSI
Nreverse (30 elements)	13.6	9.48	0.70
Quick Sort (50 elements)	15.2	14.6	0.96
Tree Traversing (40 times cons)	51.7	61.1	1.18
Lisp Interpreter (Tarai 3)	4024	4360	1.08
Lisp Interpreter (Fibonacci 10)	369	402	1.09
Lisp Interpreter (Nreverse)	173	194	1.12
8 Queens (1 solution)	96.9	97.5	1.01
8 Queens (all solutions)	1570	1580	1.01
Reverse Function	38.2	41.7	1.09
Slow Reverse (6 elements)	99.4	89.0	0.90

Table 1. Execution Speed of Benchmark Programs

(2) *8 Puzzle (PUZZLE)* .... 97 clauses

A game to find a shortest sequence to move 8 square tiles on a 3×3 board from one fixed situation to another. The breadth-first strategy is used.

(3) *Bottom Up Parser (BUP)* .... 110 clauses

A parser for natural language understanding. It accepts a short English sentence and extracts its syntactic structure.

(4) *Harmonizer (HARMON)* .... 467 clauses

A music expert system which generates 3 part harmony suitable for a given melody using some rules of the harmony generation and progression.

Table 2. shows the dynamic characteristics of these programs. "*Backtrack Ratio*" is a frequency of unification failure in predicate calls. Except for *WINDOW*, the evaluated programs we chose are intrinsically nondeterministic in their algorithms, and their *Backtrack ratios* in user-defined predicate calls are as high as we expected.

On the other hand, *WINDOW* is an Input/Output handling programs and rather deterministic. However, its *Backtracking ratio* is not so low (14%). This is because the backtracking is used frequently for implementing conditional branches in *WINDOW*.

Backtrack Ratios in builtin predicate calls are relatively lower than that of user-defined predicates as we expected. The ratios of shallow-backtrack and deep-backtrack are different largely depending on their program logics and programming styles. The shallow-

		Window	Puzzle	Bup	Harmon
Backtrack Ratio	User Pred.	14.1%	38.9%	35.3%	44.2%
	Builtin-pred.	5.6%	2.6%	24.4%	16.8%
	Shallow-back	55.2%	61.1%	90.9%	79.0%
	Deep-back	44.8%	38.9%	9.1%	21.0%
Predicate Ratio	User Pred.	10.6%	6.9%	29.7%	25.6%
	Builtin-pred.	82.0%	93.1%	64.6%	70.3%
	Inner-cls OR	7.4%	0.0%	5.7%	4.1%
Av. head args /clause		5.00	3.80	2.86	4.53
Value of Caller args					
variable(undefined)		17.4%	25.0%	49.0%	18.0%
atomic*1		58.8%	62.5%	17.0%	62.3%
structure*2		23.8%	12.5%	34.0%	19.7%
Value of Callee args					
variable(undefined)		10.5%	46.2%	10.9%	29.1%
atomic*1		87.6%	44.8%	54.6%	61.4%
structure*2		1.9%	9.0%	34.5%	9.5%

atomic*1 : integer, atom
structure*2 : compound tem, stack vector, heap vector, string

Table 2. Dynamic Characteristics of KL0 Application Programs

backtrack ratio also depends on the clause indexing strategy in the compilation. The current compiler generates a clause indexing code if more than three indexable clauses exist. This is because the processing cost of one shallow-backtracking is almost the same as that of one clause indexing in the interpreter, and therefore, in the case of three clauses or less, average cost of shallow-backtrackings with non-indexing code seems to be less than one indexing cost.

"*Predicate Call Ratio*" is a ratio of calls of user-defined predicate, builtin predicate and inner-clause OR. *WINDOW* and *PUZZLE* use more builtin predicates than others, because *WINDOW* uses many system control builtin predicates, while in *PUZZLE* the positions of tiles are calculated by arithmetic and logical builtin predicates.

Average numbers of head arguments are all less than five, so the size of *Local Buffer* (32 w) seems to be sufficient (there was no case to exceed the *Local Buffer* size in fact).

Value types of caller and callee arguments in the unification were also evaluated. The ratios of atomic values of both caller and callee are very high. In *Argument copy* operation, if the caller argument is a variable, a pointer(instead of its value) to that variable cell should be put into the "new" *Local Buffer* with extra steps. Except for *BUP*, these cases are not so frequent.

Table 3. is the "*Execution Time Ratio*" in each submodule of the interpreter. They were measured by counting micro execution steps in each submodules. Although the ratio of user-defined predicate calls is less than 30% in all programs, the execution time for them(Control+Unify+Trail) are large(46%–77%). This result suggests us that there is a possibility of improving the performance more than 20% by rewriting the critical parts of the interpreter.

Submodule	Control	Unify	Trail	Cut	Get-arg	Built
Window	31.3%	17.1%	2.0%	10.0%	13.6%	26.2%
Puzzle	27.5%	11.0%	7.5%	0.0%	22.7%	31.3%
Bup	22.3%	43.0%	4.7%	5.6%	5.2%	19.2%
Harmon	25.5%	46.4%	5.4%	4.0%	7.3%	11.0%

Control : Call/Return/Backtrack
Unify : Unification/Argument Copy
Trail : Trailing
Cut : Cut
Get-arg : Argument Preparation for Builtin-predicate
Built : Builtin-predicate

Table 3. Execution Time Ratio of Submodules in PSI Interpreter

4 ANOTHER IMPLEMENTATION

Generally, there are three approaches to implement a high-level language processor. The first is the simple compilation to a machine instruction set similar to its high-level language. The firmware interpreter can attain relatively high performance for any kind of operation. The second is the compilation with medium optimization to a specially designed high-level machine instruction set executed by a firmware emulator. The third is the compilation with heavy optimization to a reduced instruction set that can be executed quickly by simple firmware, or directly by hardware (RISC method).

The design of PSI interpreter is based on the first approach mainly because we considered it most easy to implement the extended control feature of *KL0* flexibly and efficiently. And common optimization technique by compiler also introduced such as clause indexing, compact code generation and classification of variables to *first* and *second occurrence*. We have attained the expected performance "comparable to DEC-10 Prolog". However, it is valuable to evaluate the other approach, so we made an experiment on the second approach.

D.H.D.Warren[5] proposed a virtual Prolog machine that executes a high-level stack oriented instruction set. We added some instructions to it such as "cut" to keep the *KL0* specification, and we emulated the instructions by firmware on PSI. The number of stacks was three, and the manipulation method for structure data was *Structure Copying* as Warren's method. We evaluated the performance with the benchmark programs used in 3.1.

Table 4. shows that its performance is about twice as fast as the current PSI interpreter.

Clearly their instruction set is rather effective when optimization by compiler is fully attained. This

Test Program	PSI* (msec)	PSI (msec)	PSI/PSI*
Nreverse (30 elements)	4.35	13.6	3.13
Quick Sort (50 elements)	6.73	15.2	2.26
Tree Traversing (40 times cons)	28.95	51.7	1.79
8 Queens (1 solution)	48.10	96.9	2.01
Reverse Function	22.31	38.2	1.71
Slow Reverse (5 elements)	10.31	24.8	2.41

PSI* : Prolog Instruction Set Emulation

Table 4. Execution Speed by Prolog Instruction Set Emulation

is because their instructions correspond to each argument in unification, and thus, the number of the instructions passing the arguments from caller to callee can be reduced by skillful allocation of arguments to temporary registers[5].

Furthermore, by reducing the number of stacks and by *Structure Copying* method, the emulation firmware becomes so simple that we can save the microprogram steps statically and dynamically.

We are sure that the difference in performance between the emulation and the current PSI interpreter should be smaller. Because practical programs tend to use many builtin predicates and use large structure data frequently. Still, this approach is reasonable to attain appropriate load balancing between compiler and firmware in executing Prolog or *KL0* on a sequential machine.

5 CONCLUSION

We have developed the PSI interpreter and evaluated it. The performance in execution speed is comparable to DEC-10 Prolog on DEC 2060, especially better for nondeterministic programs. The critical parts in the execution time are the operations for user-defined predicates in the interpreter.

Using these evaluation results, we are now designing the new version of PSI based on the high-level Prolog instruction set. Most of our interest is to achieve the complete set of *KL0* functions in high performance.

REFERENCES

[1]: Taki,K.,et.al., "*Hardware Design and Implementation of the Personal Sequential Inference Machine(PSI)*", Proc. of FGCS'84, 1984.

[2]: Pereira,L.M.,et.al., "*User's Guide to DECsystem-10 Prolog*",Dept. of AI, Univ. of Edinburgh, 1978.

[3]: Yokota,M.,et.al., "*A Microprogrammed Interpreter for the Personal Sequential Inference Machine*", Proc. of FGCS'84, 1984.

[4]: Warren,D.H.D., "*Implementing Prolog - Compiling Predicate Logic Program*", D.A.I.Research Report, No.39-40, Dept. of AI, Univ. of Edinburgh, 1977.

[5]: Warren,D.H.D., "*An Abstract Prolog Instruction Set*", Technical Note 309, AI Center, SRI International,1983.